JAKARTA EE

Jakarta Persistence

Jakarta Persistence Team, https://projects.eclipse.org/projects/ee4j.jpa

4.0-M4, May 25, 2026: DRAFT

Table of Contents

Eclipse Foundation Specification License - v1.1 . . . 2
Disclaimers . . . 2
1. Introduction . . . 3
1.1. Authorship . . . 3
1.2. Document Conventions . . . 3
2. Entities . . . 4
2.1. The Entity Class . . . 4
2.2. Persistent Fields and Properties . . . 4
2.2.1. Persistent Attribute Type . . . 5
2.2.2. Value Types and Reference Types . . . 6
2.2.3. Property Access . . . 7
2.3. Access Type . . . 8
2.3.1. Default Access Type . . . 8
2.3.2. Explicit Access Type . . . 9
2.3.3. Access Type of an Embeddable Class . . . 10
2.3.4. Defaulted Access Types of Embeddable Classes and Mapped Superclasses . . . 10
2.4. Primary Keys and Entity Identity . . . 10
2.4.1. Composite Primary Keys . . . 11
2.4.2. Primary Keys Involving Derived Identities . . . 12
2.4.2.1. Use of `MapsId` to Represent Derived Identity . . . 12
2.4.2.2. Specification of Derived Identities . . . 13
2.4.2.3. Mapping of Derived Identities . . . 14
2.4.2.4. Examples of Derived Identities . . . 14
2.5. Entity Versions . . . 23
2.6. Basic Types . . . 24
2.7. Embeddable Classes . . . 24
2.8. Collections of Embeddable Classes and Basic Types . . . 25
2.9. Map Collections . . . 25
2.9.1. Map Values . . . 25
2.9.2. Map Keys . . . 26
2.10. Mapping Defaults for Non-Relationship Fields or Properties . . . 26
2.11. Entity Relationships . . . 27
2.12. Relationship Mapping Defaults . . . 29
2.12.1. Bidirectional OneToOne Relationships . . . 29
2.12.2. Bidirectional ManyToOne / OneToMany Relationships . . . 30
2.12.3. Unidirectional Single-Valued Relationships . . . 31
2.12.3.1. Unidirectional OneToOne Relationships . . . 31
2.12.3.2. Unidirectional ManyToOne Relationships . . . 32
2.12.4. Bidirectional ManyToMany Relationships . . . 33
2.12.5. Unidirectional Multi-Valued Relationships . . . 34
2.12.5.1. Unidirectional OneToMany Relationships . . . 34
2.12.5.2. Unidirectional ManyToMany Relationships . . . 35
2.13. Inheritance . . . 36
2.13.1. Abstract Entity Classes . . . 37

2.13.2. Mapped Superclasses . . . 37
2.13.3. Non-Entity Classes in the Entity Inheritance Hierarchy . . . 39
2.14. Inheritance Mapping Strategies . . . 40
2.14.1. Single Table per Class Hierarchy Strategy . . . 40
2.14.2. Joined Subclass Strategy . . . 41
2.14.3. Table per Concrete Class Strategy . . . 41
2.14.4. Duplicate Attributes in Inheritance Hierarchies . . . 41
2.14.5. Duplicate Column Mappings . . . 42
2.15. Naming of Database Objects . . . 43
3. Entity Operations . . . 47
3.1. Overview . . . 47
3.2. Entities and Database Records . . . 47
3.3. Properties, Hints, and Options . . . 48
3.4. EntityHandler Interface . . . 48
3.5. EntityManager Interface . . . 49
3.6. Entity Instance Lifecycle . . . 50
3.6.1. Lifecycle Operations . . . 50
3.6.2. Entity Instance Creation . . . 51
3.6.3. Persisting an Entity Instance . . . 51
3.6.4. Removal . . . 51
3.6.5. Orphan Removal . . . 52
3.6.6. Synchronization to the Database . . . 52
3.6.7. Refreshing an Entity Instance . . . 53
3.6.8. Evicting an Entity Instance from the Persistence Context . . . 54
3.6.9. Detached Entities . . . 54
3.6.9.1. Detached Entities and Lazy Loading . . . 54
3.6.9.2. Merging Detached Entity State . . . 55
3.6.9.3. Detached Entities and Serialization . . . 56
3.6.10. Managed Entities . . . 56
3.6.11. Entity Instance Retrieval . . . 56
3.6.12. Load State . . . 57
3.7. Persistence Context Lifetime and Transaction Synchronization . . . 58
3.7.1. Persistence Context Types . . . 58
3.7.2. Entity Manager Operations and Transactions . . . 60
3.7.3. Synchronization with the Current Transaction . . . 61
3.7.4. Transaction Commit . . . 62
3.7.5. Transaction Rollback . . . 62
3.8. EntityAgent interface . . . 63
3.9. Locking and Concurrency . . . 63
3.9.1. Optimistic Locking . . . 64
3.9.2. Entity Versions and Optimistic Locking . . . 65
3.9.3. Pessimistic Locking . . . 65
3.9.4. Lock Modes . . . 66
3.9.4.1. OPTIMISTIC, OPTIMISTIC_FORCE_INCREMENT . . . 66
3.9.4.2. PESSIMISTIC_READ, PESSIMISTIC_WRITE, PESSIMISTIC_FORCE_INCREMENT . . . 67
3.9.4.3. Lock Mode Properties and Uses . . . 69

3.9.5. OptimisticLockException . . . 69
3.10. Entity Listeners and Callback Methods . . . 70
3.10.1. Lifecycle Callback Methods . . . 71
3.10.2. Semantics of the Lifecycle Callback Methods for Entities . . . 72
3.10.3. Multiple Lifecycle Callback Methods for an Entity Lifecycle Event . . . 74
3.10.4. Exceptions . . . 77
3.10.5. Entity Listeners and Dependency Injection . . . 77
3.10.6. Lifecycle Callbacks, Transactions, and Security . . . 77
3.10.7. Specification of Callback Listener Classes and Lifecycle Methods in the XML Descriptor . . . 77
3.10.7.1. Specification of Callback Listeners . . . 77
3.10.7.2. Specification of the Binding of Entity Listener Classes to Entities . . . 77
3.11. Jakarta Validation integration . . . 78
3.11.1. Configuring the Jakarta Validation provider . . . 78
3.11.1.1. Providing the ValidatorFactory . . . 78
3.11.1.2. Enabling/Disabling the Jakarta Validation integration . . . 79
3.11.2. Automatic Validation Upon Lifecycle Events . . . 79
3.11.2.1. Requirements for Automatic Validation upon Lifecycle Events . . . 79
3.12. Entity Graphs . . . 80
3.12.1. Use of Entity Graphs in find and query operations . . . 81
3.12.1.1. Fetch Graph Semantics . . . 81
3.12.1.2. Load Graph Semantics . . . 84
3.13. Type Conversion of Basic Attributes . . . 86
3.13.1. Auto-Apply Converters . . . 86
3.13.2. Explicit Converters . . . 86
3.13.3. Converter Invocation . . . 87
3.13.4. Attribute Converters and Dependency Injection . . . 87
3.14. Second-Level Cache . . . 87
3.14.1. The Shared Cache Mode and Cacheable Annotation . . . 88
3.14.2. Cache Modes . . . 88
3.14.3. Cache Interface . . . 91
3.15. Query APIs . . . 91
3.15.1. Query Execution . . . 91
3.15.2. Query Execution and Transactions . . . 92
3.15.3. Queries and the Flush Mode . . . 93
3.15.4. Queries and the Lock Mode . . . 96
3.15.5. Query Hints and Options . . . 96
3.15.6. Parameter Objects . . . 97
3.15.7. Named Parameters . . . 97
3.15.8. Positional Parameters . . . 98
3.15.9. Arguments to Query Parameters . . . 98
3.15.10. Result Classes for Jakarta Persistence Queries . . . 98
3.15.11. Named Queries and Named Statements . . . 98
3.15.12. Polymorphic Queries . . . 100
3.15.13. SQL Queries . . . 100
3.15.13.1. Returning Managed Entities from Native Queries . . . 100
3.15.13.2. Returning Unmanaged Instances . . . 103

3.15.13.3. Annotated Result Set Mappings . . . 104
3.15.13.4. Programmatic Result Set Mappings . . . 104
3.15.14. Result Classes for SQL Queries . . . 106
3.15.14.1. Restrictions . . . 106
3.15.15. Stored Procedures . . . 106
3.15.15.1. Named Stored Procedure Queries . . . 106
3.15.15.2. Dynamically-specified Stored Procedure Queries . . . 107
3.15.15.3. Stored Procedure Query Execution . . . 107
3.16. Summary of Exceptions . . . 108
4. Query Language . . . 112
4.1. Overview . . . 112
4.2. Statement Types . . . 112
4.2.1. Select Statements . . . 113
4.2.1.1. Set Operators in Select Statements . . . 113
4.2.2. Update and Delete Statements . . . 113
4.3. Abstract Schema Types and Query Domains . . . 114
4.3.1. Naming . . . 114
4.3.2. Example . . . 114
4.4. The FROM Clause and Navigational Declarations . . . 115
4.4.1. Identifiers . . . 116
4.4.2. Identification Variables . . . 117
4.4.3. Range Variable Declarations . . . 118
4.4.4. Path Expressions . . . 118
4.4.4.1. Path Expression Syntax . . . 120
4.4.5. Joins . . . 121
4.4.5.1. Inner Joins . . . 122
4.4.5.2. Outer Joins . . . 123
4.4.5.3. Fetch Joins . . . 125
4.4.6. Collection Member Declarations . . . 125
4.4.7. FROM Clause and SQL . . . 126
4.4.8. Polymorphism . . . 126
4.4.9. Downcasting . . . 126
4.5. WHERE Clause . . . 127
4.6. Conditional Expressions . . . 127
4.6.1. Conditional Expression Composition . . . 127
4.6.2. Operators and Operator Precedence . . . 128
4.6.3. Comparison Expressions . . . 128
4.6.4. Between Expressions . . . 128
4.6.5. In Expressions . . . 129
4.6.6. Like Expressions . . . 130
4.6.7. Null Comparison Expressions . . . 130
4.6.8. Empty Collection Comparison Expressions . . . 130
4.6.9. Collection Member Expressions . . . 131
4.6.10. Exists Expressions . . . 131
4.6.11. All or Any Expressions . . . 132
4.6.12. Subqueries . . . 132

4.6.13. Null Values . . . 133
4.6.14. Equality and Comparison Semantics . . . 134
4.6.14.1. Queries Using Input Parameters . . . 136
4.7. Scalar Expressions . . . 136
4.7.1. Literals . . . 136
4.7.2. Identification Variables . . . 137
4.7.3. Path Expressions . . . 137
4.7.4. Input Parameters . . . 137
4.7.4.1. Positional Parameters . . . 137
4.7.4.2. Named Parameters . . . 138
4.7.5. Arithmetic Expressions . . . 138
4.7.6. String concatenation operator . . . 138
4.7.7. Built-in String, Arithmetic, and Datetime Functional Expressions . . . 138
4.7.7.1. String Functions . . . 138
4.7.7.2. Arithmetic Functions . . . 139
4.7.7.3. Datetime Functions . . . 140
4.7.8. Typecasts . . . 141
4.7.9. Invocation of Predefined and User-defined Database Functions . . . 141
4.7.10. Case Expressions . . . 142
4.7.11. Identifier and Version Functions . . . 143
4.7.12. Entity Type Expressions and Literal Entity Types . . . 143
4.7.13. Numeric Expressions and Type Promotion . . . 144
4.8. GROUP BY, HAVING . . . 145
4.9. SELECT Clause . . . 146
4.9.1. Result Type of the SELECT Clause . . . 147
4.9.2. Constructor Expressions in the SELECT Clause . . . 148
4.9.3. Null Values in the Query Result . . . 148
4.9.4. Embeddables in the Query Result . . . 148
4.9.5. Aggregate Functions in the SELECT Clause . . . 149
4.10. ORDER BY Clause . . . 150
4.11. Bulk Update and Delete Operations . . . 152
4.12. BNF . . . 153
5. Metamodel API . . . 160
5.1. Static Metamodel Classes . . . 160
5.1.1. Bootstrapping the Static Metamodel . . . 160
5.2. Canonical Metamodel . . . 160
5.2.1. Canonical Metamodel Classes . . . 161
5.2.2. Canonical Metamodel for Managed Classes . . . 161
5.2.3. Canonical Metamodel for Eligible Classes . . . 162
5.2.4. Example Canonical Metamodel . . . 163
6. Criteria API . . . 165
6.1. Overview . . . 165
6.2. Criteria Query API Usage . . . 165
6.3. Constructing Criteria Queries . . . 166
6.3.1. CriteriaQuery Creation . . . 166
6.3.2. Query Roots . . . 166

6.3.3. Joins . . . 167
6.3.4. Fetch Joins . . . 168
6.3.5. Path Navigation . . . 169
6.3.6. Restricting the Query Result . . . 170
6.3.7. Downcasting . . . 170
6.3.8. Expressions . . . 171
6.3.8.1. Result Types of Expressions . . . 173
6.3.9. Literals . . . 174
6.3.10. Parameter Expressions . . . 175
6.3.11. Specifying the Select List . . . 175
6.3.11.1. Assigning Aliases to Selection Items . . . 177
6.3.12. Subqueries . . . 177
6.3.13. GroupBy and Having . . . 180
6.3.14. Ordering the Query Results . . . 180
6.3.15. Bulk Update and Delete Operations . . . 182
6.4. Constructing Strongly-typed Queries using the jakarta.persistence.metamodel Interfaces . . . 184
6.5. Use of the Criteria API with Strings to Reference Attributes . . . 184
6.6. Query Modification . . . 186
6.7. Query Execution . . . 186
7. Entity Managers and Persistence Contexts . . . 188
7.1. Persistence Contexts . . . 188
7.2. Obtaining an EntityManager . . . 188
7.2.1. Obtaining an Entity Manager in the Jakarta EE Environment . . . 188
7.2.2. Obtaining an Entity Agent in the Jakarta EE Environment . . . 189
7.2.3. Obtaining an Application-managed Entity Manager or Entity Agent . . . 190
7.3. Obtaining an Entity Manager Factory . . . 190
7.3.1. Obtaining an Entity Manager Factory in a Jakarta EE Container . . . 190
7.3.2. Obtaining an Entity Manager Factory in a Java SE Environment . . . 190
7.3.3. Obtaining an Entity Manager Factory for a programmatically-defined persistence unit . . . 190
7.4. EntityManagerFactory Interface . . . 191
7.5. Controlling Transactions . . . 191
7.5.1. The EntityTransaction Interface . . . 192
7.6. The runInTransaction and callInTransaction methods . . . 192
7.7. Container-managed Persistence Contexts . . . 193
7.7.1. Persistence Context Synchronization Type . . . 193
7.7.2. Container-managed Transaction-scoped Persistence Context . . . 194
7.7.3. Container-managed Extended Persistence Context . . . 194
7.7.3.1. Inheritance of Extended Persistence Context . . . 195
7.7.4. Persistence Context Propagation . . . 195
7.7.4.1. Requirements for Persistence Context Propagation . . . 195
7.8. Application-managed Persistence Contexts . . . 197
7.9. Requirements on the Container . . . 200
7.9.1. Application-managed Persistence Contexts . . . 200
7.9.2. Container Managed Persistence Contexts . . . 200
7.10. Runtime Contracts between the Container and Persistence Provider . . . 201
7.10.1. Container Responsibilities . . . 201

7.10.2. Provider Responsibilities . . . 202
7.11. PersistenceUnitUtil Interface . . . 202
7.12. SchemaManager Interface . . . 202
8. Entity Packaging . . . 204
8.1. Persistence Unit Packaging . . . 204
8.2. persistence.xml file . . . 205
8.2.1. Elements of persistence.xml . . . 205
8.2.1.1. name . . . 206
8.2.1.2. transaction-type . . . 206
8.2.1.3. description . . . 206
8.2.1.4. provider . . . 206
8.2.1.5. qualifier . . . 207
8.2.1.6. scope . . . 207
8.2.1.7. jta-data-source, non-jta-data-source . . . 207
8.2.1.8. mapping-file, jar-file, class, exclude-unlisted-classes . . . 207
8.2.1.9. shared-cache-mode . . . 210
8.2.1.10. validation-mode . . . 210
8.2.1.11. default-to-one-fetch-type . . . 211
8.2.1.12. properties . . . 211
8.2.2. Persistence Unit Properties . . . 211
8.2.3. Example Persistence Unit Configurations . . . 214
8.3. Persistence Unit Scope . . . 215
8.4. persistence.xml Schema . . . 216
9. Container and Provider Contracts for Deployment and Bootstrapping . . . 223
9.1. Jakarta EE Deployment . . . 223
9.1.1. Integration with JNDI . . . 223
9.2. Bootstrapping in Java SE Environments . . . 224
9.2.1. Schema Generation . . . 225
9.3. Determining the Available Persistence Providers . . . 225
9.4. Schema Generation . . . 226
9.4.1. Properties for Schema Generation . . . 226
9.4.2. Data Loading . . . 228
9.5. Responsibilities of the Persistence Provider . . . 229
9.5.1. jakarta.persistence.spi.PersistenceProvider . . . 229
9.5.2. jakarta.persistence.spi.ProviderUtil . . . 230
9.6. jakarta.persistence.spi.PersistenceUnitInfo Interface . . . 230
9.6.1. jakarta.persistence.spi.ClassTransformer Interface . . . 232
9.7. jakarta.persistence.Persistence Class . . . 232
9.7.1. Properties for Java SE Bootstrap . . . 232
9.8. jakarta.persistence.PersistenceConfiguration Class . . . 235
9.9. PersistenceUtil Interface . . . 235
9.9.1. Contracts for Determining the Load State of an Entity or Entity Attribute . . . 235
9.10. Integration with CDI . . . 236
10. Metadata Annotations . . . 239
10.1. Entity . . . 239
10.2. Entity Listeners and Callback Annotations . . . 239

10.2.1. `EntityListener` and `EntityListeners` . . . 239
10.2.2. Callback methods . . . 240
10.3. EntityGraph Annotations . . . 241
10.3.1. NamedEntityGraph and NamedEntityGraphs Annotations . . . 241
10.3.2. NamedAttributeNode Annotation . . . 242
10.3.3. NamedSubgraph Annotation . . . 242
10.3.4. Fetch Annotation . . . 243
10.4. Annotations for Queries . . . 244
10.4.1. NamedQuery Annotation . . . 244
10.4.2. NamedStatement Annotation . . . 247
10.4.3. NamedNativeQuery Annotation . . . 247
10.4.4. NamedNativeStatement Annotation . . . 251
10.4.5. NamedStoredProcedureQuery Annotation . . . 251
10.4.6. Annotations for SQL Result Set Mappings . . . 254
10.5. References to EntityManager, EntityAgent, and EntityManagerFactory . . . 257
10.5.1. PersistenceContext Annotation . . . 257
10.5.2. PersistenceAgent Annotation . . . 258
10.5.3. PersistenceUnit Annotation . . . 259
10.6. Annotations for Attribute Converter Classes . . . 259
10.7. Annotations for Static Queries . . . 260
10.7.1. JakartaQuery Annotation . . . 260
10.7.2. NativeQuery Annotation . . . 260
10.7.3. QueryOptions Annotation . . . 261
11. Metadata for Object/Relational Mapping . . . 262
11.1. Fetching Policies . . . 262
11.2. Annotations for Object/Relational Mapping . . . 262
11.2.1. Access Annotation . . . 262
11.2.2. AssociationOverride Annotation . . . 263
11.2.3. AssociationOverrides Annotation . . . 265
11.2.4. AttributeOverride Annotation . . . 267
11.2.5. AttributeOverrides Annotation . . . 269
11.2.6. Basic Annotation . . . 269
11.2.7. Cacheable Annotation . . . 271
11.2.8. CheckConstraint Annotation . . . 271
11.2.9. CollectionTable Annotation . . . 272
11.2.10. Column Annotation . . . 275
11.2.11. Convert Annotation . . . 279
11.2.12. Converts Annotation . . . 282
11.2.13. DiscriminatorColumn Annotation . . . 282
11.2.14. DiscriminatorValue Annotation . . . 284
11.2.15. ElementCollection Annotation . . . 285
11.2.16. Embeddable Annotation . . . 286
11.2.17. Embedded Annotation . . . 287
11.2.18. EmbeddedId Annotation . . . 288
11.2.19. Enumerated Annotation . . . 289
11.2.20. EnumeratedValue Annotation . . . 290

11.2.21. ExcludedFromVersioning Annotation . . . 291
11.2.22. ForeignKey Annotation . . . 291
11.2.23. GeneratedValue Annotation . . . 292
11.2.24. Id Annotation . . . 293
11.2.25. IdClass Annotation . . . 294
11.2.26. Index Annotation . . . 295
11.2.27. Inheritance Annotation . . . 296
11.2.28. JoinColumn Annotation . . . 296
11.2.29. JoinColumns Annotation . . . 301
11.2.30. JoinTable Annotation . . . 302
11.2.31. Lob Annotation . . . 304
11.2.32. ManyToMany Annotation . . . 304
11.2.33. ManyToOne Annotation . . . 307
11.2.34. MapKey Annotation . . . 310
11.2.35. MapKeyClass Annotation . . . 311
11.2.36. MapKeyColumn Annotation . . . 312
11.2.37. MapKeyEnumerated Annotation . . . 316
11.2.38. MapKeyJoinColumn Annotation . . . 317
11.2.39. MapKeyJoinColumns Annotation . . . 321
11.2.40. MapKeyTemporal Annotation . . . 321
11.2.41. MappedSuperclass Annotation . . . 322
11.2.42. MapsId Annotation . . . 322
11.2.43. OneToMany Annotation . . . 323
11.2.44. OneToOne Annotation . . . 326
11.2.45. OrderBy Annotation . . . 330
11.2.46. OrderColumn Annotation . . . 331
11.2.47. PrimaryKeyJoinColumn Annotation . . . 333
11.2.48. PrimaryKeyJoinColumns Annotation . . . 335
11.2.49. SecondaryTable Annotation . . . 337
11.2.50. SecondaryTables Annotation . . . 339
11.2.51. SequenceGenerator Annotation . . . 339
11.2.52. SequenceGenerators Annotation . . . 341
11.2.53. Table Annotation . . . 341
11.2.54. TableGenerator Annotation . . . 343
11.2.55. TableGenerators Annotation . . . 345
11.2.56. Temporal Annotation . . . 345
11.2.57. Transient Annotation . . . 346
11.2.58. UniqueConstraint Annotation . . . 347
11.2.59. Version Annotation . . . 347
11.3. Object/Relational Metadata Used in Schema Generation . . . 348
11.3.1. Table-level elements . . . 348
11.3.1.1. Table . . . 349
11.3.1.2. Inheritance . . . 349
11.3.1.3. SecondaryTable . . . 349
11.3.1.4. CollectionTable . . . 349
11.3.1.5. JoinTable . . . 349

11.3.1.6. TableGenerator . . . 349
11.3.2. Column-level elements . . . 349
11.3.2.1. Column . . . 350
11.3.2.2. MapKeyColumn . . . 350
11.3.2.3. Enumerated, MapKeyEnumerated . . . 350
11.3.2.4. Temporal, MapKeyTemporal . . . 351
11.3.2.5. Lob . . . 351
11.3.2.6. OrderColumn . . . 351
11.3.2.7. DiscriminatorColumn . . . 351
11.3.2.8. Version . . . 351
11.3.3. Primary Key mappings . . . 351
11.3.3.1. Id . . . 351
11.3.3.2. EmbeddedId . . . 352
11.3.3.3. GeneratedValue . . . 352
11.3.4. Foreign Key Column Mappings . . . 352
11.3.4.1. JoinColumn . . . 352
11.3.4.2. MapKeyJoinColumn . . . 353
11.3.4.3. PrimaryKeyJoinColumn . . . 353
11.3.4.4. ForeignKey . . . 353
11.3.5. Other Elements . . . 353
11.3.5.1. SequenceGenerator . . . 353
11.3.5.2. Index . . . 353
11.3.5.3. UniqueConstraint . . . 354
11.4. Examples of the Application of Annotations for Object/Relational Mapping . . . 354
11.5. Derivation of schema metadata from Jakarta Validation constraints . . . 361
12. XML Object/Relational Mapping Descriptor . . . 366
12.1. Use of the XML Descriptor . . . 366
12.2. XML Overriding Rules . . . 366
12.2.1. persistence-unit-defaults Subelements . . . 366
12.2.1.1. schema . . . 366
12.2.1.2. catalog . . . 367
12.2.1.3. delimited-identifiers . . . 367
12.2.1.4. access . . . 367
12.2.1.5. cascade-persist . . . 367
12.2.1.6. entity-listeners . . . 367
12.2.2. Other Subelements of the entity-mappings element . . . 368
12.2.2.1. package . . . 368
12.2.2.2. schema . . . 368
12.2.2.3. catalog . . . 368
12.2.2.4. access . . . 368
12.2.2.5. sequence-generator . . . 368
12.2.2.6. table-generator . . . 368
12.2.2.7. named-query, named-native-query . . . 369
12.2.2.8. named-statement, named-native-statement . . . 369
12.2.2.9. named-stored-procedure-query . . . 369
12.2.2.10. sql-result-set-mapping . . . 369

12.2.2.11. entity . . . 369
12.2.2.12. mapped-superclass . . . 370
12.2.2.13. embeddable . . . 370
12.2.2.14. converter . . . 370
12.2.3. entity Subelements and Attributes . . . 370
12.2.3.1. metadata-complete . . . 370
12.2.3.2. access . . . 370
12.2.3.3. cacheable . . . 370
12.2.3.4. name . . . 371
12.2.3.5. table . . . 371
12.2.3.6. secondary-table . . . 371
12.2.3.7. primary-key-join-column . . . 371
12.2.3.8. id-class . . . 371
12.2.3.9. inheritance . . . 371
12.2.3.10. discriminator-value . . . 371
12.2.3.11. discriminator-column . . . 371
12.2.3.12. sequence-generator . . . 372
12.2.3.13. table-generator . . . 372
12.2.3.14. attribute-override . . . 372
12.2.3.15. association-override . . . 372
12.2.3.16. convert . . . 372
12.2.3.17. named-entity-graph . . . 372
12.2.3.18. named-query . . . 372
12.2.3.19. named-native-query . . . 373
12.2.3.20. named-stored-procedure-query . . . 373
12.2.3.21. sql-result-set-mapping . . . 373
12.2.3.22. exclude-default-listeners . . . 373
12.2.3.23. exclude-superclass-listeners . . . 373
12.2.3.24. entity-listeners . . . 374
12.2.3.25. pre-* and post-* . . . 374
12.2.3.26. attributes . . . 374
12.2.4. mapped-superclass Subelements and Attributes . . . 375
12.2.4.1. metadata-complete . . . 375
12.2.4.2. access . . . 376
12.2.4.3. id-class . . . 376
12.2.4.4. exclude-default-listeners . . . 376
12.2.4.5. exclude-superclass-listeners . . . 376
12.2.4.6. entity-listeners . . . 376
12.2.4.7. pre-* and post-* . . . 376
12.2.4.8. attributes . . . 376
12.2.5. embeddable Subelements and Attributes . . . 378
12.2.5.1. metadata-complete . . . 378
12.2.5.2. access . . . 378
12.2.5.3. attributes . . . 378
12.3. XML Schema . . . 379
Related Documents . . . 423

Appendix A: Revision History . . . 424
A.1. Jakarta Persistence 4.0 . . . 424
A.1.1. Changes to API . . . 424
A.1.2. Changes to Query Language . . . 425
A.1.3. Changes to Configuration and Bootstrap . . . 425
A.1.4. Minor Changes and Fixes . . . 426
A.1.5. Deprecations . . . 426
A.1.6. Removals . . . 427
A.2. Jakarta Persistence 3.2 . . . 427
A.2.1. Deprecations . . . 429
A.2.2. Deprecations for removal . . . 429
A.3. Jakarta Persistence 3.1 . . . 429
A.4. Jakarta Persistence 3.0 . . . 430
A.5. Java Persistence 2.2 (Maintenance Release Draft) . . . 430
Appendix B: Persistence API Interfaces . . . 432
B.1. `EntityHandler` . . . 432
B.2. `EntityManager` . . . 453
B.3. `EntityAgent` . . . 467
B.4. `EntityTransaction` . . . 473
B.5. `EntityManagerFactory` . . . 474
B.6. `LockModeType` . . . 485
B.7. `PessimisticLockScope` . . . 488
B.8. `Cache` . . . 490
B.9. `Query` . . . 491
B.10. `Statement` . . . 506
B.11. `TypedQuery` . . . 514
B.12. `StoredProcedureQuery` . . . 527
B.13. `Tuple` . . . 542
B.14. `TupleElement` . . . 544
B.15. `Parameter` . . . 544
B.16. `Graph` . . . 545
B.17. `EntityGraph` . . . 554
B.18. `Subgraph` . . . 555
B.19. `AttributeNode` . . . 556
B.20. `AttributeConverter` . . . 558
B.21. `SchemaManager` . . . 559
B.22. `Persistence` . . . 560
B.23. `PersistenceConfiguration` . . . 569
B.24. `PersistenceUtil` . . . 576
B.25. `PersistenceUnitUtil` . . . 577
B.26. `Reference` . . . 579
B.27. `TypedQueryReference` . . . 581
B.28. `StatementReference` . . . 583
Appendix C: Criteria API Interfaces . . . 585
C.1. `CriteriaBuilder` . . . 585
C.2. `CriteriaQuery` . . . 622

C.3. CriteriaSelect . . . 627
C.4. CriteriaStatement . . . 628
C.5. CriteriaUpdate . . . 628
C.6. CriteriaDelete . . . 630
C.7. AbstractQuery . . . 631
C.8. CollectionJoin . . . 635
C.9. CommonAbstractCriteria . . . 636
C.10. CompoundSelection . . . 637
C.11. Expression . . . 637
C.12. Fetch . . . 641
C.13. FetchParent . . . 641
C.14. AbstractQuery . . . 643
C.15. Join . . . 649
C.16. JoinType . . . 651
C.17. ListJoin . . . 651
C.18. LocalDateField . . . 653
C.19. LocalDateTimeField . . . 653
C.20. LocalTimeField . . . 655
C.21. MapJoin . . . 655
C.22. Nulls . . . 657
C.23. Order . . . 657
C.24. ParameterExpression . . . 658
C.25. Path . . . 658
C.26. PluralJoin . . . 661
C.27. Predicate . . . 662
C.28. Root . . . 662
C.29. Selection . . . 663
C.30. SetJoin . . . 664
C.31. Subquery . . . 665
C.32. TemporalField . . . 670
Appendix D: Metamodel API Interfaces . . . 671
D.1. Metamodel . . . 671
D.2. StaticMetamodel . . . 672
D.3. Attribute . . . 673
D.4. BasicType . . . 675
D.5. Bindable . . . 675
D.6. CollectionAttribute . . . 676
D.7. EmbeddableType . . . 676
D.8. EntityType . . . 676
D.9. IdentifiableType . . . 677
D.10. ListAttribute . . . 679
D.11. ManagedType . . . 679
D.12. MapAttribute . . . 685
D.13. MappedSuperclassType . . . 685
D.14. PluralAttribute . . . 685
D.15. SetAttribute . . . 686

D.16. SingularAttribute . . . 686
D.17. Type . . . 687
Appendix E: Persistence SPI Interfaces . . . 689
E.1. ClassTransformer . . . 689
E.2. LoadState . . . 689
E.3. PersistenceProvider . . . 690
E.4. PersistenceProviderResolver . . . 693
E.5. PersistenceProviderResolverHolder . . . 694
E.6. PersistenceUnitInfo . . . 698
E.7. ProviderUtil . . . 703
Appendix F: Static Query API . . . 705
F.1. JakartaQuery . . . 705
F.2. NativeQuery . . . 707
F.3. QueryOptions . . . 709
F.4. StaticStatementReference . . . 711
F.5. StaticTypedQueryReference . . . 713
Appendix G: Result Set Mapping API . . . 717
G.1. ResultSetMapping . . . 717
G.2. MappingElement . . . 720
G.3. MemberMapping . . . 721
G.4. ColumnMapping . . . 721
G.5. FieldMapping . . . 722
G.6. EntityMapping . . . 723
G.7. EmbeddedMapping . . . 725
G.8. CompoundMapping . . . 727
G.9. ConstructorMapping . . . 727
G.10. TupleMapping . . . 729

```
Specification: Jakarta Persistence

Version: 4.0-M4

Status: DRAFT

Release: May 25, 2026
```

Copyright (c) 2019, 2024 Eclipse Foundation.

Eclipse Foundation Specification License - v1.1

By using and/or copying this document, or the Eclipse document from which this statement is linked or incorporated by reference, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [$date-of-document] Eclipse Foundation AISBL [url to this license] "

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

> "Copyright (c) [$date-of-document] Eclipse Foundation AISBL. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

> THIS DOCUMENT IS PROVIDED "AS IS," AND TO THE EXTENT PERMITTED BY APPLICABLE LAW THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION AISBL MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
>
> TO THE EXTENT PERMITTED BY APPLICABLE LAW THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION AISBL WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation AISBL may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Chapter 1. Introduction

This document is the specification of the Jakarta API for the management of persistence and object/relational mapping in the Jakarta EE and Java SE platforms. The technical objective of this work is to provide a standard object/relational mapping facility for the Java application developer using a Java domain model to manage data held in a relational database.

- Jakarta Persistence 4.0 is a major revision of the APIs, adding the `EntityAgent`, programmatic result set mappings, support for static queries, enhancements to entity graphs and `StoredProcedureQuery`, and new lifecycle events. Its feature set is targeted to support usage with Jakarta Data.
- The Jakarta Persistence 3.2 specification began the process of modernizing and updating the APIs for ease of use and greater type safety.
- The Jakarta Persistence 3.1 specification was the first release with new features and enhancements after the specification was moved to the Eclipse Foundation.
- The Jakarta Persistence 3.0 specification was the first release after moving the project to Eclipse Foundation. All APIs were moved from the package `javax.*` to the package `jakarta.*`. Every property name containing `javax` was renamed so that `javax` is replaced with `jakarta`.
- The Java Persistence 2.2 specification enhanced the API with support for repeating annotations; injection into attribute converters; support for mapping the `LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetTime`, and `OffsetDateTime` types from `java.time`; and methods to retrieve the results of `Query` and `TypedQuery` as streams.
- The Java Persistence 2.1 specification added support for schema generation, type conversion methods, use of entity graphs in queries and find operations, unsynchronized persistence contexts, stored procedure invocation, and injection into entity listener classes. It also included enhancements to the query language, the Criteria API, and to the mapping of native queries.

1.1. Authorship

The Jakarta Persistence Specification incorporates work done over two decades by the EJB 3.0 expert group, the Java Persistence 2.0, 2.1, and 2.2 expert groups, under the aegis of the Java Community Process, and by the Jakarta Persistence project at the Eclipse Foundation.

1.2. Document Conventions

Regular serif font is used for information that is prescriptive under this specification.

Admonitions are used for paragraphs that contain descriptive information clarifications, explanations, and recommendations.

`Monospaced font` is used for code examples and to specify the BNF of the Jakarta Persistence query language.

Java types belonging to the namespace `jakarta.persistence` are usually referred to by their unqualified names when there is no ambiguity.

This document defines the semantics of a set of Java language annotations. An XML descriptor (as specified in Chapter 12) may be used as an alternative to annotations or to augment or override annotations. The elements of this descriptor mirror the annotations and have identical semantics to the corresponding annotations. When semantic requirements are written in terms of annotations, it should be understood that the same semantics apply to the corresponding elements of the XML descriptor.

Chapter 2. Entities

An entity is a lightweight persistent domain object.[1] Entities support inheritance, polymorphic associations, and polymorphic queries.

The primary programming artifact is the *entity class*. An entity class may make use of auxiliary classes that serve as helper classes or that are used to represent the state of the entity.

This chapter describes requirements on entity classes and instances.

2.1. The Entity Class

The entity class must be annotated with the `Entity` annotation or declared as an `entity` in its XML descriptor.

- The entity class must be a top-level class or a static inner class. An enum, record, or interface may not be designated as an entity.
- The entity class must have a `public` or `protected` constructor with no parameters, which is called by the persistence provider runtime to instantiate the entity.[2] This constructor is usually declared in the Java source code, but it might be added using source or bytecode processing by the build process or runtime environment.[3] The entity class may have additional constructors for use by the application.
- The entity class must be non-`final`. Every method and persistent instance variable of the entity class must also be non-`final`.

An entity might be an abstract class, or it might be a concrete class. An entity may extend a non-entity class, or it may extend another entity class. A non-entity class may extend an entity class.

The persistent state of an entity is represented by instance variables, which may correspond to JavaBeans properties. An instance variable may be directly accessed only within the methods of the entity, by the entity instance itself. An instance variable of an entity must *not* be directly accessed by a client of the entity. The state of the entity is available to clients only through the methods of the entity—that is, via accessor (getter/setter) methods, or via other business methods.

2.2. Persistent Fields and Properties

The persistent state of an entity is accessed by the persistence provider runtime via either:

- *property access* using JavaBeans-style property accessors, or
- *field access*, that is, direct access to instance variables.

The instance variables of a class must have `private`, `protected`, or package visibility, independent of whether field access or property access is used. When property access is used, the property accessor methods must be `public` or `protected`.

The type of a persistent field or property of an entity class may be:

- any basic type listed below in Section 2.6, including any Java `enum` type,
- an entity type or a collection of some entity type, as specified in Section 2.11,
- an embeddable class, as defined in Section 2.7, or
- a collection of a basic type or embeddable type, as specified in Section 2.8.

The declared type of a persistent field or property must not involve a type variable.

Object/relational mapping metadata may be specified to customize the object/relational mapping and the loading and storing of the entity state and relationships, as specified in Chapter 11.

The placement of object/relational mapping annotations depends on whether property access or field access is used:

- When field access is used, mapping annotations must be placed on instance variables, and the persistence provider runtime accesses instance variables directly. Every non-`transient` instance variable not annotated with the `Transient` annotation is persistent.
- When property-based access is used, mapping annotations must be placed on getter methods[4], and the persistence provider runtime accesses persistent state via the property accessor methods. Every property not annotated with the `Transient` annotation is persistent.

Mapping annotations must not be applied to a field or property marked `static`, `transient`, or `@Transient`, such a field or property is not persistent.

Whether property access, field access, or a mix of the two options is used by the provider to access the persistent state of a given entity class or entity hierarchy is determined by the rules defined in Section 2.3.

> Terminology Note: The persistent fields and properties of an entity class are generically referred to in this document as *attributes* of the class.

Collection-valued persistent fields and properties must be defined in terms of one of the following collection-valued interfaces, regardless of whether the entity class otherwise adheres to the JavaBeans method conventions noted below, and of whether field or property access is used: `java.util.Collection`, `java.util.Set`, `java.util.List` [5], `java.util.Map`.

Use of the generic variants of these collection types is strongly encouraged, for example, `Set<Order>` is preferred to the raw type `Set`.

> Terminology Note: The terms *collection* and *collection-valued* are used in this specification to denote any of the above types, unless further qualified. In cases where a `java.util.Collection` type—or one of its subtypes—must be distinguished, the type is identified explicitly. The terms *map* and *map collection* are used to denote a collection of type `java.util.Map`.

A collection implementation type such as `HashSet` or `ArrayList` may be used by the application to initialize a collection-valued field or property before the entity is made persistent. Once the entity becomes managed (or detached), subsequent access to the collection must be through the interface type. The persistence provider runtime is permitted to replace, at any time, any reference held by an entity to a collection instantiated by the application with a different provider-specific object implementing the same interface type.

A null value assigned to a collection-valued persistent field or property is interpreted as an empty collection. The persistence provider runtime is permitted to replace a null value held by a collection-valued field or property of an entity with an empty collection.

2.2.1. Persistent Attribute Type

Every persistent field or property of an entity is either:

- an attribute of **basic type**, as defined in Section 2.6, or of a type converted to a basic type through use of a *converter*, as defined in Section 3.13,
- an **embedded attribute** whose type is a fine-grained embeddable class, as defined Section 2.7,
- an **element collection**, that is, a collection whose elements are instances of a basic type or embeddable class, as defined in Section 2.8,
- a **relationship** (or **association**) with another entity type, as defined in Section 2.11, which is further classified by its multiplicity: one-to-one, many-to-one, one-to-many, or many-to-many.

The enumeration `jakarta.persistence.metamodel.Attribute.PersistentAttributeType` defines a categorization of persistent entity attributes:

- `BASIC` for basic attributes,
- `EMBEDDED` for embedded attributes,
- `ELEMENT_COLLECTION` for element collections, and
- `MANY_TO_ONE`, `ONE_TO_ONE`, `ONE_TO_MANY`, and `MANY_TO_MANY` for associations of the indicated multiplicity.

Each persistent attribute of an entity belongs to exactly one of the listed types.

- An attribute annotated `@Basic` is a basic attribute.
- An attribute annotated `@Embedded` is an embedded attribute.
- An attribute annotated `@ElementCollection` is an element collection.
- An attribute annotated `@ManyToOne`, `@OneToOne`, `@OneToMany`, or `@ManyToMany` is an association of the indicated multiplicity.

If none of the above annotations occur on a persistent attribute, the defaulting rules specified below in Section 2.10 determine which category it belongs to.

An attribute of an entity must not have mapping annotations indicating conflicting persistent attribute types. For example, a field must not be annotated `@Basic @Embedded`, `@ManyToOne @ElementCollection`, or `@OneToOne @ManyToMany`. The persistence provider must detect such contradictory combinations of mapping annotations and report the error.[6]

2.2.2. Value Types and Reference Types

In the Java language, an instance of a non-primitive type has a well-defined object identity unique to the instance and is passed *by reference*, meaning that assignment of the instance to a field, local variable, or method parameter, followed by later access to the assigned value, preserves the identity of the instance.

In general, operations performed by a persistence provider are not required to preserve the object identity of a value stored in an attribute of an entity. The provider is permitted to modify the value held in the attribute to a new value with a distinct object identity. But such modifications never alter the *persistent* representation of the entity because the Java object identity of a value held in an attribute is never part of the persistent state of the entity.

In particular:

- A basic type, an embeddable class, or any collection type is considered a *value type*, and the provider is not required to respect the integrity of shared references to an instance of such a type. If multiple fields referenced the same instance of a value type when an entity was made persistent, those fields may hold references to distinct instances of the value type when the entity is reloaded from the database.
- An entity type is considered a *reference type*, and, instead of object identity, every managed instance of an entity type has a well-defined *persistent identity*. The persistence provider must respect its persistent identity by:
  - preserving the persistent identity of an instance when it is made persistent and then later reloaded from the database, and
  - ensuring that, within a single persistence context, at most one object represents the state of each distinct persistent identity, thus preserving the integrity of shared references to entities belonging to the given persistence context.

  This requirement notwithstanding, the container is permitted to introduce an indirection or *proxy* between the object which represents the state of an entity and attributes which hold references to the entity. Thus, object identity might still not be respected in a strict sense, not even for a reference type.

2.2.3. Property Access

When property access is used, persistent properties of the entity class must follow the method signature conventions for JavaBeans read/write properties, as defined by the JavaBeans `Introspector` class. For every persistent property `property` of type `T` of the entity, there must be a getter method, `getProperty`, and setter method `setProperty`. For boolean properties, `isProperty` may be used as an alternative name for the getter method.[7]

For single-valued persistent properties, these method signatures are:

```
T getProperty()

void setProperty(T t)
```

For collection-valued persistent properties, the type `T` in the method signatures above must be one of the collection interface types listed above in Section 2.2.

In addition to returning and setting the persistent state of the entity instance, a property accessor method may contain additional logic, for example, logic to perform validation. The persistence provider runtime triggers execution of this logic when property-based access is used.

Therefore, caution should be exercised in adding business logic to accessor methods when property access is used. The order in which the persistence provider runtime calls these methods when loading or storing persistent state is not defined. Logic contained in such methods should therefore not rely on any specific invocation order.

If property access is used and lazy fetching is specified, portable applications should not directly access the entity state underlying the property methods of managed instances until after it has been fetched by the persistence provider.[8]

If a persistence context is joined to a transaction, runtime exceptions thrown by property accessor methods cause the current transaction to be marked for rollback; any exception thrown by such methods when called by the persistence runtime to load or store persistent state causes the persistence runtime to mark the current transaction for rollback and to throw a `PersistenceException` wrapping the application exception.

An entity subclass may override a property accessor method inherited from a superclass. However, portable applications must not override the object/relational mapping metadata applied to the persistent fields and properties of entity superclasses.

For example:

```
@Entity
public class Customer implements Serializable {
    private Long id;
    private String name;
    private Address address;
    private Collection<Order> orders = new HashSet();
    private Set<PhoneNumber> phones = new HashSet();

    // No-arg constructor
    public Customer() {}

    @Id // property access is used
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @OneToMany
    public Collection<Order> getOrders() {
        return orders;
    }

    public void setOrders(Collection<Order> orders) {
        this.orders = orders;
    }

    @ManyToMany
    public Set<PhoneNumber> getPhones() {
        return phones;
    }

    public void setPhones(Set<PhoneNumber> phones) {
        this.phones = phones;
    }

    // Business method to add a phone number to the customer
    public void addPhone(PhoneNumber phone) {
        this.getPhones().add(phone);

        // Update the phone entity instance to refer to this customer
        phone.addCustomer(this);
    }
}
```

2.3. Access Type

An *access type* determines how the persistence provider runtime reads and writes the persistent state of an entity from and to an instance of the entity class, as specified above in Section 2.2. `AccessType` enumerates the two possibilities:

```
public enum AccessType {
    FIELD,
    PROPERTY
}
```

The access type for a persistent attribute depends on the placement of object/relational mapping annotations in the entity class, and may be explicitly overridden via use of the `Access` annotation defined in Section 11.2.1.

2.3.1. Default Access Type

By default, a single access type (`FIELD` or `PROPERTY`) is inferred for an entity hierarchy. The *default access type* of an entity hierarchy is determined by the placement of mapping annotations on the attributes of the entity classes and mapped superclasses of the entity hierarchy which do not explicitly specify an access type.

- If mapping annotations are placed on instance variables, `FIELD` access is inferred.
- If mapping annotations are placed on getter methods, `PROPERTY` access is inferred.

An access type may be explicitly specified by means of the `Access` annotation[9], as described below in Section 2.3.2.

Every class in an entity hierarchy whose access type is defaulted in this way must be consistent in its placement of mapping annotations on either fields or properties, such that a single, consistent default access type applies within the hierarchy. Any embeddable class used by an entity within the hierarchy has the same access type as the default access type of the hierarchy unless the `Access` annotation is specified, as defined below.

It is an error if a default access type cannot be determined and an access type is not explicitly specified by a class-level `Access` annotation or the XML descriptor. The behavior of applications which mix the placement of mapping annotations on fields and properties within an entity hierarchy without explicitly specifying the class-level `Access` annotation is undefined.[10]

2.3.2. Explicit Access Type

The access type of an individual entity class, mapped superclass, or embeddable class may be specified for that class, independent of the default for the entity hierarchy to which it belongs, by annotating the class with the `Access` annotation.

- When `Access(FIELD)` is applied to an entity class, mapped superclass, or embeddable class, mapping annotations may be placed on the instance variables of that class, and the persistence provider runtime accesses persistent state via direct access to the instance variables declared by the class. Every non-`transient` instance variable not annotated with the `Transient` annotation is persistent.
- When `Access(PROPERTY)` is applied to an entity class, mapped superclass, or embeddable class, mapping annotations may be placed on the properties of that class, and the persistence provider runtime accesses persistent state via the properties declared by that class. Every property not annotated with the `Transient` annotation is persistent.

The explicit access type may be overridden at the attribute level. That is, a class which explicitly specifies an access type using the `Access` annotation may also have fields or properties annotated `Access`, and so the class may have a mix of access types.

- When `Access(FIELD)` is specified at the class level, an individual attribute within the class may be selectively designated for property access by annotating a property getter `Access(PROPERTY)`. Mapping annotations for this attribute must be placed on the getter. If a mapping annotation is placed on a property getter which is not annotated `Access(PROPERTY)`, the behavior is undefined.
- When `Access(PROPERTY)` is specified at the class level, an individual attribute within the class may be selectively designated for field access by annotating an instance variable `Access(FIELD)`. Mapping annotations for this attribute must be placed on the field. If a mapping annotation is placed on a field which is not annotated `Access(FIELD)`, the behavior is undefined.

It is permitted (but redundant) to place `Access(FIELD)` on a field whose class has field access or `Access(PROPERTY)` on a property whose class has property access. On the other hand, the behavior is undefined if:

- `Access(PROPERTY)` annotates a field,
- `Access(FIELD)` annotates a property getter, or
- the `Access` annotation occurs on a property setter.

Portable application should avoid such misplaced `@Access` annotations.

When access types are combined within a class, the `Transient` annotation should be used to avoid duplicate persistent mappings. For example:

```
@Entity @Access(PROPERTY)
public class Customer {
    private Long id;

    @Access(FIELD) // use field access for name
    private String name;

    @Id
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Transient // suppress duplicated name attribute
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    ...
}
```

The `Access` annotation does not affect the access type of other entity classes or mapped superclasses in the entity hierarchy. In particular, persistent state inherited from a superclass is always accessed according to the access type of that superclass.

2.3.3. Access Type of an Embeddable Class

The access type of an embeddable class is determined by the access type of the entity class, mapped superclass, or embeddable class in which it is embedded (including as a member of an element collection) independent of whether the access type of the containing class is explicitly specified or defaulted. A different access type for an embeddable class can be specified for that embeddable class by means of the `Access` annotation as described above in Section 2.3.2.

2.3.4. Defaulted Access Types of Embeddable Classes and Mapped Superclasses

Care must be taken when implementing an embeddable class or mapped superclass which is used both in a context of field access and in a context of property access, and whose access type is not explicitly specified by means of the `Access` annotation or XML mapping file.

Such a class should be implemented so that the number, names, and types of its persistent attributes are independent of the access type in use. If an embeddable class violates this principle, and if it is used in contexts of non-uniform access type, the representation of the embeddable class in the metamodel API defined in Chapter 5 is undefined and unportable.

2.4. Primary Keys and Entity Identity

Every entity must have a primary key. The value of its primary key uniquely identifies an entity instance within a persistence context and to operations of the `EntityManager` or `EntityAgent`, as described in Chapter 3, and represents the *persistent identity* of the instance, a concept which is defined more carefully later in Section 3.6.10.

The primary key must be declared by:

- the entity class that is the root of the entity hierarchy, or
- a mapped superclass that is a (direct or indirect) superclass of all entity classes in the entity hierarchy.

A primary key must be defined exactly once in each entity hierarchy.

> Terminology Note: In the community, the primary key of an entity is usually called its *identifier*, but for historical reasons that term is not used in this specification.

A primary key comprises one or more fields or properties (attributes) of the entity class.

- A *simple primary key* is a single persistent field or property of the entity class whose type is one of the legal simple primary key types listed below. The `Id` annotation defined in Section 11.2.24 or `id` XML element must be used to identify the simple primary key.
- A *composite primary key* may be declared as either a single persistent field or property, or as a set of fields or properties, as described below.[11] A *primary key class* must be defined to represent the composite primary key.
  - If the composite primary key is represented by a single field or property of the entity, the `EmbeddedId` annotation defined by Section 11.2.18 identifies the primary key, and the type of the annotated field or property is the primary key class.
  - Otherwise, when the composite primary key encompasses multiple fields or properties, the `Id` annotation defined by Section 11.2.24 identifies the fields and properties which belong to the composite key, and the `IdClass` annotation defined by Section 11.2.25 must specify the primary key class.

A simple primary key or a field or property belonging to a composite primary key must have one of the following basic types:

- any Java primitive type, or `java.lang` wrapper for a primitive type, [12]
- `java.lang.String`,
- `java.util.UUID`,
- `java.time.LocalDate`, `java.util.Date`, or `java.sql.Date`,
- `BigDecimal` or `BigInteger` from `java.math`.

If a primary key field or property has type `java.util.Date`, the temporal type must be explicitly specified as `DATE` using the `Temporal` annotation defined by Section 11.2.56, or by equivalent XML.

If the primary key is a composite primary key derived from the primary key of another entity, the primary key may contain an attribute whose type is that of the primary key of the referenced entity, as specified below in Section 2.4.2.

An entity with a primary key involving any type other than the types listed above is not portable. If the primary key is generated by the persistence provider, as defined by Section 11.2.23, and its type is not `long`, `int`, `java.util.UUID`, `java.lang.String`, `java.lang.Long`, or `java.lang.Integer`, the entity is not portable.

The application must not change the value of the primary key of an entity instance after the instance is made persistent[13]. If the application does change the value of a primary key of an entity instance after the entity instance is made persistent, the behavior is undefined.[14]

2.4.1. Composite Primary Keys

The following rules apply to composite primary keys:

- The primary key class may be a non-abstract regular Java class with a `public` or `protected` constructor with no parameters. Alternatively, the primary key class may be any Java record type, in which case it is not required to have a constructor with no parameters.

- The access type (`FIELD` or `PROPERTY`) of a primary key class is determined by the access type of the entity for which it is the primary key, unless the primary key is an embedded id and an explicit access type is specified using the `Access` annotation, as defined in Section 2.3.2.
- If property-based access is used, the property accessors of the primary key class must be `public` or `protected`.
- The primary key class must define `equals` and `hashCode` methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.
- A composite primary key must either be represented and mapped as an embeddable class (see Section 11.2.18) or it must be represented as an id class and mapped to multiple fields or properties of the entity class (see Section 11.2.25).
- If the composite primary key is represented as an id class, the names of the primary key fields or properties of the primary key class and those of the entity class to which the id class is mapped must match and their types must be the same, unless the primary key involves a derived identity.
- A primary key which involves a derived identity must conform to the rules specified below in Section 2.4.2.

2.4.2. Primary Keys Involving Derived Identities

The identity of an entity is said to be partially *derived* from the identity of a second entity when the *child* or *dependent* first entity is the owner of a many-to-one or one-to-one relationship which targets the *parent* second entity and the foreign key referencing the parent entity forms part of the primary key of the dependent entity.

A derived identity might be represented as a simple primary key or as a composite primary key, as described in Section 2.4.2.2 below. The dependent entity class has a composite primary key if

- it declares one or more primary key attributes in addition to those corresponding to the primary key of the parent, or
- the parent itself has a composite primary key

and then an embedded id or id class must be used to represent the primary key of the dependent entity. In the case that the parent has a composite key, it is *not* required that parent entity and dependent entity both use embedded ids, nor that both use id classes.

A `ManyToOne` or `OneToOne` relationship which maps a primary key column or columns may be declared using either:

- the `Id` annotation, when no other `Id` or `EmbeddedId` attribute maps the same primary key column or columns, or
- the `MapsId` annotation, as defined below in Section 2.4.2.1, if some other attribute or attributes annotated `Id` or `EmbeddedId` also map the primary key column or columns.

If a `ManyToOne` or `OneToOne` relationship declared by a dependent entity is annotated `Id` or `MapsId`, an instance of the entity cannot be made persistent until the relationship has been assigned a reference to an instance of the parent entity, since the identity of the dependent entity declaring the relationship is derived from the referenced parent entity. [15]

A dependent entity may have more than one parent entity.

2.4.2.1. Use of `MapsId` to Represent Derived Identity

The `MapsId` annotation defined in Section 11.2.42 allows a relationship to the parent entity of an entity with a derived identity to be redundantly represented by:

1. a relationship attribute annotated `ManyToOne` or `OneToOne`, and
2. a simple or composite primary key field or property, as defined above in Section 2.4.

As specified in Section 11.2.18, an embedded id class cannot declare a field or property representing a relationship. Therefore, a `ManyToOne` or `OneToOne` relationship mapping the primary key column or columns of an entity with an

embedded id must be declared using `MapsId`. On the other hand, a `ManyToOne` or `OneToOne` relationship mapping the primary key column or columns of an entity with an id class may be declared with either `Id` or `MapsId`, as specified below in Section 2.4.2.2.

When `MapsId` is used to map a `ManyToOne` or `OneToOne` relationship, the entity class, id class, or embedded id class must have a corresponding field or property with the same type as the identifier of the associated entity. The `value` member of the `MapsId` annotation identifies this field or property, which is said to be *mapped by* the relationship attribute annotated `MapsId`.

- If the dependent entity has the same primary key type as the parent entity, or if the dependent entity represents its composite primary key as an id class, the field or property mapped by the relationship attribute must be declared using the `Id` or `EmbeddedId` annotation in the entity class. If the associated entity has a simple primary key, it is declared using the `Id` annotation; otherwise, if the associated entity has a composite primary key, it must be declared using the `EmbeddedId` annotation.
- If the dependent entity represents its composite primary key as an embedded id, no `Id` or `EmbeddedId` annotation is required on the field or property mapped by the relationship attribute.

2.4.2.2. Specification of Derived Identities

A dependent entity may have a composite primary key represented as an id class or embedded id, or it may have a simple primary key.

Derived Identity Represented as Id Class

If a dependent entity uses an id class to represent its primary key, one of the two following rules must be observed:

- The names and types of the attributes of the id class and the `Id` and `EmbeddedId` attributes of the dependent entity class must correspond as follows:
  - The `Id` or and `EmbeddedId` attribute of the dependent entity class and the corresponding attribute in the id class must have the same name.
  - If an `Id` attribute of the dependent entity class is of basic type, the corresponding attribute in the id class must have the same type.
  - If an `Id` attribute of the entity is a `ManyToOne` or `OneToOne` relationship to the parent entity, the corresponding attribute in the id class must be of the same Java type as the id class or embedded id of the parent entity (if the parent entity has a composite primary key) or the type of the `Id` attribute of the parent entity (if the parent entity has a simple primary key).
  - An `Id` attribute of the dependent entity class which is mapped by a relationship to the parent entity annotated `MapsId` must be of the same Java type as the `Id` attribute of the parent entity. The `value` element of the `MapsId` annotation of the relationship attribute must specify the name of the `Id` attribute.
  - An `EmbeddedId` attribute of the dependent entity class which is mapped by a relationship to the parent entity annotated `MapsId` must be of the same Java type as the id class or embedded id of the parent entity. The `value` element of the `MapsId` annotation of the relationship attribute must specify the name of the `EmbeddedId` attribute.
- Alternatively, if the dependent entity declares a single primary key attribute, that is, a `OneToOne` relationship attribute annotated `Id`, then the id class specified by the dependent entity must be the same as the primary key class of the parent entity.

Derived Identity Represented as Embedded Id

If a dependent entity uses an embedded id to represent its primary key, the relationship attribute which targets the parent entity must be annotated `MapsId`.

- If the embedded id of the dependent entity is of the same Java type as the primary key of the parent entity, then the relationship attribute maps both the relationship to the parent and the primary key of the dependent entity, the relationship attribute must be a `OneToOne` association, and the `MapsId` annotation must leave the `value` element unspecified. [16]
- Otherwise, the `value` element of the `MapsId` annotation must specify the name of the attribute within the embedded id which is mapped by the annotated relationship attribute and this attribute of the embedded id must be of the same type as the primary key of the parent entity.

An attribute of an embedded id which corresponds to a relationship targeting a parent entity is treated by the provider as "read-only"—that is, any direct mutation of the attribute is not propagated to the database.

Derived Identity Represented as Simple Primary Key

If a dependent entity has a single primary key attribute annotated `Id`, and the primary key of the parent entity is a simple primary key, then the primary key of the dependent entity is a simple primary key of the same Java type as that of the parent entity, the relationship attribute must be a `OneToOne` association targeting the parent entity, and either:

1. the primary key attribute annotated `Id` is the relationship attribute itself, or
2. the primary key attribute annotated `Id` has the same type as the simple primary key of the parent entity, the relationship attribute is annotated `MapsId`, and the `value` element of the `MapsId` annotation is left unspecified.

Neither `EmbeddedId` nor `IdClass` is specified for the dependent entity.

2.4.2.3. Mapping of Derived Identities

A dependent entity has derived primary key attributes, and might also have additional primary key attributes which are not derived from any parent entity.

- Any primary key attribute of a dependent entity which is derived from the identity of a parent entity is mapped by annotations of the corresponding `ManyToOne` or `OneToOne` relationship attribute. The default mapping for this relationship is specified in Section 2.12. The default mapping may be overridden by annotating the relationship attribute with the `JoinColumn` or `JoinColumns` annotation.
- If the dependent entity uses an id class, the `Column` annotation may be used to override the default mapping of `Id` attributes which are *not* derived from any parent entity.
- If the dependent entity uses an embedded id to represent its primary key, the `AttributeOverride` annotation applied to the `EmbeddedId` attribute may be used to override the default mapping of embedded id attributes which are *not* derived from any parent entity.

2.4.2.4. Examples of Derived Identities

The following examples illustrate the rules specified above.

Example 1:

The parent entity has a simple primary key:

```
@Entity
public class Employee {
    @Id long empId;
    String empName;

    // ...
}
```

Case (a): The dependent entity uses `IdClass` to represent a composite key:

```
public class PaycheckId {
    int period; // matches name of @Id attribute
    long emp; // matches name of @Id attribute and type of Employee PK
}

@Entity
@IdClass(PaycheckId.class)
public class Paycheck {
    @Id int period;

    // id attribute mapped by join column default
    @Id @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```
SELECT d
FROM Paycheck d
WHERE d.period = 1 AND d.emp.empName = 'Sam'
```

Case (b): The dependent entity uses `IdClass` and `MapsId` to represent a composite key:

```
public class PaycheckId {
    int period; // matches name of @Id attribute
    long empId; // matches name of @Id attribute and type of Employee PK
}

@Entity
@IdClass(PaycheckId.class)
public class Paycheck {
    @Id int period;
    @Id long empId;

    // id attribute mapped by join column default
    @MapsId("empId")
    @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```
SELECT d
FROM Paycheck d
WHERE d.period = 1 AND d.emp.empName = 'Sam'
```

Case(c): The dependent entity uses `EmbeddedId` to represent a composite key:

```
@Embeddable
public class PaycheckId {
    int period;
    long empPK; // corresponds to PK type of Employee
}

@Entity
public class Paycheck {
    @EmbeddedId PaycheckId id;
```

```
    // id attribute mapped by join column default
    @MapsId("empPK") // maps empPK attribute of embedded id
    @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```
SELECT d
FROM Paycheck d
WHERE d.id.period = 1 AND d.emp.empName = 'Sam'
```

Example 2:

The parent entity uses `IdClass`:

```
public class EmployeeId {
    String firstName;
    String lastName;

    // ...
}

@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id String firstName;
    @Id String lastName;

   // ...
}
```

Case (a): The dependent entity uses `IdClass`:

```
public class PaycheckId {
    int period; // matches name of attribute
    EmployeeId emp; //matches name of attribute and type of Employee PK
}

@Entity
@IdClass(PaycheckId.class)
public class Paycheck {
    @Id
    int period;

    @Id
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @ManyToOne
    Employee emp;
}
```

Sample query:

```
SELECT d
FROM Paycheck d
WHERE d.period = 1 AND d.emp.firstName = 'Sam'
```

Case (b): The dependent entity uses `IdClass` and `MapsId`:

```
public class PaycheckId {
    int period; // matches name of attribute
    EmployeeId empId; //matches name of attribute and type of Employee PK
}

@Entity
@IdClass(PaycheckId.class)
public class Paycheck {
    @Id
    int period;

    @EmbeddedId
    EmployeeId empId;

    @MapsId("empId")
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @ManyToOne
    Employee emp;
}
```

Sample query:

```
SELECT d
FROM Paycheck d
WHERE d.period = 1 AND d.empId.firstName = 'Sam'
```

Case (c): The dependent entity uses `EmbeddedId`. The type of the `empPK` attribute is the same as that of the primary key of `Employee`. The `EmployeeId` class needs to be annotated `Embeddable` or denoted as an embeddable class in the XML descriptor.

```
@Embeddable
public class PaycheckId {
    int period;
    EmployeeId empPK;
}

@Entity
public class Paycheck {
    @EmbeddedId
    PaycheckId id;

    @MapsId("empPK")
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```
SELECT d
FROM Paycheck d
WHERE d.id.period = 1 AND d.emp.firstName = 'Sam'
```

Note that the following alternative query will yield the same result:

```
SELECT d
FROM Paycheck d
WHERE d.id.period = 1 AND d.id.empPK.firstName = 'Sam'
```

Example 3:

The parent entity uses `EmbeddedId`:

```
@Embeddable
public class EmployeeId {
    String firstName;
    String lastName;

    // ...
}

@Entity
public class Employee {
    @EmbeddedId
    EmployeeId empId;

    // ...
}
```

Case (a): The dependent entity uses `IdClass`:

```
public class PaycheckId {
    int period; // matches name of @Id attribute
    EmployeeId emp; // matches name of @Id attribute and type of embedded id of Employee
}

@Entity
@IdClass(PaycheckId.class)
public class Paycheck {
    @Id
    @Column(name="pay_period") // default column name is overridden
    int period;

    @Id
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @ManyToOne Employee
    emp;
}
```

Sample query:

```
SELECT d
FROM Paycheck d
WHERE d.period = 1 and d.emp.empId.firstName = 'Sam'
```

Case (b): The dependent entity uses `IdClass` and `MapsId`:

```
public class PaycheckId {
    int period; // matches name of @Id attribute
    EmployeeId empId; // matches name of @Id attribute and type of embedded id of Employee
}
```

```java
@Entity
@IdClass(PaycheckId.class)
public class Paycheck {
    @Id
    @Column(name="pay_period") // default column name is overridden
    int period;

    @EmbeddedId
    EmployeeId empId;

    @MapsId("empId")
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @ManyToOne Employee
    emp;
}
```

Sample query:

```select
SELECT d
FROM Paycheck d
WHERE d.period = 1 and d.empId.firstName = 'Sam'
```

Case (c): The dependent entity uses `EmbeddedId`:

```java
@Embeddable
public class PaycheckId {
    int period;
    EmployeeId empPK; // corresponds to PK type of Employee
}

@Entity
public class Paycheck {
    // default column name for 'period' attribute is overridden
    @AttributeOverride(name="period", column=@Column(name="pay_period"))
    @EmbeddedId PaycheckId id;

    @MapsId("empPK")
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @ManyToOne
    Employee emp;

    // ...
}
```

Sample query:

```select
SELECT d
FROM Paycheck d
WHERE d.id.period = 1 and d.emp.empId.firstName = 'Sam'
```

Note that the following alternative query will yield the same result:

```select
SELECT d
FROM Paycheck d
WHERE d.id.period = 1 AND d.id.empPK.firstName = 'Sam'
```

Example 4:

The parent entity has a simple primary key:

```
@Entity
public class Person {
    @Id
    String ssn;

    // ...
}
```

Case (a): The dependent entity has a single primary key attribute which is mapped by the relationship attribute. The primary key of `MedicalHistory` is of type `String`.

```
@Entity
public class MedicalHistory {
    // default join column name is overridden
    @Id
    @OneToOne
    @JoinColumn(name="FK")
    Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m
WHERE m.patient.ssn = '123-45-6789'
```

Case (b): The dependent entity has a single primary key attribute corresponding to the relationship attribute. The primary key attribute is of the same basic type as the primary key of the parent entity. The `MapsId` annotation applied to the relationship attribute indicates that the primary key is mapped by the relationship attribute.[17]

```
@Entity
public class MedicalHistory {
    @Id
    String id; // overriding not allowed

    // ...

    // default join column name is overridden
    @MapsId
    @JoinColumn(name="FK")
    @OneToOne
    Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m WHERE m.patient.ssn = '123-45-6789'
```

Example 5:

The parent entity uses `IdClass`. The dependent's primary key class is of same type as that of the parent entity.

```
public class PersonId {
    String firstName;
    String lastName;
}

@Entity
@IdClass(PersonId.class)
public class Person {
    @Id
    String firstName;

    @Id
    String lastName;

    // ...
}
```

Case (a): The dependent entity uses `IdClass`:

```
@Entity
@IdClass(PersonId.class)
public class MedicalHistory {
    @Id
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @OneToOne
    Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m
WHERE m.patient.firstName = 'Charles'
```

Case (b): The dependent entity uses `IdClass` and `MapsId:

```
@Entity
@IdClass(PersonId.class)
public class MedicalHistory {

    @EmbeddedId
    PersonId personId;

    @MapsId("personId")
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @OneToOne
    Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m
```

```
WHERE m.personId.firstName = 'Charles'
```

Case (c): The dependent entity uses the `EmbeddedId` and `MapsId` annotations. The `PersonId` class needs to be annotated `Embeddable` or denoted as an embeddable class in the XML descriptor.

```
@Entity
public class MedicalHistory {
    // all attributes map to relationship:
    // AttributeOverride not allowed

    @EmbeddedId
    PersonId id;

    // ...

    @MapsId
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @OneToOne Person patient;

    // ...
}
```

Sample query:

```
SELECT m
FROM MedicalHistory m
WHERE m.patient.firstName = 'Charles'
```

Note that the following alternative query will yield the same result:

```
SELECT m
FROM MedicalHistory m
WHERE m.id.firstName = 'Charles'
```

Example 6:

The parent entity uses `EmbeddedId`. The dependent's primary key is of the same type as that of the parent.

```
@Embeddable
public class PersonId {
    String firstName;
    String lastName;
}

@Entity
public class Person {
    @EmbeddedId PersonId id;

    // ...
}
```

Case (a): The dependent class uses `IdClass`:

```
@Entity
@IdClass(PersonId.class)
public class MedicalHistory {
    @Id
    @OneToOne
    @JoinColumns({
```

```
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    Person patient;

    // ...
}
```

Case (b): The dependent class uses `EmbeddedId`:

```
@Entity
public class MedicalHistory {
    // All attributes are mapped by the relationship
    // AttributeOverride is not allowed
    @EmbeddedId PersonId id;

    // ...

    @MapsId
    @JoinColumns({
        @JoinColumn(name="FK1", referencedColumnName="firstName"),
        @JoinColumn(name="FK2", referencedColumnName="lastName")
    })
    @OneToOne
    Person patient;

    // ...
}
```

2.5. Entity Versions

An entity might have a *version*, a persistent field or property used by the persistence provider to perform *optimistic lock verification*, as specified in Section 3.9.2. The version field or property holds a version number or timestamp identifying the revision of the entity data held by an entity class instance. In the course of performing lifecycle operations involving the entity instance, the persistence provider gets and sets the version field or property of the entity instance to determine or modify its version number or timestamp. The `Version` annotation defined in Section 11.2.59 or `version` XML element must be used to explicitly identify the version field or property of an entity.

An entity class may access the state of its version field or property or export a method which allows other user-written code to access the version, but user-written code must not directly modify the value of the version field or property of an entity instance after the entity is made persistent. [18] With the exception noted in Section 4.11, only the persistence provider is permitted to set or update the entity version. If the application does directly modify the value of the version field or property of an entity instance after it is made persistent, the behavior is undefined.

The version must be of one of the following basic types:

- `int`, `Integer`, `short`, `Short`, `long`, `Long`, or
- `java.time.LocalDateTime`, `java.time.Instant`, or `java.sql.Timestamp`.

A portable application must not declare a version field or property with any other type.

An entity class should have at most one version. A portable application must not define an entity class having more than one version field or property.

The version should be declared by the root entity class in an entity class hierarchy, or by one of its mapped superclasses. A portable application must not declare a version field or property in a subclass of the root class of an entity class hierarchy.

2.6. Basic Types

The following Java types are considered *basic types*:

- any Java primitive type, or `java.lang` wrapper class for a primitive type,
- `java.lang.String`,
- `java.util.UUID`,
- `BigInteger` or `BigDecimal` from `java.math`,
- `LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetTime`, `OffsetDateTime`, `Instant`, or `Year` from `java.time`,
- `Date` or `Calendar` [19] from `java.util` [20],
- `Date`, `Time`, or `Timestamp` from `java.sql` [21],
- `byte[]` or `char[]`.
- any Java `enum` type,
- any other type which implements `java.io.Serializable`.

Persistence for basic types is defined in Section 11.2.6 and Section 11.2.19.

2.7. Embeddable Classes

An *embeddable class* is a fine-grained class that represents part of the state of an entity. An instance of an embeddable class, unlike an entity instance, does not have a persistent identity of its own. Instead, it exists only to represent part of the state of the entity instance to which it belongs.

An entity may have:

- an *embedded attribute*, that is, single-valued field or property whose type is an embeddable class, or even
- a collection-valued attribute whose element type is an embeddable class.

An embeddable type may even be used to represent the key or value of a map.

An embedded object belongs strictly to its owning entity instance and is not shareable across persistent entity instances; sharing of an embedded object across multiple managed entity instances has undefined semantics. As described above in Section 2.2.2, the persistence provider is not required to respect the object identity of a value type, and this identity is typically not made persistent in the database.

An embeddable class must be annotated as `Embeddable` or designated as `embeddable` in its XML descriptor. The access type for an embedded object is determined as described in Section 2.3.

An embeddable class may be a regular Java class that adheres to the requirements specified in Section 2.1 for entities, with the exception that an embeddable class is not annotated as `Entity`, and an embeddable class must not be `abstract`.

Alternatively, an embeddable class may be a Java record type.

An embeddable class may have attributes of basic type, embedded attributes, collections, and associations:

- An embeddable class may have an embedded field or property. However, an embeddable class may not occur, directly or indirectly, as the type of one of its own embedded attributes. Circular containment dependencies among embeddable classes are not permitted.
- An embeddable class, including an embeddable class embedded within another embeddable class, may have a collection-valued field or property with elements of basic type or of another embeddable class type.
- An embeddable class may have a relationship to an entity type, even a collection-valued relationship to an entity type. Since an instance of an embeddable class has no persistent identity, any inverse relationship *from* such a referenced entity must be to the *entity* that contains the embeddable instance and not to the embeddable object

itself.[22] An embeddable class that is used as an embedded id or as a map key must not declare such a relationship.

Additional requirements and restrictions on embeddable classes are described in Section 2.8.

2.8. Collections of Embeddable Classes and Basic Types

An *element collection* is a persistent field or property of an entity or embeddable class that is collection-valued, and whose elements are of a basic type, as specified above in Section 2.6, or of an embeddable class, as specified immediately above in Section 2.7.

An element collection must be explicitly declared as such by the `ElementCollection` annotation or `element-collection` XML element. If the `ElementCollection` annotation (or XML equivalent) is not specified for a collection-valued field or property, the rules of Section 2.10 apply.

An element collection must have declared type `java.util.Collection`, `java.util.Set`, `java.util.List`, or `java.util.Map`. Collections of type `Map` are special, and are described below in Section 2.9.

When an element collection has declared type `Collection<E>`, `Set<E>`, or `List<E>`, the type argument `E` is taken, by default, to be its element type. The element type may be explicitly specified using the `targetClass` member of the `ElementCollection` annotation. If an element collection is declared with a raw type, the `targetClass` member must be explicitly specified. Otherwise, if the element collection is not declared with a raw type, the `targetClass` is optional, and it must specify an entity type which is assignable to the type argument `E`.

An element collection maps to a *collection table*, as defined in Section 11.2.9.

If an embeddable class, including an embeddable class embedded within another embeddable class, is contained, directly or indirectly, within an element collection:

- it must not have its own element collection;
- it must not have a one-to-many or many-to-many relationship to an entity;
- but it may have a many-to-one or one-to-one association to an entity if the relationship is represented by a foreign key mapping, as defined in Section 2.11.

2.9. Map Collections

An element collection may be of type `java.util.Map`. Such a map collection has two different element types:

- a *key type* and
- a *value type*.

A key or value type may be a basic type, an embeddable class, or an entity type. The key type and value type are completely independent of one another.

A map collection must be declared using an `ElementCollection`, `OneToMany`, or `ManyToMany` annotation:

- when the map value type is a basic type or embeddable class, the `ElementCollection` annotation is used;
- but when the map value type is an entity, a `OneToMany` or `ManyToMany` annotation is used.

If a relationship is bidirectional relationship, only one side of the relationship may be of type `java.util.Map`.

2.9.1. Map Values

When an element collection has declared type `Map<K,V>`, the type argument `V` is taken, by default, to be its value type. The value type may be explicitly specified using the `targetClass` member of the `ElementCollection` annotation. If an

element collection is declared with a raw type, the `targetClass` member must be explicitly specified.

When the value type of the map is a basic type or an embeddable class, the map maps to a collection table. The default column mappings for the map value are derived according to the default mapping rules for the `CollectionTable` annotation, as defined in Section 11.2.9.

- If the map value type is a basic type, the `Column` annotation is used to override the default column mappings.
- If the map value type is an embeddable class, the `AttributeOverride` and `AssociationOverride` annotations are used to override the default column mappings, as described in Section 11.2.4 and Section 11.2.2.

When the value type of a map is an entity type:

- the map maps to a join table if the map represents a many-to-many relationship or, by default, if it represents a unidirectional one-to-many relationship, but
- if the map represents the collection-valued side of a bidirectional one-to-many/many-to-one relationship, then, by default, the map maps to the primary table of the value entity type.

In this case, default mappings are described in Section 2.12.

2.9.2. Map Keys

When an element collection has declared type `Map<K,V>`, the type argument `K` is taken, by default, to be its key type. The key type may be explicitly specified using the `MapKeyClass` annotation. If an element collection is declared with a raw type, the `MapKeyClass` annotation is required.

When the map value type is an entity type, the `MapKey` annotation specifies that the map key is itself the primary key or a persistent field or property of the map value entity.

The `MapKey` and `MapKeyClass` annotations are mutually exclusive; an element collection may have either a `@MapKey` or `@MapKeyClass` annotation, or neither, but not both.

Dedicated annotations are provided for specifying column mappings for map keys:

- If the map key type is a basic type, the `MapKeyColumn` annotation is used to specify the column mapping for the map key. If the `MapKeyColumn` annotation is not specified, the default values of the `MapKeyColumn` annotation apply as described in Section 11.2.36.
- If the map key type is an embeddable class, the mappings for the map key columns are defaulted according to the default column mappings for the embeddable class. (See Section 11.2.10). The `AttributeOverride` annotation can be used to override these default mappings, as described in Section 11.2.4.
- If the map key type is an entity type, the `MapKeyJoinColumn` and `MapKeyJoinColumns` annotations are used to specify its column mappings. If the primary key of the referenced entity is a simple primary key and the `MapKeyJoinColumn` annotation is not specified, the default values of the `MapKeyJoinColumn` annotation apply as specified in Section 11.2.38.

When the map key type is an embeddable class, the embeddable class must implement the `hashCode` and `equals` methods consistently with the semantics of equality for the database columns to which the embeddable is mapped[23].

2.10. Mapping Defaults for Non-Relationship Fields or Properties

If a persistent field or property other than a relationship property is *not* annotated with one of the mapping annotations listed above in Section 2.2.1, and if no equivalent mapping information is supplied in any XML descriptor, nor via any provider-specific mechanism, the following defaulting rules are applied in order:

1. If the type of the field or property is a class annotated with the `Embeddable` annotation, the field or property is treated as if it were annotated with the `Embedded` annotation, as defined in Section 11.2.16 and Section 11.2.17.

2. Otherwise, if the type of the field or property is one of the basic types listed in Section 2.6, it is treated as if it were annotated as `Basic`, as defined in Section 11.2.6, Section 11.2.19, Section 11.2.31, and Section 11.2.56.

If no annotation is present and neither of the above rules is applicable, the behavior is undefined. A persistence provider might report an error when the persistence unit is initialized.

2.11. Entity Relationships

Every relationship (or association) between two entity types has well-defined *multiplicity*, either:

- **one-to-one**,
- **one-to-many** / **many-to-one** (depending on which side the relationship is viewed from), or
- **many-to-many**.

Every attribute representing one side of such an association must be annotated with a *relationship modeling annotation*:

- `OneToOne` if the association has one-to-one multiplicity—in which case the declared type of the attribute must be an entity type,
- `ManyToOne` if the association has many-to-one multiplicity and the declared type of the attribute is an entity type—that is, if it is the many-valued (child) side of the association,
- `OneToMany` if the association has one-to-many multiplicity and the declared type of the attribute is a collection type—that is, if it is the single-valued (parent) side of the association, or
- `ManyToMany` if the association has many-to-many multiplicity—in which case the declared type of the attribute must be a collection type.

> These annotations mirror common practice in relational database schema modeling. The use of the relationship modeling annotations allows the object/relationship mapping of associations to the relational database schema to be fully defaulted, to provide an ease-of-development facility. This is described in Section 2.12.

For an attribute annotated `OneToOne` or `ManyToOne`, the declared type of the attribute is taken to be the associated entity type. If the declared type is not an entity type, the `targetClass` member must be explicitly specified, or this information must be supplied in the XML descriptor.

For an attribute annotated `OneToMany` or `ManyToMany`, the type of the attribute is a collection type.

- If the attribute has declared type `Collection<E>`, `Set<E>`, or `List<E>`, the associated entity type is taken to be the type argument `E`.
- If the attribute has declared type `Map<K,V>`, the associated entity type is taken to be the second type argument `V`.

If the inferred type `E` or `V` is not an entity type, or if the declared type of the attribute is a raw type, the `targetClass` member must be explicitly specified, or this information must be supplied in the XML descriptor.

Relationships are polymorphic. A relationship to an entity which has entity subclasses may be populated with an instance or instances of the subclass.

A relationship may be *bidirectional* or *unidirectional*.

- Every bidirectional relationship has both an *owning* side and an *inverse* (or *non-owning*, or *unowned*) side.
- A unidirectional relationship has only an owning side.

Modifications to the owning side of a relationship determine the updates made to the relationship in the database, as

described in Section 3.6.6. If the inverse side of an association is modified without a corresponding modification to the owning side, the behavior is undefined. The persistence provider is permitted to ignore any modification made only to the inverse side of a bidirectional association.

The following rules apply to bidirectional relationships:

The inverse side of a bidirectional relationship must identify the owning side by specifying the `mappedBy` element of the `OneToOne`, `OneToMany`, or `ManyToMany` annotation. The `mappedBy` element specifies the name of the property or field of the associated entity that represents the owning side of the relationship.

- The many-valued side of one-to-many / many-to-one bidirectional relationships must be the owning side; therefore, the `ManyToOne` annotation does not have a `mappedBy` element.
- The owning side of a one-to-one bidirectional relationship is declared by the entity mapped to the table that contains the corresponding foreign key column, unless the relationship maps to a join table, in which case either side may be declared as the owning side.
- A many-to-many bidirectional relationship always maps to a join table, and so either side may be declared as the owning side.

This specification recognizes two basic strategies for mapping an association:

- In a *foreign key mapping*, the relationship is mapped to a foreign key column in a (primary or secondary) table of one of the participating entities, known as the *child entity*.
- In a *join table mapping*, the relationship is mapped to a dedicated intermediate join table that represents the association itself.

Section 2.12 specifies default mappings for entity relationships. These defaulted mappings may be customized through the use of mapping annotations or via an XML descriptor. Furthermore, this specification requires support for the following additional mapping strategies:

- A unidirectional one-to-many relationship may be mapped to a foreign key column of the primary table of the target entity. The `JoinColumn` annotation found in Section 11.2.28 or equivalent XML element must be used to indicate this mapping strategy.
- A unidirectional or bidirectional one-to-one relationship, bidirectional many-to-one/one-to-many relationship, or unidirectional many-to-one relationship may be mapped to a join table. The `JoinTable` annotation found in Section 11.2.30 or corresponding XML element must be used to indicate this mapping strategy.

Annotations which customize the mapping of a relationship must be specified on the owning side of a bidirectional relationship. If a mapping annotation is specified on the inverse side of a bidirectional relationship, the behavior is undefined. A persistence provider might produce an error when the persistence unit is initialized.

Any overriding of mapping defaults must be consistent with the multiplicity implied by relationship modeling annotation. For example, a `ManyToMany` association must not be mapped to a join table with a unique key constraint on a foreign key.

The persistence provider handles the object/relational mapping of a relationship, including their persistence in the database, according to the metadata provided along with the entity class and the referential integrity constraints existing in the database (e.g., foreign key constraints).

> The application bears responsibility for maintaining the consistency of relationships—for example, the application must ensure that the “one” and “many” sides of a bidirectional relationship are consistent with one another when the application modifies either side at runtime.

When a `OneToMany` or `ManyToMany` relationship has no associated entities, the persistence provider must populate the

attribute with an empty collection when loading an entity from the database.

The `cascade` member of a relationship modeling annotation specifies a list of persistence operations which should be cascaded to the associated entity, as specified later in Section 3.6. An attribute annotated `ManyToOne` or `ManyToMany` must not specify `REMOVE` among its cascaded operation types. An application which specifies `cascade=REMOVE` for a `ManyToOne` or `ManyToMany` association is not portable.

2.12. Relationship Mapping Defaults

This section defines the mapping defaults that apply to entity associations declared using the relationship modeling annotations defined above in Section 2.11. These mapping defaults also apply when the XML descriptor is used to declare a relationship.

2.12.1. Bidirectional OneToOne Relationships

Assuming that:

- Entity A references a single instance of Entity B.
- Entity B references a single instance of Entity A.
- Entity A is specified as the owner of the relationship.

The following mapping defaults apply:

- Entity A is mapped to a table named `A`.
- Entity B is mapped to a table named `B`.
- Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; " _ "; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B` and there is a unique key constraint on it.

Example:

```
@Entity
public class Employee {
    private Cubicle assignedCubicle;

    @OneToOne
    public Cubicle getAssignedCubicle() {
        return assignedCubicle;
    }

    public void setAssignedCubicle(Cubicle cubicle) {
        this.assignedCubicle = cubicle;
    }

    // ...
}

@Entity
public class Cubicle {
    private Employee residentEmployee;

    @OneToOne(mappedBy="assignedCubicle")
    public Employee getResidentEmployee() {
        return residentEmployee;
    }

    public void setResidentEmployee(Employee employee) {
```

```
        this.residentEmployee = employee;
    }

    // ...
}
```

In this example:

- Entity `Employee` references a single instance of Entity `Cubicle`.
- Entity `Cubicle` references a single instance of Entity `Employee`.
- Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

- Entity `Employee` is mapped to a table named `EMPLOYEE`.
- Entity `Cubicle` is mapped to a table named `CUBICLE`.
- Table `EMPLOYEE` contains a foreign key to table `CUBICLE`. The foreign key column is named `ASSIGNEDCUBICLE_<PK of CUBICLE>`, where `<PK of CUBICLE>` denotes the name of the primary key column of table `CUBICLE`. The foreign key column has the same type as the primary key of `CUBICLE`, and there is a unique key constraint on it.

2.12.2. Bidirectional ManyToOne / OneToMany Relationships

Assuming that:

- Entity A references a single instance of Entity B.
- Entity B references a collection of Entity A[24].
- Entity A must be the owner of the relationship.

The following mapping defaults apply:

- Entity A is mapped to a table named `A`.
- Entity B is mapped to a table named `B`.
- Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; " _ "; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B`.

Example:

```
@Entity
public class Employee {
    private Department department;

    @ManyToOne
    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }

    // ...
}

@Entity
public class Department {
    private Collection<Employee> employees = new HashSet();
```

```
    @OneToMany(mappedBy="department")
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }

    // ...
}
```

In this example:

- Entity `Employee` references a single instance of Entity `Department`.
- Entity `Department` references a collection of Entity `Employee`.
- Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

- Entity `Employee` is mapped to a table named `EMPLOYEE`.
- Entity `Department` is mapped to a table named `DEPARTMENT`.
- Table `EMPLOYEE` contains a foreign key to table `DEPARTMENT`. The foreign key column is named `DEPARTMENT_<PK of DEPARTMENT>`, where `<PK of DEPARTMENT>` denotes the name of the primary key column of table `DEPARTMENT`. The foreign key column has the same type as the primary key of `DEPARTMENT`.

2.12.3. Unidirectional Single-Valued Relationships

Assuming that:

- Entity A references a single instance of Entity B.
- Entity B does not reference Entity A.

A unidirectional relationship has only an owning side, which in this case must be Entity A.

The unidirectional single-valued relationship modeling case can be specified as either a unidirectional `OneToOne` or as a unidirectional `ManyToOne` relationship.

2.12.3.1. Unidirectional OneToOne Relationships

The following mapping defaults apply:

- Entity A is mapped to a table named `A`.
- Entity B is mapped to a table named `B`.
- Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; " _ "; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B` and there is a unique key constraint on it.

Example:

```
@Entity
public class Employee {
    private TravelProfile profile;

    @OneToOne
    public TravelProfile getProfile() {
```

```
        return profile;
    }

    public void setProfile(TravelProfile profile) {
        this.profile = profile;
    }

    // ...
}

@Entity
public class TravelProfile {
    // ...
}
```

In this example:

- Entity `Employee` references a single instance of Entity `TravelProfile`.
- Entity `TravelProfile` does not reference Entity `Employee`.
- Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

- Entity `Employee` is mapped to a table named `EMPLOYEE`.
- Entity `TravelProfile` is mapped to a table named `TRAVELPROFILE`.
- Table `EMPLOYEE` contains a foreign key to table `TRAVELPROFILE`. The foreign key column is named `PROFILE_<PK of TRAVELPROFILE>`, where `<PK of TRAVELPROFILE>` denotes the name of the primary key column of table `TRAVELPROFILE`. The foreign key column has the same type as the primary key of `TRAVELPROFILE`, and there is a unique key constraint on it.

2.12.3.2. Unidirectional ManyToOne Relationships

The following mapping defaults apply:

- Entity A is mapped to a table named `A`.
- Entity B is mapped to a table named `B`.
- Table `A` contains a foreign key to table `B`. The foreign key column name is formed as the concatenation of the following: the name of the relationship property or field of entity A; "_"; the name of the primary key column in table `B`. The foreign key column has the same type as the primary key of table `B`.

Example:

```
@Entity
public class Employee {
    private Address address;

    @ManyToOne
    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    // ...
}

@Entity
public class Address {
    // ...
```

```
}
```

In this example:

- Entity `Employee` references a single instance of Entity `Address`.
- Entity `Address` does not reference Entity `Employee`.
- Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

- Entity `Employee` is mapped to a table named `EMPLOYEE`.
- Entity `Address` is mapped to a table named `ADDRESS`.
- Table `EMPLOYEE` contains a foreign key to table `ADDRESS`. The foreign key column is named `ADDRESS_<PK of ADDRESS>`, where `<PK of ADDRESS>` denotes the name of the primary key column of table `ADDRESS`. The foreign key column has the same type as the primary key of `ADDRESS`.

2.12.4. Bidirectional ManyToMany Relationships

Assuming that:

- Entity A references a collection of Entity B.
- Entity B references a collection of Entity A.
- Entity A is the owner of the relationship.

The following mapping defaults apply:

- Entity A is mapped to a table named `A`.
- Entity B is mapped to a table named `B`.
- There is a join table that is named `A_B` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `A` and has the same type as the primary key of table `A`. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity B; " _ "; the name of the primary key column in table `A`. The other foreign key column refers to table `B` and has the same type as the primary key of table `B`. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity A; " _ "; the name of the primary key column in table `B`.

Example:

```java
@Entity
public class Project {
    private Collection<Employee> employees;

    @ManyToMany
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }

    // ...
}

@Entity
public class Employee {
    private Collection<Project> projects;
```

```
    @ManyToMany(mappedBy="employees")
    public Collection<Project> getProjects() {
        return projects;
    }

    public void setProjects(Collection<Project> projects) {
        this.projects = projects;
    }

    // ...
}
```

In this example:

- Entity `Project` references a collection of Entity `Employee`.
- Entity `Employee` references a collection of Entity `Project`.
- Entity `Project` is the owner of the relationship.

The following mapping defaults apply:

- Entity `Project` is mapped to a table named `PROJECT`.
- Entity `Employee` is mapped to a table named `EMPLOYEE`.
- There is a join table that is named `PROJECT_EMPLOYEE` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `PROJECT` and has the same type as the primary key of `PROJECT`. The name of this foreign key column is `PROJECTS_<PK of PROJECT>`, where `<PK of PROJECT>` denotes the name of the primary key column of table `PROJECT`. The other foreign key column refers to table `EMPLOYEE` and has the same type as the primary key of `EMPLOYEE`. The name of this foreign key column is `EMPLOYEES_<PK of EMPLOYEE>`, where `<PK of EMPLOYEE>` denotes the name of the primary key column of table `EMPLOYEE`.

2.12.5. Unidirectional Multi-Valued Relationships

Assuming that:

- Entity A references a collection of Entity B.
- Entity B does not reference Entity A.

A unidirectional relationship has only an owning side, which in this case must be Entity A.

The unidirectional multi-valued relationship modeling case can be specified as either a unidirectional `OneToMany` or as a unidirectional `ManyToMany` relationship.

2.12.5.1. Unidirectional OneToMany Relationships

The following mapping defaults apply:

- Entity A is mapped to a table named `A`.
- Entity B is mapped to a table named `B`.
- There is a join table that is named `A_B` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `A` and has the same type as the primary key of table `A`. The name of this foreign key column is formed as the concatenation of the following: the name of entity A; " _ "; the name of the primary key column in table `A`. The other foreign key column refers to table `B` and has the same type as the primary key of table `B` and there is a unique key constraint on it. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity A; " _ "; the name of the primary key column in table `B`.

Example:

```
@Entity
public class Employee {
    private Collection<AnnualReview> annualReviews;

    @OneToMany
    public Collection<AnnualReview> getAnnualReviews() {
        return annualReviews;
    }

    public void setAnnualReviews(Collection<AnnualReview> annualReviews) {
        this.annualReviews = annualReviews;
    }

    // ...
}

@Entity
public class AnnualReview {
    // ...
}
```

In this example:

- Entity `Employee` references a collection of Entity `AnnualReview`.
- Entity `AnnualReview` does not reference Entity `Employee`.
- Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

- Entity `Employee` is mapped to a table named `EMPLOYEE`.
- Entity `AnnualReview` is mapped to a table named `ANNUALREVIEW`.
- There is a join table that is named `EMPLOYEE_ANNUALREVIEW` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `EMPLOYEE` and has the same type as the primary key of `EMPLOYEE`. This foreign key column is named `EMPLOYEE_<PK of EMPLOYEE>`, where `<PK of EMPLOYEE>` denotes the name of the primary key column of table `EMPLOYEE`. The other foreign key column refers to table `ANNUALREVIEW` and has the same type as the primary key of `ANNUALREVIEW`. This foreign key column is named `ANNUALREVIEWS_<PK of ANNUALREVIEW>`, where `<PK of ANNUALREVIEW>` denotes the name of the primary key column of table `ANNUALREVIEW`. There is a unique key constraint on the foreign key that refers to table `ANNUALREVIEW`.

2.12.5.2. Unidirectional ManyToMany Relationships

The following mapping defaults apply:

- Entity `A` is mapped to a table named `A`.
- Entity `B` is mapped to a table named `B`.
- There is a join table that is named `A_B` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `A` and has the same type as the primary key of table A. The name of this foreign key column is formed as the concatenation of the following: the name of entity `A`; " _ "; the name of the primary key column in table `A`. The other foreign key column refers to table `B` and has the same type as the primary key of table `B`. The name of this foreign key column is formed as the concatenation of the following: the name of the relationship property or field of entity `A`; " _ "; the name of the primary key column in table `B`.

Example:

```
@Entity
```

```
public class Employee {
    private Collection<Patent> patents;

    @ManyToMany
    public Collection<Patent> getPatents() {
        return patents;
    }

    public void setPatents(Collection<Patent> patents) {
        this.patents = patents;
    }

    // ...
}

@Entity
public class Patent {
    //...
}
```

In this example:

- Entity `Employee` references a collection of Entity `Patent`.
- Entity `Patent` does not reference Entity `Employee`.
- Entity `Employee` is the owner of the relationship.

The following mapping defaults apply:

- Entity `Employee` is mapped to a table named `EMPLOYEE`.
- Entity `Patent` is mapped to a table named `PATENT`.
- There is a join table that is named `EMPLOYEE_PATENT` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `EMPLOYEE` and has the same type as the primary key of `EMPLOYEE`. This foreign key column is named `EMPLOYEE_<PK of EMPLOYEE>`, where `<PK of EMPLOYEE>` denotes the name of the primary key column of table `EMPLOYEE`. The other foreign key column refers to table `PATENT` and has the same type as the primary key of `PATENT`. This foreign key column is named `PATENTS_<PK of PATENT>`, where `<PK of PATENT>` denotes the name of the primary key column of table `PATENT`.

2.13. Inheritance

An entity class might inherit another entity class. If this is the case, then:

- any association targeting the inherited entity class is by nature a polymorphic association, and may hold a reference to an instance of the inheriting entity class, and
- any Jakarta Persistence query against the inherited entity class is also polymorphic and may return an instance of the inheriting entity class.

An abstract class may be annotated `Entity` or declared as an `entity` in its XML descriptor. An abstract entity class may participate in a relationship, and may occur in the `from` clause of a query. Abstract entity classes are discussed below in Section 2.13.1.

An entity might extend a non-entity class, and a non-entity class might extend an entity class. Such a non-entity class might be declared as a mapped superclass defining common persistent state and object/relational mapping information inherited by its entity subclasses, as discussed below in Section 2.13.2, but, as described below in Section 2.13.3, it is not required to be. If a non-entity class defines persistent state, it must be declared as a mapped superclass.

2.13.1. Abstract Entity Classes

An abstract entity differs from a concrete entity only in that it cannot be directly instantiated. This specification defines no special restrictions applying only to abstract entity classes.

The following example shows the use of an abstract entity class in an entity inheritance hierarchy.

Example: Abstract class as an Entity

```
@Entity
@Table(name="EMP")
@Inheritance(strategy=JOINED)
public abstract class Employee {
    @Id
    protected Integer empId;

    @Version
    protected Integer version;

    @ManyToOne
    protected Address address;

    // ...
}

@Entity
@Table(name="FT_EMP")
@DiscriminatorValue("FT")
@PrimaryKeyJoinColumn(name="FT_EMPID")
public class FullTimeEmployee extends Employee {
    // Inherit empId, but mapped in this class to FT_EMP.FT_EMPID
    // Inherit version mapped to EMP.VERSION
    // Inherit address mapped to EMP.ADDRESS fk

    // Defaults to FT_EMP.SALARY
    protected Integer salary;

    // ...
}

@Entity
@Table(name="PT_EMP")
@DiscriminatorValue("PT")
// PK column is PT_EMP.EMPID due to `PrimaryKeyJoinColumn` default
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;

    // ...
}
```

2.13.2. Mapped Superclasses

An entity might inherit a superclass that declares persistent entity state and mapping information, but which is not itself an entity. Typically, the purpose of such a *mapped superclass* is to define state and mapping information that is common across multiple entity classes but which maps to disparate, unrelated database tables.

A mapped superclass must be annotated with the `MappedSuperclass` annotation or declared as a `mapped-superclass` in its XML descriptor.

A mapped superclass might be an abstract class or a concrete class.

A mapped superclass, unlike an entity, is not queryable and must not be passed as an argument to `EntityManager`,

`EntityAgent`, or `Query` operations.

Persistent relationships involving a mapped superclass must be unidirectional.

- A mapped superclass may declare a unidirectional association to an entity class, but
- the mapped superclass must not be the target of an association declared by any other class.

A mapped superclass never maps to a database table. The persistent state inherited by an entity from a mapped superclass maps to a table determined by the entity itself, as if the attributes of the mapped superclass were declared by the entity which most directly inherits the mapped superclass.

The persistent attributes of a mapped superclass may be mapped in the same way as the attributes of an entity class. Such mappings are meaningful only in the context of an entity subclass of the mapped superclass, since no table exists for the mapped superclass itself. From the point of view of such an entity subclass, the inherited mappings are interpreted in the context of the tables mapped by the subclass. Mapping information inherited from a mapped superclass can be overridden in such subclasses using the `AttributeOverride` and `AssociationOverride` annotations or corresponding XML elements.

All other entity mapping defaults apply equally to a mapped superclass.

The following example illustrates the use of a concrete class as a mapped superclass.

Example: Concrete class as a mapped superclass

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer empId;

    @Version
    protected Integer version;

    @ManyToOne
    @JoinColumn(name="ADDR")
    protected Address address;

    public Integer getEmpId() { ... }

    public void setEmpId(Integer id) { ... }

    public Address getAddress() { ... }

    public void setAddress(Address addr) { ... }
}

// Default table is FTEMPLOYEE table
@Entity
public class FTEmployee extends Employee {
    // Inherited empId field mapped to FTEMPLOYEE.EMPID
    // Inherited version field mapped to FTEMPLOYEE.VERSION
    // Inherited address field mapped to FTEMPLOYEE.ADDR fk

    // Defaults to FTEMPLOYEE.SALARY
    protected Integer salary;

    public FTEmployee() {}

    public Integer getSalary() { ... }

    public void setSalary(Integer salary) { ... }
}

@Entity
```

```
@Table(name="PT_EMP")
@AssociationOverride(name="address", joincolumns=@JoinColumn(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {
    // Inherited empId field mapped to PT_EMP.EMPID
    // Inherited version field mapped to PT_EMP.VERSION
    // address field mapping overridden to PT_EMP.ADDR_ID fk
    @Column(name="WAGE")
    protected Float hourlyWage;

    public PartTimeEmployee() {}

    public Float getHourlyWage() { ... }

    public void setHourlyWage(Float wage) { ... }
}
```

2.13.3. Non-Entity Classes in the Entity Inheritance Hierarchy

An entity might inherit a non-entity superclass, which might be concrete or abstract.[25]

The non-entity superclass serves for inheritance of behavior only. The state of a non-entity superclass is not persistent. Any state inherited from a non-entity superclass is treated as if it were declared `@Transient` by the inheriting entity class. This non-persistent state is not managed by the entity manager[26].

When a non-entity class is annotated with an annotation type defined by this specification, the behavior is undefined. The persistence provider might simply ignore such annotations.

A non-entity type may not be used as an entity type, even if it inherits an entity type:

- A non-entity class cannot be passed as an argument to a parameter of a method of `EntityManager` or `EntityAgent` which expects an entity type.
- If `object.getClass()` return a non-entity class, then `object` cannot be passed as an argument to a parameter of a method that expects an instance of an entity type, even if `object` is assignable to an entity type.
- A non-entity type may not be used in JPQL as the abstract schema type of a range variable declaration, as defined in Section 4.4.3.

The following example illustrates the use of a non-entity class as a superclass of an entity class.

Example: Non-entity superclass

```
public class Cart {
    protected Integer operationCount; // transient state

    public Cart() {
        operationCount = 0;
    }

    public Integer getOperationCount() {
        return operationCount;
    }

    public void incrementOperationCount() {
        operationCount++;
    }
}

@Entity
public class ShoppingCart extends Cart {
    Collection<Item> items = new Vector<Item>();

    public ShoppingCart() {
```

```
        super();
    }

    // ...

    @OneToMany
    public Collection<Item> getItems() {
        return items;
    }

    public void addItem(Item item) {
        items.add(item);
        incrementOperationCount();
    }
}
```

2.14. Inheritance Mapping Strategies

The mapping of entity class hierarchies to database tables is specified through metadata.

There are three basic strategies which may be used when mapping an entity class hierarchy to tables in a relational database.

- With a **single table per class hierarchy**, every field declared by any class in the hierarchy maps to a column of the same table. A subclass shares the table of its superclasses.
- With a **table per concrete entity class**, every field declared or inherited by a given subclass is mapped to the same table, but each concrete class in the hierarchy has its own separate table.
- For the **joined subclass** strategy, any field which is declared by a subclass is mapped to a separate table from the fields which are declared by its superclass. Table joins may be used to retrieve all fields declared and inherited by the subclass.

These mapping strategies are enumerated by `InheritanceType`, and the mapping strategy for an entity class hierarchy may be specified by the `@Inheritance` annotation or `inheritance` element of an XML descriptor, as defined by Section 11.2.27. The inheritance strategy for the whole entity class hierarchy is determined by the mapping of the root class of the hierarchy. If the `@Inheritance` annotation is used, it must be placed on the root class.

An implementation of this specification is required to support the single table per class hierarchy inheritance mapping strategy and the joined subclass strategy.

> Support for the table per concrete class inheritance mapping strategy is optional in this release. Applications that use this mapping strategy are not portable between all providers.

Support for the combination of inheritance strategies within a single entity inheritance hierarchy is not required by this specification.

2.14.1. Single Table per Class Hierarchy Strategy

In this strategy, all classes in a hierarchy map to the same table. The table has a column serving as a *discriminator column*, that is, a column whose value identifies the specific subclass to which the entity instance that is represented by a row of the table belongs.

This mapping strategy provides good support for polymorphic relationships between entities and for queries that range over the class hierarchy.

Its drawback is that it requires that columns mapped by attributes of subclasses be nullable.

2.14.2. Joined Subclass Strategy

In this strategy:

- The root of the class hierarchy is represented by a table with columns mapped by attributes declared by the root class.
- Each subclass is represented by a separate table containing only those columns that are mapped by attributes declared by that subclass (and not inherited from its superclass), along with the column or columns holding its primary key. The primary key of the subclass table doubles as a foreign key of the superclass table.

This strategy provides good support for polymorphic relationships between entities and is very well normalized.

Its drawback is that it requires at least one `JOIN` operation to retrieve instances of a subclass. In a very deep class hierarchy, this might lead to poor performance. Queries that range over the class hierarchy likewise require joins.

2.14.3. Table per Concrete Class Strategy

In this mapping strategy, each concrete class in the hierarchy is mapped to a completely separate table. Every attribute of the class, including inherited attributes, maps to a column or columns of the table belonging to the concrete class.

This strategy has the following drawbacks:

- It provides very poor support for polymorphic relationships.
- It requires the use of a SQL `UNION` query (or multiple SQL queries) when a query ranges over the class hierarchy.

This inheritance mapping strategy is not required in the current release of this specification.

2.14.4. Duplicate Attributes in Inheritance Hierarchies

An entity class must not have two persistent attributes—declared or inherited—with the same name. If an entity class or mapped superclass declares a persistent attribute with the same name as a persistent attribute inherited from another entity or mapped superclass, the behavior is undefined and is not portable between providers.

- When `FIELD` access is used, as defined above in Section 2.3, persistent fields declared by distinct classes are always considered distinct attributes.
- When `PROPERTY` access is used, as discussed in Section 2.2.3, persistent properties declared by distinct classes are considered distinct if and only if the property accessors of the persistent property of the subtype do not override the property accessors of the supertype.

When a persistent property of a subclass overrides a persistent property of a superclass, the property of the subclass inherits its mapping from the property of a superclass. The behavior when mapping annotations are placed in the property of the subclass is undefined.

Example (allowed):

```
@MappedSuperclass
@Access(AccessType.PROPERTY)
public abstract class Person {
    @Id
    private Long id;

    @Basic(optional=false)
    @Column(name="PERSON_NAME")
    public abstract String getName();

    public abstract void setName(String name);
}
```

```java
@Entity
@Access(AccessType.PROPERTY)
public class Employee extends Person {
    private String name;

    // Allowed: mapping is inherited from Person.getName().
    @Override
    public String getName() { return name; }

    @Override
    public void setName(String name) { this.name = name; }
}
```

Example (undefined behavior):

```java
@MappedSuperclass
@Access(AccessType.PROPERTY)
public abstract class Person {
    @Id
    private Long id;

    @Basic(optional=false)
    @Column(name="PERSON_NAME")
    public abstract String getName();

    public abstract void setName(String name);
}

@Entity
@Access(AccessType.PROPERTY)
public class Employee extends Person {
    private String name;

    // Undefined: mapping annotations on overriding property
    @Column(name="EMP_NAME")
    @Override
    public String getName() { return name; }

    @Override
    public void setName(String name) { this.name = name; }
}
```

2.14.5. Duplicate Column Mappings

Two persistent attributes (declared by the same entity class or by distinct entity classes) are permitted to map the same column of a given table if and only if:

- the persistent attributes have exactly the same type, and
- either the two attributes are declared in distinct branches of the entity class hierarchy, so that no entity class features both attributes, or one of the persistent attributes is declared with `insertable=false, updatable=false`, as defined later in Chapter 10.

It is the responsibility of the application developer to satisfy this restriction. In an entity class hierarchy mapped via the single table strategy or table per concrete class strategy, the developer might need to explicitly specify distinct column mappings for persistent attributes with identical names.

The behavior is undefined and unportable if:

- any two distinct persistent attributes with distinct types map the same column of a given table, or
- two distinct persistent attributes declared or inherited by a given entity class map the same column of a given table

and neither is mapped `insertable=false, updatable=false`.

Example (allowed):

```
@Entity
@Inheritance(strategy = SINGLE_TABLE)
public class Person {
    @Id
    private Long id;
}

@Entity
public class Employee extends Person {
    @Column(name="EMPLOYEE_CODE", length=8)
    private String code;
}

@Entity
public class Customer extends Person {
    @Column(name="CUSTOMER_CODE")
    private int code;
}
```

Example (disallowed):

```
@Entity
@Inheritance(strategy = SINGLE_TABLE)
public class Person {
    @Id
    private Long id;
}

@Entity
public class Employee extends Person {
    @Column(length=8)
    private String code;
}

@Entity
public class Customer extends Person {
    // Disallowed: maps to the same CODE column as Employee.code
    private int code;
}
```

Example (allowed):

```
@Entity
public class Job {
    @Id
    private Long id;

    @Column(name="STATUS")
    private String status;

    // Allowed: read-only copy of STATUS
    @Column(name="STATUS", insertable=false, updatable=false)
    private String statusReadOnly;
}
```

2.15. Naming of Database Objects

This specification defines many annotations with elements referencing database objects by name, and rules defining

the default name when such an annotation element is not explicitly specified. These names include the names of tables, columns, sequences, stored procedures, and so on.

By default, the name of a database object must be treated as an undelimited (unquoted) identifier and passed to the database as such.

For example, assuming the use of an English locale, the following table names must be passed to the database as undelimited identifiers so that they will be treated as equivalent by all databases that comply with the requirements of the SQL Standard for the treatment of “regular identifiers” and “delimited identifiers” [2]:

```
@Table(name="Customer")
@Table(name="customer")
@Table(name="cUsTomer")
```

Similarly, the following column names should be treated as equivalent:

```
@JoinColumn(name="CUSTOMER")
@ManyToOne Customer customer;

@JoinColumn(name="customer")
@ManyToOne Customer customer;

@ManyToOne Customer customer;
```

This behavior relieves the developer of the need to explicitly specify a table or column name when the name differs only in case from the defaulted name derived from the name of the mapped entity or attribute.

There are two ways to specify that a delimited (quoted) identifier should be passed to the database.

- The `<delimited-identifiers/>` element within the `<persistence-unit-defaults>` element of the object/relational XML mapping file specifies that all database identifiers in use by a persistence unit must be treated as delimited identifiers. If the `<delimited-identifiers/>` element is specified, it cannot be overridden.
- Alternatively, when the `<delimited-identifiers>` element is not specified, and the name of a database object given by an annotation member or in an XML mapping is enclosed within escaped double quotes, for example,
  - `@Table(name="\"customer\"")`, or
  - `<table name=""customer""/>`,

  the name is treated as a delimited identifier.

When `<delimited-identifiers>` is specified, every escaped double quote within an annotation member or XML element or attribute is interpreted literally as part of the name of the database identifier.

The following annotations contain elements whose values correspond to names of database identifiers and for which the above rules apply, including when their use is nested within that of other annotations:

- `EntityResult(discriminatorColumn element)`
- `FieldResult(column element)`
- `ColumnResult(name element)`
- `CollectionTable(name, catalog, schema elements)`
- `Column(name, table elements)`
- `DiscriminatorColumn(name elements)`
- `ForeignKey(name, foreignKeyDefinition elements)`
- `Index(name, columnList elements)`
- `JoinColumn(name, referencedColumnName, table elements)`

- `JoinTable(name, catalog, schema elements)`
- `MapKeyColumn(name, table elements)`
- `MapKeyJoinColumn(name, referencedColumnName, table elements)`
- `NamedStoredProcedureQuery(procedureName element)`
- `OrderColumn(name elements)`
- `PrimaryKeyJoinColumn(name, referencedColumnName elements)`
- `SecondaryTable(name, catalog, schema elements)`
- `SequenceGenerator(sequenceName, catalog, schema elements)`
- `StoredProcedureParameter(name element)`
- `Table(name, catalog, schema elements)`
- `TableGenerator(table, catalog, schema, pkColumnName, valueColumnName elements)`
- `UniqueConstraint(name, columnNames elements)`
- `CheckConstraint(name elements)`

The following XML elements and types contain elements or attributes whose values correspond to names of database identifiers and for which the above rules apply:

- `entity-mappings(schema, catalog elements)`
- `persistence-unit-defaults(schema, catalog elements)`
- `collection-table(name, catalog, schema attributes)`
- `column(name, table attributes)`
- `column-result(name attribute)`
- `discriminator-column(name attributes)`
- `entity-result(discriminator-column attribute)`
- `field-result(column attribute)`
- `foreign-key(name, foreign-key-definition attributes)`
- `index(name attribute, column-list element)`
- `join-column(name, referenced-column-name, table attributes)`
- `join-table(name, catalog, schema attributes)`
- `map-key-column(name, table attributes)`
- `map-key-join-column(name, referenced-column-name, table attributes)`
- `named-stored-procedure-query(procedure-name attribute)`
- `order-column(name attributes)`
- `primary-key-join-column(name, referenced-column-name attributes)`
- `secondary-table(name, catalog, schema attributes)`
- `sequence-generator(sequence-name, catalog, schema attributes)`
- `stored-procedure-parameter(name attribute)`
- `table(name, catalog, schema attributes)`
- `table-generator(table, catalog, schema, pk-column-name, value-column-name attributes)`
- `unique-constraint(name attribute, column-name element)`

[1] An entity instance is a local object inaccessible to remote processes. If instances of an entity are to be passed by value as detached objects (e.g., via a remote interface), the entity class must be serializable.

[2] The term "persistence provider runtime" refers to the runtime environment of the persistence implementation. In a JakartaEE environment, this might be the Jakarta EE container itself, or a third-party persistence provider implementation integrated with the container.

[3] When the constructor is added by tooling or by the container, the entity source code is not portable to environments where such tooling is unavailable.

[4] These annotations must not be applied to the setter methods.

[5] Portable applications should not expect the order of a list to be maintained across persistence contexts unless the `OrderColumn` or `OrderBy` annotation is used and modifications to the list observe the specified ordering.

[6] A persistence provider is permitted—but not required—to accept the combinations `@Basic @ElementCollection` and `@Embedded @ElementCollection`.

[7] Specifically, if `getX` is the name of the getter method and `setX` is the name of the setter method, where `X` is any string, the name of the persistent property is obtained by calling `java.beans.Introspector.decapitalize(X)`.

[8] Lazy fetching is a hint to the persistence provider and can be specified by means of the `Basic`, `OneToOne`, `OneToMany`, `ManyToOne`, `ManyToMany`, and `ElementCollection` annotations and their XML equivalents. See Chapter 11.

[9] The use of XML as an alternative and the interaction between Java language annotations and XML elements in defining default and explicit access types is described in Chapter 12.

[10] An `Access` annotation of a field or property getter is considered a "mapping annotation" for the purposes of this section. Therefore, an attribute-level `Access` annotation may not be used to selectively override the access type of an attribute of an entity class with a defaulted access type.

[11] Composite primary keys often arise when mapping a legacy database with primary keys comprising multiple columns.

[12] Although tolerated by Jakarta Persistence, approximate numeric types (floating point types) should never be used in primary keys.

[13] This includes not changing the value of a mutable type that is primary key or an attribute of a composite primary key.

[14] The implementation may, but is not required to, throw an exception. Portable applications must not rely on any such specific behavior.

[15] If the application does not set a primary key attribute mapped to the same column or columns as the relationship, the value of that attribute might not be available until after the entity has been flushed to the database.

[16] The primary key of the parent might be represented as an embedded id or as an id class.

[17] Note that the use of PrimaryKeyJoinColumn instead of MapsId would result in the same mapping in this example. Use of MapsId is preferred for the mapping of derived identities.

[18] Bulk update statements, however, are permitted to set the version of an entity. See Section 4.11.

[19] Note that an instance of `Calendar` must be fully initialized for the SQL type it maps.

[20] The use of `java.util.Calendar` or of `java.util.Date` is strongly discouraged. Newly written programs should use the date/time types defined in the package `java.time`.

[21] The use of date/time types defined in the package `java.sql` is strongly discouraged. Newly written programs should use the date/time types defined in `java.time`.

[22] An entity cannot have a unidirectional relationship to an embedded object belonging to another entity.

[23] Note that when an embeddable instance is used as a map key, these attributes represent its identity. Modification of an embeddable instance used as a map key has undefined semantics and should be avoided.

[24] When the relationship is modeled as a `java.util.Map`, “Entity B references a collection of Entity A” means that Entity B references a map collection in which the type of the Map `value` is Entity A. The map key may be a basic type, embeddable class, or an entity.

[25] The superclass must not be an embeddable class or id class.

[26] If a transaction-scoped persistence context is used, it is not required to be retained across transactions.

Chapter 3. Entity Operations

This chapter describes:

- the use of the `EntityManager`, `EntityAgent`, and `Query` APIs to retrieve instances of entity classes representing persistent state held in the database, and of `EntityGraph` to control the limits of the object graph returned by such operations,
- the use of the `EntityManager` API to manage the lifecycle of entity instances associated with a persistence context, and to control the synchronization of state held in the persistence context with the database,
- the use of the `EntityAgent` API to directly update persistent state held in the database *without* the use of a persistence context,
- the use of the second-level cache, and
- entity listeners and lifecycle callbacks, attribute converters, and integration with Bean Validation.

3.1. Overview

Every instance of `EntityManager` has an associated *persistence context*. A persistence context is a set of entity instances in which for any given persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed. The entity instance lifecycle is defined in Section 3.6. The relationship between entity managers and persistence contexts is described in Section 3.7, and again in further detail in Chapter 7.

The `EntityManager` interface defines the methods used to interact with its persistence context. The `EntityManager` API is used to create and remove persistent entity instances, to find persistent entities by primary key, and to query over persistent entity types. Section 3.5 describes the `EntityManager` interface and Section 3.6.1 lists its fundamental operations. Section 3.9 describes mechanisms for concurrency control and locking. Section 3.16 provides a summary of exceptions.

When a persistence context is not required, `EntityAgent` offers an alternative way to interact with the database. The operations of `EntityAgent` accept entity instances which are not associated with any persistence context. The lifecycle of these instances is not managed by the persistence provider. Section 3.8 describes the `EntityAgent` interface.

The `EntityManager` or `EntityAgent` acts as a factory for instances of `Query`, which are used to control query execution. The `Query` interface, along with its subtypes `Statement`, `TypedQuery`, `StoredProcedureQuery`, and related interfaces are described in Section 3.15. The Jakarta Persistence query language is defined in Chapter 4 and APIs for the construction of Criteria queries in Chapter 6. Section 3.12 describes the use of entity graphs to control and limit the data fetched during find and query operations.

Each `EntityManager` or `EntityAgent` belongs to an `EntityManagerFactory` with an associated *persistence unit*. A persistence unit defines a set of related entities which map to a single database. Entities belonging to the same persistence unit may participate in associations. An `EntityManager` or `EntityAgent` handles instances of entities belonging to its persistence unit. The definition of persistence units is described in Chapter 8. An `EntityManagerFactory` might have an associated second-level cache. Section 3.14 describes mechanisms for portable configuration of the second-level cache.

Jakarta Persistence features several mechanisms allowing user-written code to react to events occurring within the persistence context. Section 3.10 describes entity listeners and lifecycle callback methods for entities. Section 3.11 describes support for automatic use of Bean Validation. Section 3.12 describes mechanisms for defining conversions between entity and database representations for attributes of basic types.

3.2. Entities and Database Records

A *persistent entity identity* is defined by an entity type and a primary key value, as specified above in Section 2.4 and below in Section 3.6.10. An entity is called *managed* if it belongs to a persistence context. Every managed entity instance

has a persistent identity. The managed instance retains its persistent identity when its association with the persistence context is destroyed, as defined below in Section 3.6.9, and the entity becomes *detached*.

Persistent identity determines a mapping from managed entity instances to *records* held in the database. In this specification, the term "record" refers collectively to one or more relational table rows:

- the row held in the *primary table* mapped by the entity class, as specified in Section 11.2.53,

along with additional rows related to this row by foreign key associations and held in:

- *secondary tables*, as specified in Section 11.2.49,
- tables mapped by element collections, as specified in Section 11.2.9, and
- tables mapped by owned relationships, as specified in Section 11.2.30.

A record may be uniquely identified by the primary key value of its row in the primary table. We say that an entity instance *represents* the record if the primary key of the entity instance matches the primary key value of the row belonging to the record in the primary table. [1]

We say that an entity instance is *persistent* if it represents a database record or if insertion of a record has been scheduled with the entity manager.

3.3. Properties, Hints, and Options

The `EntityManager` and `EntityAgent` interfaces—along with some other interfaces defined by this specification—contain methods that take "properties" or "hints" as arguments. A property or hint is a named value, with the name represented as a string. This specification distinguishes between properties and hints as follows:

- A *property* defined by this specification must be observed by the provider unless otherwise explicitly stated.
- A *hint* specifies a preference on the part of the application. While a hint defined by this specification should be observed by the provider if possible, a hint may or may not always be observed. A portable application must not depend on the observance of a hint.

Alternatively, an *option* is a type safe object carrying the same sort of information which might be provided in a property or hint. The interfaces `EntityManager` and `EntityAgent` define inner interfaces `Option` and `CreationOption`, and this specification provides several built-in concrete option types implementing these option interfaces. Providers are encouraged to define additional custom option types controlling vendor-specific behavior.

> Custom options are type safe and result in more elegant code. However, their use results in a dependency on vendor-specific APIs. Application programs desiring portability between persistence providers might choose to use string-based properties and hints in preference to custom options.

3.4. EntityHandler Interface

Operations common to `EntityManager` and `EntityAgent` are declared by the interface `EntityHandler`, which may be found in Section B.1.

The methods `close()` and `isOpen()` are used to control the lifecycle of application-managed entity managers and entity agents, as defined in Section 7.8. The method `getTransaction()` returns an `EntityTransaction` which may be used to control resource-local transactions as specified in Section 7.5. Any method of `EntityManager` and `EntityAgent` except for `isOpen()`, `getTransaction()`, and `getProperties()` throws `IllegalStateException` if the entity manager or entity agent has been closed.

Any `Query`, `CriteriaBuilder`, `Metamodel`, or `EntityTransaction` object obtained from a given entity manager or entity agent is valid only while that entity manager or entity agent is open. Any method of an instance of `Query` obtained from an entity

manager or entity agent which has been closed throws `IllegalStateException`.

The method `createQuery()` is used to obtain a `Query` or `TypedQuery` representing a Jakarta Persistence Query Language query or Criteria query. Similarly, the method `createStatement()` is used to obtain a `Statement`.

The persistence provider is permitted but not required to validate a JPQL query or statement when `createQuery()` or `createStatement()` is called. An `IllegalArgumentException` might be thrown if:

- the argument to `createQuery()` or `createStatement()` is not a legal Jakarta Persistence Query Language query or statement,
- the result class specification of a Jakarta Persistence Query Language query is incompatible with the result of the query, or
- the argument to `createQuery()` is not a valid `CriteriaQuery` object.

Alternatively, `createQuery()` or `createStatement()` might return a result, and then execution of the query might later fail. In that case, a `PersistenceException` is thrown when the query is executed.

The method `createNativeQuery()` is used to obtain a `Query` or `TypedQuery` representing a native SQL query. Similarly, the method `createNativeStatement()` is used to obtain a `Statement`. A `PersistenceException` is thrown when the native query is executed if:

- the query or statement is rejected by the database server, or
- the result set specification is incompatible with the result set of the SQL query.

The `PersistenceException` should wrap the underlying database exception if possible.

3.5. EntityManager Interface

The `EntityManager` interface may be found in Section B.2.

In the Java SE environment, each instance of `EntityManager` is associated with a distinct *persistence context*. A persistence context is a set of entity instances in which there is at most one entity instance with any given persistent entity identity[2]. The entity instances associated with a persistence context are considered managed objects, with a well-defined lifecycle under the control of the persistence provider.

In the Jakarta EE environment, this programming model is augmented with *container-managed entity managers* with *transaction-scoped persistence contexts*, as defined below in Section 3.7.1 and later in Section 7.7.2. In this augmented programming model, the application holds a reference provided by the Jakarta EE container to an instance of `EntityManager` whose underlying persistence context is determined by transaction affinity.[3]

For example:

```
@Stateless
public class OrderEntryBean implements OrderEntry {
    @PersistenceContext
    EntityManager em;

    public void enterOrder(int custID, Order newOrder) {
        Customer cust = em.find(Customer.class, custID);
        cust.getOrders().add(newOrder);
        newOrder.setCustomer(cust);
        em.persist(newOrder);
    }
}
```

More information about container-managed entity managers and about transaction-scoped and extended persistence contexts may be found in Section 3.7 and Chapter 7.

Exceptions thrown by methods of an `EntityManager` are not usually considered recoverable. Any runtime exception thrown by a method of `EntityManager` causes the current transaction to be marked for rollback if the persistence context is joined to that transaction, unless the type of the exception is explicitly excepted below in Section 3.16. A persistence provider is not required to allow continued use of an `EntityManager` after one of its methods throws an unrecoverable exception type.

> When a method of `EntityManager` throws an exception, it is highly likely that the persistence context is left in an inconsistent state or in a state which is no longer consistent with the database. It is strongly recommended that the application immediately discard the `EntityManager`.

3.6. Entity Instance Lifecycle

Any entity instance can be characterized as being in one of the following *lifecycle states* with respect to a given persistence context:

- A **new** entity instance has no persistent identity and is not yet associated with any persistence context.
- A **managed** entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- A **detached** entity instance is an instance with a persistent identity that is not (or no longer) associated with a persistence context.
- A **removed** entity instance is an instance with a persistent identity, and associated with a persistence context, that is scheduled for removal from the database upon transaction commit.

Certain operations of `EntityManager` modify the lifecycle state of an entity instance. The following subsections describe the effect of such lifecycle operations.

3.6.1. Lifecycle Operations

The fundamental operations of an `EntityManager` are:

- `persist()` to make a new entity persistent, as defined in Section 3.6.3,
- `remove()`, to remove a persistent entity from the database, as defined in Section 3.6.4,
- `flush()`, to force synchronization of pending modifications with the database, as defined in Section 3.6.6,
- `refresh()`, to reload an entity from the database, as defined in Section 3.6.7,
- `detach()`, to disassociate an entity from its persistence context, as defined in Section 3.6.8,
- `merge()`, to copy the state of a detached instance into the persistence context, and
- `lock()` to obtain a lock an entity instance or to upgrade the lock level currently held, as defined in Section 3.9.

The methods `persist()`, `remove()`, `refresh()`, `detach()`, and `merge()` are called *lifecycle operations*.

The `cascade` annotation element of a relationship modeling annotation, as defined in Section 2.11, specifies which lifecycle operations propagate to associated entities. Each lifecycle operation is identified by the corresponding member of the enumeration `CascadeType`.

```
public enum CascadeType {ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH}
```

The value `cascade=ALL` is equivalent to `cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}`.

> The cascade functionality is most typically used to model parent-child relationships. A closely related feature is orphan removal, which is defined below in Section 3.6.5.

Information about how lifecycle operations interact with transactions may be found below in Section 3.7.2.

3.6.2. Entity Instance Creation

An instance of an entity class may be created using the `new` operator. A newly instantiated instance is not considered persistent—that is, it is not considered to represent a record in the database—and the new instance is not associated with a persistence context. A new instance is not considered to have a well-defined persistent identity until it is made persistent.

3.6.3. Persisting an Entity Instance

A new instance becomes persistent when it is passed as an argument to the `persist()` method of `EntityManager` or when the persist operation cascades from another instance to the new instance.

Let `e` be an instance of an entity class. If `e` is passed as an argument to the `persist()` method of `EntityManager`, or if the persist operation is applied to `e` via cascade:

- if `e` is a new entity, it becomes managed and a new record represented by `e` will be inserted into the database at or before transaction commit as a result of the flush operation specified below in Section 3.6.6;
- if `e` is a managed entity already associated with the persistence context, `e` itself is ignored by the persist operation;
- if `e` is a removed entity, it becomes managed; or
- if `e` is a detached entity, either an `EntityExistsException` is thrown immediately, or an `EntityExistsException` or some other subtype of `PersistenceException` is thrown when the persistence context is flushed or when the transaction commits.
- If `e` is new, managed, or removed, the persist operation is also applied to every entity `t` referenced by a relationship owned by `e` and annotated with the `cascade` element value `cascade=PERSIST` or `cascade=ALL` or mapped as such by the equivalent XML descriptor element.

Ordinarily, the persist operation assigns a persistent identity to the entity. If the primary key of the entity is a generated value, as defined by Section 11.2.23, the primary key is generated as part of the persist operation, and the generated value is made available to the application by setting the primary key field or property of the entity. As a special exception to this rule, if the primary key generation strategy of an entity is `IDENTITY`, the persistence provider is permitted to delay assignment of the primary key until pending modifications are synchronized with the database, as defined by Section 3.6.6. Indeed, when `persist()` is called with no active transaction, the persistence provider is *required* to delay generation and assignment of `IDENTITY` primary keys.

3.6.4. Removal

A managed entity instance becomes removed when it is passed as an argument to the `remove()` method of `EntityManager` or when the remove operation cascades from another instance to the managed instance.

Let `e` be an instance of an entity class. If `e` is passed as an argument to the `remove()` method of `EntityManager`, or if the remove operation is applied to `e` via cascade:

- if `e` is a new entity, `e` itself is ignored by the remove operation;
- if `e` is a managed entity, it is marked as removed and the record represented by `e` is scheduled for deletion from the database at or before transaction commit as a result of the flush operation specified below in Section 3.6.6;
- if `e` is a removed entity, `e` itself is ignored by the remove operation; or
- if `e` is a detached entity, an `IllegalArgumentException` is thrown or a failure occurs later when the flush operation occurs.
- If `e` is new, managed, or removed, the remove operation is also applied to every entity `t` referenced by a relationship

owned by `e` and annotated with the `cascade` element value `cascade=REMOVE` or `cascade=ALL` or mapped as such by the equivalent XML descriptor element.

The `remove()` method does not change the state held by fields of the entity instance itself.

3.6.5. Orphan Removal

The `orphanRemoval` member of the `@OneToOne` and `@OneToMany` relationship modelling annotations controls the enablement of *orphan removal*. When a relationship is mapped `orphanRemoval=true`, the remove operation is automatically applied to a child entity when the association between the child entity and its parent is destroyed.

- If a managed entity instance is removed from the relationship by setting the relationship attribute to `null` or by removing the entity from the relationship collection, the remove operation is applied to the orphaned entity. The remove operation occurs when the persistence context is flushed, as defined below in Section 3.6.6. If an entity being orphaned is a detached, new, or removed entity, it is ignored by orphan removal.
- If the remove operation is applied to a managed entity, the remove operation is cascaded to the relationship target in accordance with the rules of Section 3.6.4, and it is not therefore necessary to explicitly specify `cascade=REMOVE` for a relationship declared with `orphanRemoval=true`.[4]

Portable applications must not depend on a specific timing of the removal operation and must not reassign an entity that has been orphaned to another relationship, nor otherwise attempt to make it persistent, reversing the effect of orphan removal.

3.6.6. Synchronization to the Database

Usually, a persistence context is synchronized to the database as described below. However, as an exception, a persistence context of type `SynchronizationType.UNSYNCHRONIZED`, or an application-managed persistence context that has been created outside the scope of the current transaction, is only synchronized to the database if it has been explicitly joined to the current transaction by calling the `joinTransaction()` method of `EntityManager`, as specified below in Section 3.7.3.

Assuming that the persistence context is joined to the current transaction, and that its flush mode was not explicitly set to `EXPLICIT`, the state of its managed entities is synchronized to the database at transaction commit. This synchronization involves writing to the database any pending modifications to the state of persistent entities and their relationships.

A modification to the state of an entity might include an assignment of a new value to a persistent property or field of the entity, or even modification of a mutable value of a persistent property or field[5].

Synchronization to the database does not involve refreshing the state of managed entities with changes made to the database[6].

Bidirectional relationships between managed entities are persisted based on references held by the owning side of the relationship. It is the developer's responsibility to keep the in-memory references held on the owning side and those held on the inverse side consistent with each other when they change. In the case of unidirectional one-to-one and one-to-many relationships, it is the developer's responsibility to ensure that the semantics of the relationships are adhered to.[7]

> It is particularly important to ensure that changes made to the inverse side of a relationship are reflected by corresponding changes on the owning side. The provider is not required to synchronize changes made only at the inverse end of an association. Failure to modify the owning side of a relationship typically results in the change being lost.

When a transaction is active and the persistence context is joined to the transaction, the persistence provider runtime is permitted to flush pending modifications to the database. The application can influence the timing of synchronization via the `flush()` method of `EntityManager`, by setting the `FlushModeType`, or by specifying a `QueryFlushMode`.

- When the `flush()` method is called, the provider must synchronize every pending modification made to every entity currently associated with the persistence context.
- The `setFlushMode()` methods of the `EntityManager`, `Statement`, `Query`, `TypedQuery`, and `StoredProcedureQuery` interfaces can be used to control synchronization initiated automatically by the provider.
  - The effect of `FlushModeType.AUTO` is defined in Section 3.15.3.
  - If `FlushModeType.COMMIT` is specified, synchronization occurs at transaction commit; the persistence provider is permitted, but not required, to flush at other times.
  - If `FlushMode.EXPLICIT` is specified, the persistence provider must never automatically flush modifications to the database.
- The effect of the `FlushModeType` of the entity manager may be overridden during execution of a given query by explicitly specifying a `QueryFlushMode` via the `setQueryFlushMode()` method of `Statement`, `Query`, `TypedQuery`, or `StoredProcedureQuery`, as specified below in Section 3.15.3.

If there is no transaction active or if the persistence context has not been joined to the current transaction, the persistence provider must not flush modifications to the database.

Let `e` be an instance of an entity class associated with a persistence context when synchronization occurs:

- If `e` is a managed entity:
  - Any pending modifications to the values held by its persistent attributes are synchronized to the database by updating the record represented by `e`.
  - The persist operation is applied to every entity `t` referenced by a relationship owned by `e` and annotated with the `cascade` element value `cascade=PERSIST` or `cascade=ALL` or mapped as such by the equivalent XML descriptor element.
  - Every entity `t` referenced by a relationship owned by `e` which is *not* annotated with the `cascade` element value `cascade=PERSIST` or `cascade=ALL` or mapped as such by the equivalent XML descriptor element is checked:
    - if `t` is new or removed, an `IllegalStateException` is thrown and the transaction is marked for rollback, or the transaction commit fails; or
    - if `t` is detached, the semantics depend upon the ownership of the relationship: if `e` owns the relationship, any changes to the relationship are synchronized with the database; otherwise, if `t` owns the relationship, the behavior is undefined.
- If `e` is a removed entity, the record represented by `e` is removed from the database; no `cascade` options are relevant.

3.6.7. Refreshing an Entity Instance

The state of a managed entity instance is refreshed from the database when it is passed as an argument to the `refresh()` method of `EntityManager` or when the refresh operation cascades from another instance to the managed instance.

Let `e` be an instance of an entity class. If `e` is passed as the first argument to the `refresh()` method of `EntityManager`, or if the refresh operation is applied to `e` via cascade:

- if `e` is a managed entity, the state of `e` is refreshed from the database by reading the record represented by `e` and modifying the state of `e` to match, overwriting any pending modifications made to the entity; or
- if `e` is a new, detached, or removed entity, the `IllegalArgumentException` is thrown.
- If `e` is managed, the refresh operation is also applied to every entity `t` referenced by a relationship owned by `e` and annotated with the `cascade` element value `cascade=REFRESH` or `cascade=ALL` or mapped as such by the equivalent XML

descriptor element.

3.6.8. Evicting an Entity Instance from the Persistence Context

A managed entity instance is removed from the persistence context when it is passed as an argument to the `detach()` method of `EntityManager` or when the detach operation cascades from another instance to the managed instance.

Let `e` be an instance of an entity class. If `e` is passed as an argument to the `detach()` method of `EntityManager`, or if the detach operation is applied to `e` via cascade:

- if `e` is a managed entity, it is removed from the persistence context and becomes detached;
- if `e` is a new or detached entity, it is ignored by the detach operation;
- if `e` is a removed entity, it becomes detached, and the scheduled removal of the record represented by `e` is cancelled.
- If `e` is managed or removed, the detach operation is also applied to every entity `t` referenced by a relationship owned by `e` and annotated with the `cascade` element value `cascade=DETACH` or `cascade=ALL` or mapped as such by the equivalent XML descriptor element.

Portable applications should not pass removed entities that have been detached from the persistence context to methods of `EntityManager`.

Entities which referenced `e` before detachment continue to reference `e` after detachment.

> Pending modifications to the state of the entity, including removal of the entity, are never synchronized to the database after the entity is evicted from the persistence context. If such pending modifications exist, an application must call the `flush()` method before calling `detach()` if portable behavior is required. A portable application must not assume that pending modifications have not been written to the database if the `flush()` method was not called before `detach()`. More information about synchronization may be found above in Section 3.6.6.

3.6.9. Detached Entities

A managed entity instance associated with a persistence context becomes detached whenever:

- the persistence context is a container-managed transaction-scoped persistence context, as defined in Section 3.7.1, and the transaction associated with the persistence context commits, as specified in Section 3.7.4,
- the persistence context is a container-managed persistence context, and the transaction associated with the persistence context rolls back, as specified in Section 3.7.5,
- the entity is passed to the `detach()` method of the `EntityManager` associated with the persistence context,
- the `clear()` method is called on the `EntityManager` associated with the persistence context, or
- the `EntityManager` associated with the persistence context is closed.

A detached entity instance continues to live outside the persistence context in which it was persisted or retrieved. Its state does not remain synchronized with the database record it represents.

When an entity instance is passed to an operation like `persist()` or `merge()`, the persistence provider is permitted to use the value of the identifier and version fields or properties of an entity instance to determine whether the instance is new or detached.

3.6.9.1. Detached Entities and Lazy Loading

The *available* state of a detached entity instance is the state which was already loaded when the instance became detached. The application may access the available state of available detached entity instances after the persistence

context ends.

The available state of a detached entity or of an embeddable object belonging to a detached entity includes every persistent field or property:

- not explicitly mapped as `fetch=LAZY`,
- accessed by the application while the entity was managed, or
- fetched by means of an entity graph, as specified below in Section 3.12.

If the persistent field or property is an association, the available state of an associated instance may be safely accessed if and only if the associated instance is available. The available instances include every entity instance:

- retrieved using `find()`, `get()`, `findMultiple()`, or `getMultiple()`, as specified in Section 3.6.11,
- returned by a query or explicitly requested in a fetch join or entity graph,
- with a persistent attribute other than the primary key which was accessed by the application while the entity was managed, or
- reachable from an available instance by navigating associations mapped explicitly or by default as `fetch=EAGER`.

> A persistence provider might allow access to the primary key of an unavailable associated entity.

A `PersistenceException` is thrown if the application accesses an unavailable attribute, collection, or entity.

3.6.9.2. Merging Detached Entity State

The merge operation allows for the propagation of state from detached entities to managed instances associated with a persistence context. Given a detached instance, the merge operation returns a managed instance with the same persistent identity and state as the detached instance, but with a distinct Java object identity.

Let `e` be an instance of an entity class. If `e` is passed as an argument to the `merge()` method of `EntityManager`, or if the merge operation is applied to `e` via cascade, a merged object `e2` is produced:

- if `e` is a detached entity, the persistent state of `e` is copied to a pre-existing managed entity instance `e2` of the same identity or, if there is no such pre-existing managed entity, a new managed copy `e2` of `e` is created;
- if `e` is a new entity instance, a new managed entity instance `e2` is created and the state of `e` is `copied` to `e2`;
- if `e` is a removed entity instance, an `IllegalArgumentException` is thrown or a failure occurs later when the flush operation occurs; or
- if `e` is a managed entity, `e` itself is ignored by the merge operation and the merged object `e2` is taken to be identical to `e`.
- If `e` is new, managed, or detached, the merge operation is also applied to every entity `t` referenced by a relationship owned by `e` and annotated with the `cascade` element value `cascade=MERGE` or `cascade=ALL` or mapped as such by the equivalent XML descriptor element. For every such `t` referenced by `e`, `e2` is set to reference the object `t2` produced by applying the merge operation to `t`.
- For every entity `t` referenced by `e` via an association, navigation of the same association from `e2` yields a reference to a managed object `s` with the same persistent identity as `t`.

If `e` has a version, as defined in Section 2.5, the persistence provider must validate that `e` and `e2` have the same version, either during processing of the merge operation, or later when the flush operation occurs or when the transaction commits.

The persistence provider must ignore unfetched attributes of detached instances when merging. A persistence provider is permitted to report an exception if an entity instance with unfetched lazy attributes instantiated by a different persistence provider runtime is passed to `merge()`.

When this sort of interoperability across different persistence provider runtimes from different vendors is required, the application must not use lazy loading.

3.6.9.3. Detached Entities and Serialization

A persistence provider is required to support:

1. the serialization of detached entity instances with unfetched lazy attributes instantiated by the persistence provider runtime,
2. subsequent deserialization in a separate Java Virtual Machine with access to the same persistence provider runtime and the same entity classes, and
3. merging of the deserialized instances in a new persistence context implemented by the same persistence provider runtime in the second virtual machine.

3.6.10. Managed Entities

It is the responsibility of the persistence provider to ensure that at most one entity instance belonging to a given persistence context has a given persistent identity. The persistent identity of an entity class instance is represented as a pair comprising:

1. the class of the root entity of the entity class inheritance hierarchy to which the entity class belongs, together with
2. the value of the primary key of the entity instance, as defined above in Section 2.4.

Therefore, if the entity type `X` is a subclass of the root entity type `Y`, and `x` is an instance of `X` with primary key `k`, the persistent identity of `x` is *(`Y`,`k`)*.

It is the responsibility of the application to ensure that an instance of an entity class is associated with at most one persistence context. If an entity instance is made managed in more than one persistence context, the behavior is undefined and unportable.

The `contains()` method of `EntityManager` determines whether an entity instance is managed in the associated persistence context. The `contains()` method returns `true` if and only if:

- the argument is a managed entity instance associated with the given persistence context, and
- the argument entity is not marked as removed.

The effect of cascading persist, merge, remove, or detach is immediately visible to the `contains()` method, whereas the actual insertion, update, or deletion of the database record represented by the entity is deferred until the flush operation occurs.

3.6.11. Entity Instance Retrieval

The interface `EntityHandler`, found in Section B.1 declares operations common to `EntityManager` and `EntityAgent`, including:

- `find()`, `get()`, `findMultiple()`, and `getMultiple()`, which allow *retrieval* of an entity or entities—that is, given a persistent identity corresponding to a record which already exists in the database, or a list of persistent identities, these operations return an entity instance or instances representing the corresponding record or records.
- the methods `createQuery()`, `createNamedQuery()`, `createNativeQuery()`, and `createStoredProcedureQuery()`, which allow creation and subsequent execution of a query which returns entity instances representing records which exist in the database.

The status of an entity instance returned by an operation declared by `EntityHandler`, `EntityManager`, or `EntityAgent` depends on whether the operation is called on an entity manager or on an entity agent. Similarly, the status of an entity instance returned by an operation declared by `Query`, `TypedQuery`, or `StoredProcedureQuery` depends on whether the query object was obtained from an entity manager or from an entity agent.

- For an `EntityAgent`, there is no associated persistence context, and so every entity instance returned by any operation of the agent itself, or of any query object obtained from the agent, is in the detached state. Operations on an entity agent never return the same entity instance twice.
- For an `EntityManager`, every entity instance returned by any operation of the manager itself or of any query object obtained from the manager, with the exception noted in Section 4.9.2, is in the managed state and belongs to the persistence context associated with the entity manager. The instance remains in the managed state until the entity manager is closed and its persistence context is destroyed unless the instance is explicitly detached via a call to `detach()` or `clear()`, as specified above in Section 3.6.9. An entity manager is required to ensure that if an entity instance representing a given record already exists in its associated persistence context, then exactly that instance is returned by every operation that returns an entity representing the record.

> When a container-managed `EntityManager` with a transaction-scoped persistence context is called with no active transaction, the call appears to the client to return detached entity instances, as described below in Section 3.7.2 and Section 3.15.2. However, strictly speaking, this is no more than a consequence of the extremely short lifetime of the persistence context in this scenario. From the point of view of the persistence provider, the operations of `EntityManager` return managed instances exactly as described here.

A retrieval operation like `find()` or `get()` accepts an entity class and a primary key value. Retrieval is by nature polymorphic, and the given entity class might be a subclass in an entity class inheritance hierarchy. Therefore, given an entity type `X` and primary key `k`, any retrieval operation must:

1. infer the persistent identity *(`Y`,`k`)*, where `Y` is the root entity class of the entity class inheritance hierarchy to which `X` belongs,
2. determine if there is a database record corresponding to that persistent identity, and
3. determine if the database record is represented by an instance of an entity class assignable to `X`.

If a record exists with persistent identity *(`Y`,`k`)* but it is not represented by an entity class assignable to `X`, a `PersistenceException` must be thrown.

Otherwise, if a record exists with persistent identity *(`Y`,`k`)* and it is represented by an entity class `Z` assignable to `X`, an instance of `Z` with primary key `k` is returned to the caller.

3.6.12. Load State

An instance of an entity or embeddable class is considered *loaded* if every attribute of the entity or embeddable class mapped (explicitly or by default) as `FetchType.EAGER`, including relationships and element collections, has been loaded from the database or assigned by the application. A loaded entity or embeddable object might have an unloaded attribute if and only if the attribute is mapped as `FetchType.LAZY`.

The available state of an entity instance and its associated instances is described above in Section 3.6.9.1.

An attribute of an entity or embedded class might also be considered *loaded*.

- An embedded attribute, as defined in Section 2.7 is considered to be loaded if the embedded attribute was loaded from the database or assigned by the application, and, if the attribute references a non-null instance, the embeddable instance is loaded.

- A collection-valued attribute is considered to be loaded if the collection was loaded from the database or the value of the attribute was assigned by the application and, if the attribute references a non-null instance of the collection type, each element of the collection is considered to be loaded.
- A single-valued relationship attribute is considered to be loaded if the relationship attribute was loaded from the database or assigned by the application and, if the attribute references a non-null entity instance, the entity instance is considered to be loaded.
- A basic attribute is considered to be loaded if its state has been loaded from the database or assigned by the application.

The `isLoaded()` methods of the `PersistenceUtil` interface found in Section B.24 determine the load state of a given entity or attribute of a given entity, regardless of the persistence unit with which the entity is associated, returning `true` if given entity or attribute is considered loaded according to the rules given above, or `false` otherwise.

If the persistence unit to which the entity belongs is known, the `isLoaded()` methods of the interface `PersistenceUnitUtil` found in Section B.24 and described in Section 7.11 may be used instead.

Persistence provider contracts for determining the load state of an entity or entity attribute are defined in Section 9.9.1.

3.7. Persistence Context Lifetime and Transaction Synchronization

In the Jakarta EE environment, a *container-managed persistence context* is a persistence context whose lifecycle is managed by the Jakarta EE container. The container is responsible for:

- creating and destroying instances of `EntityManager`,
- injecting appropriate contextual references at application-declared injection points of type `EntityManager`, as specified in Section 7.2.1, and
- propagating persistence contexts according to the rules given in Section 7.7.4.

An *application-managed persistence context* is any persistence context whose lifecycle is *not* managed by a Jakarta EE container. Typically, this means that an application or third-party library or framework creates an `EntityManager` by calling `createEntityManager()` on an `EntityManagerFactory` which might or might not have been created by a Jakarta EE container. In the Java SE environment, every persistence context is considered an application-managed persistence context by this specification, even if its lifecycle is managed by some container-like framework.

More information about container-managed persistence contexts may be found later in Section 7.7. More information about application-managed persistence contexts may be found in Section 7.8.

3.7.1. Persistence Context Types

There are two kinds of container-manager persistence context, distinguished by distinct rules for their lifecycle and propagation:

- a *transaction-scoped persistence context* has a lifetime which extends to the moment of completion of the current transaction and propagates with the transaction, as specified in Section 7.7.2, while
- an *extended persistence context* has a lifetime which is tied to an instance of a stateful session bean, as specified in Section 7.7.3, and might span multiple transactions and even multiple non-transactional invocations of the `EntityManager`; the extended persistence context is destroyed by the container when its stateful session bean is destroyed.

The persistence context type is determined when an instance of `EntityManager` is created by the Jakarta EE container. By default, a persistence context is created as a transaction-scoped persistence context. When a persistence context is created for an injection point annotated `@PersistenceContext`, the `type` member may explicitly specify a

PersistenceContextType.

```
/**
 * Specifies the lifecycle of a container-managed persistence context
 * in a Jakarta EE container environment.
 * <ul>
 * <li>A {@linkplain #TRANSACTION transaction-scoped} persistence
 *     context has a lifecycle bound to the current JTA transaction.
 *     The container is responsible for maintaining transactional
 *     affinity when a reference to an {@link EntityManager} is
 *     injected.
 * <li>An {@linkplain #EXTENDED extended} persistence context has a
 *     lifecycle bound to an instance of a stateful session bean and
 *     may be inherited by other stateful session beans directly or
 *     indirectly instantiated by the first bean.
 * </ul>
 *
 * <p>By default, a container-managed persistence context is
 * transaction-scoped. The persistence context type for a
 * container-managed persistence context is specified by
 * {@link PersistenceContext#type}.
 *
 * @see PersistenceContext#type
 *
 * @since 1.0
 */
public enum PersistenceContextType {

    /**
     * A new persistence context begins when the container-managed entity
     * manager is invoked in the scope of an active JTA transaction, and
     * there is no current persistence context already associated with
     * the transaction. The persistence context is created and associated
     * with the transaction. This affinity of the persistence context with
     * the JTA transaction is independent of the
     * {@link SynchronizationType synchronization type} of the persistence
     * context and of whether the persistence context has been joined to
     * the transaction.
     * <p>
     * The persistence context propagates with the current JTA transaction
     * to other injected transaction-scoped entity manager references with
     * the same persistence unit and synchronization type.
     * <p>
     * The persistence context ends when the associated transaction commits
     * or rolls back, and all entities belonging to the persistence context
     * become detached.
     * <p>
     * If the entity manager is invoked outside the scope of a transaction,
     * all entities loaded from the database immediately become detached at
     * the end of the method call.
     */
    TRANSACTION,

    /**
     * The persistence context exists from the point at which the stateful
     * session bean declaring a dependency on the persistence context is
     * created. The container destroys the persistence context after the
     * {@code @Remove} method of the stateful session bean completes or
     * when the stateful session bean instance is otherwise destroyed.
     *
     * <p>If a stateful session bean instantiates a stateful session bean
     * (executing in the same EJB container instance) which also declares a
     * dependency on an extended persistence context with the same unit and
     * synchronization type, the extended persistence context of the first
     * stateful session bean is inherited by the second stateful session
     * bean. The container does not destroy an extended persistence context
```

```
     * until every stateful session bean inheriting the persistence context
     * has been removed or otherwise destroyed.
     */
    EXTENDED
}
```

Extended persistence contexts and application-managed persistence contexts behave similarly in almost every respect.

3.7.2. Entity Manager Operations and Transactions

The lifecycle operations defined above in Section 3.6.1 may not be invoked on an entity manager with a transaction-scoped persistence context when no transaction is active. If one of these methods is called on an entity manager with a transaction-scoped persistence context when there is no active transaction context, the method must throw `jakarta.persistence.TransactionRequiredException`. As an exception to this rule, `detach()` may be called on an entity manager with a transaction-scoped persistence context when there is no active transaction context, but it is treated as a noop.

On the other hand, the same lifecycle operations may be called on an entity manager with an extended or application-managed persistence context even when no transaction is active, subject to the restriction on lock modes given below. But the effect of such an operation is never synchronized with the database, as defined in Section 3.6.6, until the extended persistence context is joined to an active transaction.

The entity retrieval methods defined above in Section 3.6.11 may be invoked when there is no active transaction, subject to the restriction on lock modes given below:

- if the entity manager has a transaction-scoped persistence context, the entity or entities returned by a retrieval method are detached;
- otherwise, if the entity manager has an extended persistence context, the returned entity or entities are managed.

Similar rules applying to entity instances returned by execution of a query object obtained from an `EntityManager` are specified below in Section 3.15.2.

The `getReference()` method may be called without an active transaction.

The `contains()` method may be called without an active transaction. If the entity manager has a transaction-scoped persistence context and there is no active transaction, it returns `false`.

The `clear()` method may be called without an active transaction. If the entity manager has a transaction-scoped persistence context and there is no active transaction, it is treated as a noop.

The `flush()` method must not be called without an active transaction. If `flush()` is called when there is no active transaction, the provider must throw `jakarta.persistence.TransactionRequiredException`.

The `lock()` method must not be called without an active transaction. If `lock()` is called when there is no active transaction, the provider must throw `jakarta.persistence.TransactionRequiredException`.

The method `joinTransaction()` associates the persistence context underlying the `EntityManager` with the currently active container transaction. If `joinTransaction()` is called when there is no active transaction, the provider or container must throw `jakarta.persistence.TransactionRequiredException`.

If a method of `EntityManager` accepts a lock mode and the method is called when no transaction is active, then the specified lock mode must be `LockModeType.NONE`. If there is no active transaction, or if the entity manager has not been joined to the transaction, and any lock mode other than `LockModeType.NONE` is passed to the method, the method must throw `jakarta.persistence.TransactionRequiredException`.

An `EntityManager` with an extended persistence context maintains its persistence context and references to entity objects belonging to its persistence context across transaction commits. Those objects remain managed by the `EntityManager` even when there is no transaction active, and can be updated as managed objects between transactions.[8] Navigation from a managed object associated with an extended persistence context results in one or more other managed objects regardless of whether a transaction is active.

3.7.3. Synchronization with the Current Transaction

When a persistence unit is configured to use container-managed JTA transactions, as defined in Section 8.2.1.2, any entity manager associated with that persistence unit is considered a *JTA entity manager*, as specified in Section 7.5.

Every persistence context associated with a JTA entity manager has a well-defined *synchronization type*.

- A *synchronized* persistence context is automatically joined to the current container transaction, if any, when the `EntityManager` is created or when any method of the `EntityManager` is called.
- An *unsynchronized* persistence context is not joined to a transaction until `EntityManager.joinTransaction()` is called explicitly by the application.

As specified above in Section 3.6.6, modifications held in a persistence context are not synchronized with the database unless the persistence context is joined to the current transaction.

The synchronization type of a persistence context is determined when an instance of `EntityManager` is created by the application or Jakarta EE container. By default, a persistence context is created as a synchronized persistence context.

- The application may create an unsynchronized persistence context by passing `SynchronizationType.UNSYNCHRONIZED` to `createEntityManager()`.
- When a container-managed persistence context is created for an injection point annotated `@PersistenceContext`, the `synchronization` member may explicitly specify a `SynchronizationType`.

```
/**
 * Specifies whether the persistence context associated with an
 * {@link EntityManager} is always automatically synchronized with
 * the current transaction.
 * <ul>
 * <li>{@link #SYNCHRONIZED} indicates that the persistence context
 *     is automatically joined to the current transaction, and that
 *     modifications are always automatically synchronized with the
 *     database according to the current {@linkplain FlushModeType
 *     flush mode}.
 * <li>{@link #UNSYNCHRONIZED} indicates that the persistence context
 *     is not automatically joined to the current transaction, and
 *     that its modifications are not synchronized with the database
 *     until it is explicitly joined to the current transaction by
 *     calling {@link EntityManager#joinTransaction()}.
 * </ul>
 *
 * <p>By default, a container-managed persistence context is of type
 * {@code SYNCHRONIZED} and is automatically joined to the current
 * transaction. A container-managed persistence context of type
 * {@code UNSYNCHRONIZED} is not enlisted in the current transaction
 * until {@code joinTransaction()} is called. The synchronization
 * type for a container-managed persistence context is specified by
 * {@link PersistenceContext#synchronization}.
 *
 * <p>By default, an application-managed persistence context
 * associated with a JTA entity manager and created within the
 * scope of an active JTA transaction is automatically joined to
 * that transaction. An application-managed JTA persistence context
 * created outside the scope of an active JTA transaction, or any
 * application-managed persistence context created with type
```

```
 * {@code UNSYNCHRONIZED}, is not enlisted with a transaction until
 * {@code joinTransaction()} is called. The synchronization type for
 * an application-managed JTA persistence context is specified via
 * {@link EntityManagerFactory#createEntityManager(EntityManager.CreationOption...)}.
 *
 * <p>An application-managed persistence context associated with a
 * {@linkplain PersistenceUnitTransactionType#RESOURCE_LOCAL
 * resource-local} entity manager is always automatically joined to
 * any {@linkplain EntityTransaction resource-local transaction}
 * begun for that entity manager.
 *
 * @see PersistenceContext#synchronization
 * @see EntityManagerFactory#createEntityManager(EntityManager.CreationOption...)
 * @see FlushModeType
 * @see PersistenceContextType
 *
 * @since 2.1
 */
public enum SynchronizationType implements EntityManager.CreationOption {

    /**
     * The persistence context is automatically joined to the current
     * transaction, and modifications are automatically synchronized
     * with the database according to the current
     * {@linkplain FlushModeType flush mode}.
     */
    SYNCHRONIZED,

    /**
     * The persistence context must be explicitly joined to the current
     * transaction by calling {@link EntityManager#joinTransaction()}.
     * When the persistence context is not joined to a transaction,
     * modifications cannot be synchronized with the database, regardless
     * of the current flush mode. The persistence provider must not flush
     * pending modifications to the database until the persistence context
     * is joined to a transaction.
     */
    UNSYNCHRONIZED,
}
```

An application-managed persistence context associated with a resource-local entity manager is always automatically joined to any resource-local transaction begun for that entity manager.

Persistence context synchronization type is further described in Section 7.7.1.

3.7.4. Transaction Commit

Transaction commit triggers synchronization of pending modifications held in any persistence context joined to the transaction, as specified in Section 3.6.6.

The managed entities of a transaction-scoped persistence context joined to an active transaction become detached when the transaction commits; the managed entities of an extended persistence context remain managed across transaction commits.

3.7.5. Transaction Rollback

Transaction rollback typically results in a loss of consistency between records in the database and any persistence context joined to the transaction. The persistence provider is not required to resynchronize the state of the persistence context with state held in the database after a transaction rolls back; nor is it required to refresh the entity instances belonging to the persistence context; nor is it required to revert the values of generated primary keys or of updated

versions.

> It is therefore strongly recommended that the application discard references to entities modified or made persistent in a transaction when the transaction rolls back.

When a container-managed persistence context is joined to a transaction and the transaction rolls back:

- the container closes the `EntityManager`, destroying the persistence context;
- every managed or removed entity instance representing a record which existed at the beginning of the transaction becomes detached;
- every managed or removed entity instance representing a record which did not exist at the beginning of the transaction is left in a state which is undefined by this specification.

The state of entities belonging to the destroyed persistence context is left unmodified: that is, the attributes of each such entity continue to hold the values they held immediately before the transaction rolled back.

Transaction rollback has no effect on any persistence context not joined to the transaction.

3.8. EntityAgent interface

The `EntityAgent` interface may be found in Section B.3.

The fundamental operations of an `EntityAgent` are: `insert()`, `update()`, `delete()`, `upsert()`, and `refresh()`. These operations perform immediate access to the database.

Every method of `EntityAgent` may be called without an active transaction, subject to the restriction on lock modes given below.

> Write operations performed outside of a transaction are permitted but not encouraged.

If a method of `EntityAgent` accepts a lock mode and the method is called when no transaction is active, then the specified lock mode must be `LockModeType.NONE`. If there is no active transaction and any lock mode other than `LockModeType.NONE` is passed to the method, the method must throw `jakarta.persistence.TransactionRequiredException`.

Entity instances returned by any method of `EntityAgent`, or by execution of any query object obtained by calling `createQuery()`, `createNamedQuery()`, `createNativeQuery()`, or `createStoredProcedureQuery()` on an `EntityAgent`, are always detached.

Exceptions thrown by methods of an `EntityAgent` are considered recoverable, and do not usually cause the active transaction to be marked for rollback.

3.9. Locking and Concurrency

This specification assumes the use of optimistic concurrency control. It assumes that the databases to which persistence units are mapped will be accessed by the implementation using read-committed isolation (or a vendor equivalent in which long-term read locks are not held), and that writes to the database will typically occur only when the `flush` method has been invoked—whether explicitly by the application, or by the persistence provider runtime in accordance with the flush mode setting.

> If a transaction is active and the persistence context is joined to the transaction, a compliant implementation of this specification is permitted to write to the database immediately (i.e., whenever a managed entity is updated, created, and/or removed), however, the configuration of an implementation

> to require such non-deferred database writes is outside the scope of this specification.[9]

In addition, both pessimistic and optimistic locking are supported for selected entities by means of specified lock modes. Optimistic locking is described in Section 3.9.1 and Section 3.9.2; pessimistic locking in Section 3.9.3. Section 3.9.4 describes the setting of optimistic and pessimistic lock modes. The configuration of the setting of optimistic lock modes is described in Section 3.9.4.1, and the configuration of the setting of pessimistic lock modes is described in Section 3.9.4.2.

3.9.1. Optimistic Locking

Optimistic locking is a system of concurrency control where each revision of an item of data is assigned a version number or timestamp. When the data is read and then updated within a given unit of work, the version or timestamp is:

1. read from the database when the data itself is read, and
2. verified and then updated in the database when the data is updated.

Similarly, when the data is read and then deleted within a given unit of work, the version or timestamp is:

1. read from the database when the data itself is read, and
2. verified when the data is deleted.

An *optimistic locking conflict* occurs when verification fails, that is, if the version or timestamp held in the database changes between reading the data (step 1), and attempting to update or delete the data (step 2).

Thus, the unit of work is prevented from updating the data and creating a new revision, or from deleting the data, unless the revision it previously obtained is still the current revision. Optimistic lock verification ensures that an update of a given item is successful only when no intervening transaction has already updated the item, preventing the loss of updates made by such intervening transactions.

The persistence provider is required to perform optimistic locking automatically for every entity with a version, as defined in Section 2.5. A portable application which wishes to take advantage of automatic optimistic locking must specify a version field or property for each optimistically-locked entity using the `@Version` annotation defined in Section 11.2.59 or equivalent XML element.

When an optimistic locking conflict is detected, the persistence provider must:

- throw an `OptimisticLockException` and
- mark the current transaction for rollback.

A persistence provider might offer alternative implementations of optimistic locking, which do not depend on the entity having a version, but such functionality is not portable between providers.[10]

> Applications are strongly encouraged to enable optimistic locking for every entity which may be concurrently accessed or which may be merged from a detached state. Failure to make use of optimistic locking often leads to inconsistent entity state, lost updates, and other anomalies. If an entity does not have a version, the application itself must bear the burden of maintaining data consistency during optimistic units of work.

For the purposes of versioning and optimistic locking, the state of a given entity is considered to include:

- every persistent field or property which is not a relationship to another entity and is not annotated `ExcludedFromVersioning`, and

- every relationship owned by the entity, as defined by Section 2.11, which is not annotated `ExcludedFromVersioning` on the owning side. [11]

Unowned relationships are not considered part of the state of the entity.

3.9.2. Entity Versions and Optimistic Locking

The entity version must be updated by the persistence provider each time the state of an entity instance is written to the database. [12] Furthermore, if the current persistence context contains a revision of the entity instance when the instance is written to the database, the persistence provider must verify that the revision held in the persistence context is identical to the revision held in the database by comparing the versions held in memory and in the database. [13] If the versions do not match, the persistence provider must throw an `OptimisticLockException`.

The persistence provider must examine the version field or property of a detached entity instance when it is merged, as defined in Section 3.6.9.2, and throw an `OptimisticLockException` if the instance being merged holds a stale revision of the state of the entity—that is, if the entity was updated since the entity instance became detached. The timing of this version check is provider-dependent:

- the version check might occur synchronously with the call to `merge()`, or
- a provider might choose to delay the version check until a flush operation occurs, as defined in Section 3.6.6, or until the transaction commits.

If an update or merge operation involves entities with versions, and entities without versions, the persistence provider runtime is only required to perform optimistic lock verification for those entities which do have a version, and the consistency of the whole object graph is not guaranteed. The absence a version for some entity involved in the update or merge operation does not impede completion of the operation.

3.9.3. Pessimistic Locking

While optimistic locking is typically appropriate in dealing with moderate contention among concurrent transactions, in some applications it may be useful to immediately obtain long-term database locks for selected entities because of the often late failure of optimistic transactions. Such immediately obtained long-term database locks are referred to here as “pessimistic” locks.[14]

Pessimistic locking guarantees that once a transaction has obtained a pessimistic lock on an entity instance:

- no other transaction (whether a transaction of an application using the Jakarta Persistence API or any other transaction using the underlying resource) may successfully modify or delete that instance until the transaction holding the lock has ended.
- if the pessimistic lock is an exclusive lock[15], that same transaction may modify or delete that entity instance.

The `PessimisticLockScope` enum found in Section B.7 controls how a pessimistic lock applied to an entity affects associated collections and relationships.

`NORMAL`

The persistence provider must lock the row or rows of the primary and secondary tables mapped by the entity. In the case of a joined inheritance hierarchy, this includes all rows in tables mapped by the entity superclasses of the entity. This is the default behavior of pessimistic locking.

`EXTENDED`

The persistence provider must lock the row or rows of:

- the primary and secondary tables mapped by the entity, including all rows in tables mapped by the entity

superclasses of the entity in the case of a joined inheritance hierarchy,

- join tables mapped by associations owned by the entity, and
- collection tables mapped by collections belonging to the entity.

`FETCHED`

The persistence provider must lock the row or rows of:

- the primary and secondary tables mapped by the entity, including all rows in tables mapped by the entity superclasses of the entity in the case of a joined inheritance hierarchy,
- join tables mapped by associations which were fetched as part of the operation which obtained the pessimistic lock, along with the primary and secondary tables mapped by the associated entities,
- collection tables mapped by collections belonging to the entity which were fetched as part of the operation which obtained the pessimistic lock.

The values of the property `jakarta.persistence.lock.scope` property are defined by the `PessimisticLockScope` enum.

This specification does not define the mechanisms a persistence provider uses to obtain database locks, and a portable application should not rely on how pessimistic locking is achieved on the database.[16] In particular, a persistence provider or the underlying database management system may lock more rows than the ones selected by the application.

Whenever a pessimistically locked entity containing a version attribute is updated on the database, the persistence provider must also update (increment) the entity's version column to enable correct interaction with applications using optimistic locking. See Section 3.9.2 and Section 3.9.4.

Pessimistic locking may be applied to entities that do not contain version attributes. However, in this case correct interaction with applications using optimistic locking cannot be ensured.

3.9.4. Lock Modes

Lock modes are intended to provide a facility that enables the effect of "repeatable read" semantics for the items read, whether "optimistically" (as described in Section 3.9.4.1) or "pessimistically" (as described in Section 3.9.4.2).

A lock mode may be explicitly specified as an argument to the `lock()` method of `EntityManager` or to any other method of `EntityManager`, `Query`, and `TypedQuery` which accepts a lock mode, or via the `NamedQuery` annotation.

Lock mode values are defined by the `LockModeType` enum which may be found in Section B.6. Six distinct lock modes are defined. [17] The lock mode type values `READ` and `WRITE` are synonyms for `OPTIMISTIC` and `OPTIMISTIC_FORCE_INCREMENT` respectively. The latter are to be preferred for new applications.

3.9.4.1. OPTIMISTIC, OPTIMISTIC_FORCE_INCREMENT

The lock modes `OPTIMISTIC` and `OPTIMISTIC_FORCE_INCREMENT` are used for optimistic locking. The lock mode type values `READ` and `WRITE` are synonymous with `OPTIMISTIC` and `OPTIMISTIC_FORCE_INCREMENT` respectively.

The semantics of requesting locks of type `LockModeType.OPTIMISTIC` and `LockModeType.OPTIMISTIC_FORCE_INCREMENT` are the following.

If transaction T1 calls `lock(entity, LockModeType.OPTIMISTIC)` on a versioned object, the entity manager must ensure that neither of the following phenomena can occur:

- P1 (Dirty read): Transaction T1 modifies a row. Another transaction T2 then reads that row and obtains the modified value, before T1 has committed or rolled back. Transaction T2 eventually commits successfully; it does not matter whether T1 commits or rolls back and whether it does so before or after T2 commits.

- P2 (Non-repeatable read): Transaction T1 reads a row. Another transaction T2 then modifies or deletes that row, before T1 has committed. Both transactions eventually commit successfully.

This will generally be achieved by the entity manager acquiring a lock on the underlying database row. While with optimistic concurrency concurrency, long-term database read locks are typically not obtained immediately, a compliant implementation is permitted to obtain an immediate lock (so long as it is retained until commit completes). If the lock is deferred until commit time, it must be retained until the commit completes. Any implementation that supports repeatable reads in a way that prevents the above phenomena is permissible.

The persistence implementation is not required to support calling `lock(entity, LockModeType.OPTIMISTIC)` on a non-versioned object. When it cannot support such a lock call, it must throw the `PersistenceException`. When supported, whether for versioned or non-versioned objects, `LockModeType.OPTIMISTIC` must always prevent the phenomena P1 and P2. Applications that call `lock(entity, LockModeType.OPTIMISTIC)` on non-versioned objects are not portable.

If transaction T1 calls `lock(entity, LockModeType.OPTIMISTIC_FORCE_INCREMENT)` on a versioned object, the entity manager must avoid the phenomena P1 and P2 (as with `LockModeType.OPTIMISTIC`) and must also force an update (increment) to the entity's version column. A forced version update may be performed immediately, or may be deferred until a flush or commit. If an entity is removed before a deferred version update was to have been applied, the forced version update is omitted.

The persistence implementation is not required to support calling `lock(entity, LockModeType.OPTIMISTIC_FORCE_INCREMENT)` on a non-versioned object. When it cannot support such a lock call, it must throw the `PersistenceException`. When supported, whether for versioned or non-versioned objects, `LockModeType.OPTIMISTIC_FORCE_INCREMENT` must always prevent the phenomena P1 and P2. For non-versioned objects, whether or not `LockModeType.OPTIMISTIC_FORCE_INCREMENT` has any additional behavior is vendor-specific. Applications that call `lock(entity, LockModeType.OPTIMISTIC_FORCE_INCREMENT)` on non-versioned objects are not portable.

For versioned objects, it is permissible for an implementation to use `LockModeType.OPTIMISTIC_FORCE_INCREMENT` where `LockModeType.OPTIMISTIC` was requested, but not vice versa.

If a versioned object is otherwise updated or removed, then the implementation must ensure that the requirements of `LockModeType.OPTIMISTIC_FORCE_INCREMENT` are met, even if no explicit call to `EntityManager.lock` was made.

For portability, an application should not depend on vendor-specific hints or configuration to ensure repeatable read for objects that are not updated or removed via any mechanism other than the use of version attributes and the EntityManager `lock` method. However, it should be noted that if an implementation has acquired up-front pessimistic locks on some database rows, then it is free to ignore `lock(entity, LockModeType.OPTIMISTIC)` calls on the entity objects representing those rows.

3.9.4.2. PESSIMISTIC_READ, PESSIMISTIC_WRITE, PESSIMISTIC_FORCE_INCREMENT

The lock modes `PESSIMISTIC_READ`, `PESSIMISTIC_WRITE`, and `PESSIMISTIC_FORCE_INCREMENT` are used to immediately obtain long-term database locks.[18]

The semantics of requesting locks of type `LockModeType.PESSIMISTIC_READ`, `LockModeType.PESSIMISTIC_WRITE`, and `LockModeType.PESSIMISTIC_FORCE_INCREMENT` are the following.

If transaction T1 calls `lock(entity, LockModeType.PESSIMISTIC_READ)` or `lock(entity, LockModeType.PESSIMISTIC_WRITE)` on an object, the entity manager must ensure that neither of the following phenomena can occur:

- P1 (Dirty read): Transaction T1 modifies a row. Another transaction T2 then reads that row and obtains the modified value, before T1 has committed or rolled back.
- P2 (Non-repeatable read): Transaction T1 reads a row. Another transaction T2 then modifies or deletes that row, before T1 has committed or rolled back.

Any such lock must be obtained immediately and retained until transaction T1 completes (commits or rolls back).

Avoidance of phenomena P1 and P2 is generally achieved by the entity manager acquiring a long-term lock on the underlying database row(s). Any implementation that supports pessimistic repeatable reads as described above is permissible.

> A lock with `LockModeType.PESSIMISTIC_WRITE` can be obtained on an entity instance to force serialization among transactions attempting to update the entity data. A lock with `LockModeType.PESSIMISTIC_READ` can be used to query data using repeatable-read semantics without the need to reread the data at the end of the transaction to obtain a lock, and without blocking other transactions reading the data. A lock with `LockModeType.PESSIMISTIC_WRITE` can be used when querying data and there is a high likelihood of deadlock or update failure among concurrent updating transactions.

The persistence implementation must support calling `lock(entity, LockModeType.PESSIMISTIC_READ)` and `lock(entity, LockModeType.PESSIMISTIC_WRITE)` on a non-versioned entity as well as on a versioned entity.

It is permissible for an implementation to use `LockModeType.PESSIMISTIC_WRITE` where `LockModeType.PESSIMISTIC_READ` was requested, but not vice versa.

When the lock cannot be obtained, and the database locking failure results in transaction-level rollback, the provider must throw the `PessimisticLockException` and ensure that the JTA transaction or `EntityTransaction` has been marked for rollback.

When the lock cannot be obtained, and the database locking failure results in only statement-level rollback, the provider must throw the `LockTimeoutException` (and must not mark the transaction for rollback).

When an application locks an entity with `LockModeType.PESSIMISTIC_READ` and later updates that entity, the lock must be converted to an exclusive lock when the entity is flushed to the database.[19] If the lock conversion fails, and the database locking failure results in transaction-level rollback, the provider must throw the `PessimisticLockException` and ensure that the JTA transaction or `EntityTransaction` has been marked for rollback. When the lock conversion fails, and the database locking failure results in only statement-level rollback, the provider must throw the `LockTimeoutException` (and must not mark the transaction for rollback).

When `lock(entity, LockModeType.PESSIMISTIC_READ)`, `lock(entity, LockModeType.PESSIMISTIC_WRITE)`, or `lock(entity, LockModeType.PESSIMISTIC_FORCE_INCREMENT)` is invoked on a versioned entity that is already in the persistence context, the provider must also perform optimistic version checks when obtaining the lock. An `OptimisticLockException` must be thrown if the version checks fail. Depending on the implementation strategy used by the provider, it is possible that this exception may not be thrown until flush is called or commit time, whichever occurs first.

If transaction T1 calls `lock(entity, LockModeType.PESSIMISTIC_FORCE_INCREMENT)` on a versioned object, the entity manager must avoid the phenomenon P1 and P2 (as with `LockModeType.PESSIMISTIC_READ` and `LockModeType.PESSIMISTIC_WRITE`) and must also force an update (increment) to the entity's version column.

The persistence implementation is not required to support calling `lock(entity, LockModeType.PESSIMISTIC_FORCE_INCREMENT)` on a non-versioned object. When it cannot support such a lock call, it must throw the `PersistenceException`. When supported, whether for versioned or non-versioned objects, `LockModeType.PESSIMISTIC_FORCE_INCREMENT` must always prevent the phenomena P1 and P2. For non-versioned objects, whether or not `LockModeType.PESSIMISTIC_FORCE_INCREMENT` has any additional behavior is vendor-specific. Applications that call `lock(entity, LockModeType.PESSIMISTIC_FORCE_INCREMENT)` on non-versioned objects are not portable.

For versioned objects, it is permissible for an implementation to use `LockModeType.PESSIMISTIC_FORCE_INCREMENT` where `LockModeType.PESSIMISTIC_READ` or `LockModeType.PESSIMISTIC_WRITE` was requested, but not vice versa.

If a versioned object locked with `LockModeType.PESSIMISTIC_READ` or `LockModeType.PESSIMISTIC_WRITE` is updated, then the

implementation must ensure that the requirements of `LockModeType.PESSIMISTIC_FORCE_INCREMENT` are met.

3.9.4.3. Lock Mode Properties and Uses

The following property is defined by this specification for use in pessimistic locking, as described in Section 3.9.3:

```
jakarta.persistence.lock.scope
```

This property may be used with the methods of the `EntityManager` interface that allow lock modes to be specified, the `Query` and `TypedQuery` `setLockMode` methods, and the `NamedQuery` annotation. When specified, this property must be observed. The provider is permitted to lock more (but not fewer) rows than requested.

The following hint is defined by this specification for use in pessimistic locking.

```
jakarta.persistence.lock.timeout // time in milliseconds
```

This hint may be used with the methods of the `EntityManager` interface that allow lock modes to be specified, the `Query.setLockMode` method and the `NamedQuery` annotation. It may also be passed as a property to the `Persistence.createEntityManagerFactory` method and used in the `properties` element of the `persistence.xml` file. See Section 3.3, Section 3.15.4, Section 8.2.1.12, Section 9.7, and Section 10.4.1. When used in the `createEntityManagerFactory` method, the `persistence.xml` file, and the `NamedQuery` annotation, the timeout hint serves as a default value which can be selectively overridden by use in the methods of the `EntityManager`, `Query`, and `TypedQuery` interfaces as specified above. When this hint is not specified, database timeout values are assumed to apply.

A timeout value of `0` is interpreted to mean “no wait” locking.

Portable applications should not rely on this hint. Depending on the database in use and the locking mechanisms used by the persistence provider, the hint may or may not be observed.

Vendors are permitted to support the use of additional, vendor-specific locking hints. Vendor-specific hints must not use the `jakarta.persistence` namespace. Vendor-specific hints must be ignored if they are not understood.

If the same property or hint is specified more than once, the following order of overriding applies, in order of decreasing precedence:

- argument to method of `EntityManager`, `Query`, or `TypedQuery` interface
- specification to `NamedQuery` (annotation or XML)
- argument to `createEntityManagerFactory` method
- specification in `persistence.xml`

3.9.5. OptimisticLockException

Provider implementations may defer writing to the database until the end of the transaction, when consistent with the lock mode and flush mode settings in effect. In this case, an optimistic lock check may not occur until commit time, and the `OptimisticLockException` may be thrown in the “before completion” phase of the commit. If the `OptimisticLockException` must be caught or handled by the application, the `flush` method should be used by the application to force the database writes to occur. This will allow the application to catch and handle optimistic lock exceptions.

The `OptimisticLockException` provides an API to return the object that caused the exception to be thrown. The object reference is not guaranteed to be present every time the exception is thrown but should be provided whenever the persistence provider can supply it. Applications cannot rely upon this object being available.

In some cases an `OptimisticLockException` will be thrown and wrapped by another exception, such as a `RemoteException`, when VM boundaries are crossed. Entities that may be referenced in wrapped exceptions should implement `Serializable` so that marshalling does not fail.

An `OptimisticLockException` throw by an `EntityManager` always causes the transaction to be marked for rollback. An `OptimisticLockException` throw by an `EntityAgent` is considered recoverable and does not cause the transaction to be marked for rollback.

Refreshing objects or reloading objects in a new transaction context and then retrying the transaction is a potential response to an `OptimisticLockException`.

3.10. Entity Listeners and Callback Methods

A method may be designated as a lifecycle callback method to receive notification of *entity lifecycle events*. A lifecycle callback method can be declared by:

- an entity class,
- a mapped superclass, or
- an entity listener class associated with an entity or mapped superclass.

An *entity listener class* is a class whose methods are invoked in response to lifecycle events on an entity. Any number of entity listener classes can be defined for an entity class or mapped superclass.

A callback may also be registered programmatically at runtime by calling `EntityManagerFactory.addListener()`, passing:

- an entity class, or any supertype of an entity class, including `java.lang.Object`,
- an annotation class object representing a type of entity lifecycle event, and
- an instance of `Consumer`.

Such a callback is treated as if the given `Consumer` were an instance of an entity listener class associated with the given entity type or with every entity type which is a subtype of the given type, and as if the `accept()` method were annotated with the given entity lifecycle callback annotation type.

Lifecycle callback methods and entity listener classes are defined by means of metadata annotations or the XML descriptor.

Default entity listeners—entity listener classes whose callback methods apply to all entities in the persistence unit—can be specified by means of the XML descriptor.

Additional entity listeners applying only to certain entities may be declared using annotations or XML.

- A list of entity listeners applying to an entity class may be specified by annotating the entity class or one of its mapped superclasses with the `EntityListeners` annotation. The listener classes must be explicitly listed in the order they are invoked.
- A list of entity listeners applying to every entity class in a given package or module may be specified by annotating the package or module descriptor with the `EntityListeners` annotation. The listener classes must be explicitly listed in the order they are invoked.
- An entity listener class annotated with the `EntityListener` annotation applies to an entity class if the entity class is assignable to one of its lifecycle callback methods.
- The XML descriptor may be used as an alternative to specify the invocation order of entity listeners or to override the order specified in metadata annotations.

Any subset or combination of annotations may be specified on an entity class, mapped superclass, or listener class. The same method may be used to process multiple callback events. A given entity class or mapped superclass must not have

more than one lifecycle callback method for a given lifecycle event, but this restriction does not apply to entity listener classes.

Multiple entity classes and mapped superclasses in an inheritance hierarchy may define listener classes and/or lifecycle callback methods directly on the class. Section 3.10.3 describes the rules that apply to method invocation order in this case.

3.10.1. Lifecycle Callback Methods

Entity lifecycle callback methods can be defined on an entity listener class and/or directly on an entity class or mapped superclass.

A lifecycle callback method must be either:

- annotated with annotations designating the callback events for which it is invoked, or
- mapped to a callback event type using the XML descriptor.

The same annotations (and XML elements) are used to declare:

- callback methods of an entity class or mapped superclass, and
- callback methods of an entity listener class.

The signatures of the callback methods differ between these two cases:

- a callback method defined by an entity class or mapped superclass must have the signature:

  ```
  void method()
  ```

  where `method` is a legal Java method name, but

- a callback method defined by an entity listener class has the signature:

  ```
  void method(E entity)
  ```

  where `method` is a legal Java method name, and `E` is any supertype of an entity class or mapped superclass to which the entity listener applies. At runtime, the argument to the entity listener callback method is the entity instance for which the callback method is being invoked.

Callback methods can have `public`, `private`, `protected`, or package level access, but must not be `static` or `final`.

The following annotations designate lifecycle event callback methods of the corresponding types.

Annotations for callbacks corresponding to managed entity instance lifecycle operations:

- `@PrePersist`
- `@PostPersist`
- `@PreRemove`
- `@PostRemove`
- `@PreMerge`

Annotations for callbacks corresponding to database operations:

- `@PreInsert`
- `@PostInsert`
- `@PreUpdate`

- `@PostUpdate`
- `@PreUpsert`
- `@PostUpsert`
- `@PreDelete`
- `@PostDelete`

Finally, this annotation corresponds to a callback that occurs after persistent state read from the database is written to an entity instance, managed or otherwise.

- `@PostLoad`

The following rules apply to lifecycle callback methods:

- Lifecycle callback methods may throw unchecked/runtime exceptions. A runtime exception thrown by a callback method that executes within a transaction causes that transaction to be marked for rollback if the persistence context is joined to the transaction.
- Lifecycle callbacks can invoke JNDI, JDBC, JMS, and enterprise beans.
- A lifecycle callback method may modify the non-relationship state of the entity on which it is invoked.
- In general, the lifecycle method of a portable application should not invoke `EntityManager` or query operations, access other entity instances, or modify relationships within the same persistence context[20].

An entity or entity listener may have multiple lifecycle callback methods:

- an entity class may have at most one callback method for a given type of lifecycle event, but
- an entity listener class may have multiple callback methods for a given type of lifecycle event, but at most one callback method for a given type of event and given parameter type.

3.10.2. Semantics of the Lifecycle Callback Methods for Entities

A `@PrePersist`, `@PreRemove`, or `@PreMerge` callback occurs before the respective persist, remove, or merge operation for an entity is executed.

- The `@PrePersist` callback for an entity being made persistent occurs before the entity is marked as managed and associated with the persistence context.
- The `@PreRemove` callback occurs before the entity being removed is marked as removed in the persistence context.
- The `@PreMerge` callback for an entity being merged occurs before state is copied from the entity being merged to the corresponding managed entity, when copying is necessary.
- When the `merge()` operation results in the creation of a new managed instance, the `@PrePersist` callback for the managed instance occurs after the merged entity state has been copied to it.

A `@PrePersist`, `@PreRemove`, or `@PreMerge` method is always invoked synchronously during execution of the `persist()`, `remove()`, or `merge()` operation of the `EntityManager`.

A `@PostPersist` or `@PostRemove` callback occurs after the effect of a respective persist or remove operation for an entity has been realized.

- The `@PostPersist` callback for an entity which has been made persistent occurs after the entity has been marked as managed and associated with the persistence context, and after the database insert operations have been executed.
- The `@PostRemove` callback for an entity which has been removed occurs after the entity has been marked as removed in the persistence context, and after the database delete operations have been executed.

The database insert or delete operations might occur immediately during execution of `persist()`, `merge()`, or `remove()`, or they might occur later when the persistence context is flushed. The persistence context might not be flushed until the

end of the transaction, or, if the transaction rolls back, it might not be flushed at all. If the persistence context is never flushed, and as a result the insert or delete operations are never executed, the `@PostPersist` or `@PostRemove` callback never occurs.

Generated primary key values might not be available during a `@PrePersist` callback.

- Primary key values generated using the `SEQUENCE`, `TABLE`, or `UUID` strategy are available in the `@PrePersist` method. Primary key values generated using the `IDENTITY` strategy are not available in the `@PrePersist` method.
- Generated primary key values are always available in the `@PostPersist` method.

When a persist, remove, or merge operation cascades to an associated entity, the corresponding `@PrePersist`, `@PreRemove`, or `@PreMerge`, and `@PostPersist` or `@PostRemove` callbacks also occur for the associated entity.

The `@PreInsert`, `@PostInsert`, `@PreUpdate`, `@PostUpdate`, `@PreDelete` and `@PostDelete` callbacks occur before and after database insert, update, and delete operations to entity data, respectively. These callbacks occur whenever entity state is modified via `EntityManager` or `EntityAgent`, and are not tied to the lifecycle of managed entities associated with a persistence context. These callbacks are not triggered when entity data is modified by execution of a `Query`.

In this context, the *entity data* belonging to an entity instance is considered to include:

- every basic or embedded attribute of the entity instance, recursively,
- every `@ElementCollection` belonging to the entity instance or to an embeddable object belonging to the instance,
- every basic or embedded attribute of an embeddable object belonging to an `@ElementCollection` belonging to the entity instance or to an embeddable object belonging to the instance, recursively,
- every `@ManyToOne` association field or property of the entity instance or of an embeddable object belonging to the instance (but not the state of the associated target entity),
- every *owned* `@OneToOne` association field or property of the entity instance or of an embeddable object belonging to the instance, and
- every *owned* `@OneToMany` or `@ManyToMany` association belonging to the entity instance or to an embeddable object belonging to the instance.

Therefore, changes to the persistent state of owned associations result in invocation of the `@PreUpdate` and `@PostUpdate` callbacks. Changes which modify only unowned associations never result in `@PreUpdate` or `@PostUpdate` callbacks.

> It is implementation-dependent as to whether `@PreUpdate` and `@PostUpdate` callbacks occur when an entity is persisted and then subsequently modified in a single transaction, or when an entity is modified and subsequently removed within a single transaction. Portable applications should not rely on such behavior.

The `@PostLoad` callback occurs:

- after an entity instance has been loaded from the database by any means, or
- after the `refresh()` operation of `EntityManager` or `EntityAgent` has been applied to an entity instance.

The non-relationship state of an entity is always set to the values just retrieved from the database before the `@PostLoad` method is invoked. The `@PostLoad` method is invoked before a newly loaded entity is returned or made accessible to the application.

> This specification does not define whether callback methods are invoked before or after an operation cascades to associated entities. Applications should not depend on this ordering.

For example:

```java
@Entity
@EntityListeners(com.acme.AlertMonitor.class)
public class Account {
    Long accountId;
    Integer balance;
    boolean preferred;

    @Id
    public Long getAccountId() { ... }

    // ...

    public Integer getBalance() { ... }

    // ...

    @Transient // because status depends upon non-persistent context
    public boolean isPreferred() { ... }

    // ...

    public void deposit(Integer amount) { ... }

    public Integer withdraw(Integer amount) throws NSFException { ... }

    @PrePersist
    protected void validateCreate() {
        if (getBalance() < MIN_REQUIRED_BALANCE)
            throw new AccountException("Insufficient balance to open an account");
    }

    @PostLoad
    protected void adjustPreferredStatus() {
        preferred = (getBalance() >= AccountManager.getPreferredStatusLevel());
    }
}

public class AlertMonitor {
    @PostPersist
    public void newAccountAlert(Account acct) {
        Alerts.sendMarketingInfo(acct.getAccountId(), acct.getBalance());
    }
}
```

3.10.3. Multiple Lifecycle Callback Methods for an Entity Lifecycle Event

If multiple callback methods are defined for an entity lifecycle event, the order of invocation of the callback methods is governed by the following rules.

1. Default listeners, if any, are invoked first, in the order specified in the XML descriptor.
2. Listeners declared using the `EntityListener` annotation are invoked after default listeners, and before listeners declared by the `EntityListeners` annotation. This specification does not define any specific order for the invocation of these listeners.
3. Listeners declared by a module descriptor, via the `EntityListeners` annotation, are invoked in the same order in which they are declared by the annotation.
4. Listeners declared by a package descriptor, via the `EntityListeners` annotation, are invoked in the same order in which they are declared by the annotation.
5. Listeners declared by the entity class itself, via the `EntityListeners` annotation or `<entity-listeners>` XML element, are invoked in the same order in which they are declared by the annotation or XML element. If multiple classes in an entity inheritance hierarchy—including both entity classes and mapped superclasses—declare entity listeners, the

listeners declared by a superclass are invoked before the listeners declared by its subclasses.

A *default listener* is an entity listener declared:

- under `<persistence-unit-defaults>` in the XML descriptor, or
- by applying `@EntityListeners` to a module or package descriptor.

A default listener declared using XML applies to all entities in the persistence unit, unless explicitly excluded by means of the `ExcludeDefaultListeners` annotation or `<exclude-default-listeners>` XML element. A default listener declared by annotating a package or module descriptor applies to every entity contained in the package or module, unless explicitly excluded by means of the `ExcludeDefaultListeners` annotation or `<exclude-default-listeners>` XML element. The `ExcludeDefaultListeners` annotation or `exclude-default-listeners` XML element may be applied to an entity class or mapped superclass to exclude the invocation of default listeners.

The `ExcludeSuperclassListeners` annotation or `exclude-superclass-listeners` XML element may be applied to an entity class or mapped superclass to exclude the invocation of the listeners defined by the entity listener classes for the superclasses of the entity or mapped superclass. The excluded listeners are excluded from the class to which the `ExcludeSuperclassListeners` annotation or element has been specified and its subclasses[21]. The `ExcludeSuperclassListeners` annotation (or `exclude-superclass-listeners` XML element) does not cause default entity listeners to be excluded from invocation.

If a lifecycle callback method for the same lifecycle event is also specified on the entity class and/or one or more of its entity or mapped superclasses, the callback methods on the entity class and/or superclasses are invoked after the other lifecycle callback methods, most general superclass first. A class is permitted to override an inherited callback method of the same callback type, and in this case, the overridden method is not invoked[22].

Callback methods are invoked by the persistence provider runtime in the order specified. If the callback method execution terminates normally, the persistence provider runtime then invokes the next callback method, if any.

The XML descriptor may be used to override the lifecycle callback method invocation order specified in annotations.

For example:

There are several entity classes and listeners for animals:

```
@Entity
public class Animal {

    // ...

    @PostPersist
    protected void postPersistAnimal() {
        // ...
    }
}

@Entity
@EntityListeners(PetListener.class)
public class Pet extends Animal {
    // ...
}

@Entity
@EntityListeners({CatListener.class, CatListener2.class})
public class Cat extends Pet {
    // ...
}

public class PetListener {
    @PostPersist
```

```
    protected void postPersistPetListenerMethod(Object pet) {
        // ...
    }
}

public class CatListener {
    @PostPersist
    protected void postPersistCatListenerMethod(Object cat) {
        // ...
    }
}

public class CatListener2 {
    @PostPersist
    protected void postPersistCatListener2Method(Object cat) {
        // ...
    }
}
```

If a `PostPersist` event occurs on an instance of `Cat`, the following methods are called in order:

1. `postPersistPetListenerMethod`
2. `postPersistCatListenerMethod`
3. `postPersistCatListener2Method`
4. `postPersistAnimal`

Assume that `SiameseCat` is defined as a subclass of `Cat`:

```
@EntityListeners(SiameseCatListener.class)
@Entity
public class SiameseCat extends Cat {
    // ...

    @PostPersist
    protected void postPersistSiameseCat() {
        // ...
    }
}

public class SiameseCatListener {
    @PostPersist
    protected void postPersistSiameseCatListenerMethod(Object cat) {
        // ...
    }
}
```

If a `PostPersist` event occurs on an instance of `SiameseCat`, the following methods are called in order:

1. `postPersistPetListenerMethod`
2. `postPersistCatListenerMethod`
3. `postPersistCatListener2Method`
4. `postPersistSiameseCatListenerMethod`
5. `postPersistAnimal`
6. `postPersistSiameseCat`

Assume the definition of `SiameseCat` were instead:

```
@EntityListeners(SiameseCatListener.class)
@Entity
public class SiameseCat extends Cat {
```

```
    // ...

    @PostPersist
    protected void postPersistAnimal() {
        // ...
    }
}
```

In this case, the following methods would be called in order, where `postPersistAnimal` is the `PostPersist` method defined in the `SiameseCat` class:

1. `postPersistPetListenerMethod`
2. `postPersistCatListenerMethod`
3. `postPersistCatListener2Method`
4. `postPersistSiameseCatListenerMethod`
5. `postPersistAnimal`

3.10.4. Exceptions

Lifecycle callback methods may throw runtime exceptions. A runtime exception thrown by a callback method that executes within a transaction causes that transaction to be marked for rollback if the persistence context is joined to the transaction. No further lifecycle callback methods will be invoked after a runtime exception is thrown.

3.10.5. Entity Listeners and Dependency Injection

The entity listener class must have a public constructor with no parameters.

As specified in Section 9.10, an entity listener class may be defined to make use of dependency injection through the Contexts and Dependency Injection (CDI) API [7] when CDI is available.

Entity listeners that do not make use of CDI injection are stateless. The lifecycle of such entity listeners is unspecified.

3.10.6. Lifecycle Callbacks, Transactions, and Security

When invoked within a Jakarta EE environment, the lifecycle callback listeners for an entity share the enterprise naming context of the invoking component, and the entity callback methods are invoked in the transaction and security contexts of the calling component at the time at which the callback method is invoked. [23]

3.10.7. Specification of Callback Listener Classes and Lifecycle Methods in the XML Descriptor

The XML descriptor can be used as an alternative to metadata annotations to specify entity listener classes and their binding to entities or to override the invocation order of lifecycle callback methods as specified in annotations.

3.10.7.1. Specification of Callback Listeners

The `entity-listener` XML descriptor element specifies the lifecycle listener methods of an entity listener class. The lifecycle listener methods are specified by using the `pre-persist`, `post-persist`, `pre-remove`, `post-remove`,`pre-merge`, `pre-insert`, `post-insert`, `pre-delete`, `post-delete`, `pre-upsert`, `post-upsert`, `pre-update`, `post-update`, and/or `post-load` elements.

3.10.7.2. Specification of the Binding of Entity Listener Classes to Entities

The `entity-listeners` subelement of the `persistence-unit-defaults` element specifies the default entity listeners for the persistence unit.

The `entity-listeners` subelement of the `entity` or `mapped-superclass` element specifies the entity listener classes for the respective entity or mapped superclass and its subclasses.

The binding of entity listeners to entity classes is additive. The entity listener classes bound to the superclasses of an entity or mapped superclass are applied to it as well.

The `exclude-superclass-listeners` element specifies that the listener methods for superclasses are not to be invoked for an entity class (or mapped superclass) and its subclasses.

The `exclude-default-listeners` element specifies that default entity listeners are not to be invoked for an entity class (or mapped superclass) and its subclasses.

Explicitly listing an excluded default or superclass listener for a given entity class or mapped superclass causes it to be applied to that entity or mapped superclass and its subclasses.

In the case of multiple callback methods for a single lifecycle event, the invocation order rules described in Section 3.10.3 apply.

3.11. Jakarta Validation integration

Jakarta Persistence defines integration points with Jakarta Validation [5]. When the Jakarta Persistence provider is aware of the Jakarta Validation provider's presence, and the integration is enabled, the persistence provider must:

- perform automatic validation upon lifecycle events: The provider must automatically validate managed types (entities, mapped superclasses, and embeddable classes) at the entity lifecycle events described in Section 3.10.2. This process occurs internally within the persistence lifecycle, ensuring validation without requiring explicit application logic beyond the constraint definitions.
- Derive schema metadata from Jakarta Validation constraints: The provider must utilize specific Jakarta Validation constraint metadata to derive persistent attribute characteristics (such as nullability, length, precision) and database constraints for the purpose of DDL schema generation.

3.11.1. Configuring the Jakarta Validation provider

The persistence provider is not expected to implement Jakarta Validation. Instead, an instance of `jakarta.validation.ValidatorFactory` must be made available to the provider.

3.11.1.1. Providing the ValidatorFactory

In Jakarta EE environments, a `ValidatorFactory` instance is made available by the Jakarta EE container. The container is responsible for passing this validator factory to the persistence provider via a map, which is passed as the second argument to `createContainerEntityManagerFactory`. The map key must be the standard property name `jakarta.persistence.validation.factory`.

In Java SE environments, the application can supply the `ValidatorFactory` instance via a map passed as the second argument to `Persistence.createEntityManagerFactory`. The map key must be the standard property name `jakarta.persistence.validation.factory`. If no `ValidatorFactory` instance is provided by the application, and if a Jakarta Validation provider is present in the classpath, the persistence provider must instantiate the `ValidatorFactory` using the default bootstrapping approach defined by the Jakarta Validation specification [5], namely `Validation.buildDefaultValidatorFactory()`.

3.11.1.2. Enabling/Disabling the Jakarta Validation integration

Enabling and disabling the Jakarta Validation integration is achieved through configuring the validation mode.

This specification defines the following *validation modes*:

- `CALLBACK` — the persistence provider must perform the automatic validation of entities as described in Section 3.11.2.1. If no Jakarta Validation provider is available, the persistence provider must throw a `PersistenceException`.
- `NONE` — the persistence provider must not perform any automatic validation.
- `AUTO` — if a Jakarta Validation provider is obtainable — persistent provider must perform the automatic validation of entities. Otherwise, the behavior is the same as with `none`.

> The Jakarta Validation provider is obtained via the `jakarta.validation.ValidatorFactory` (Section 3.11.1.1).

The validation mode may be specified via the `validation-mode` element of the `persistence.xml` file. Allowed values for the `validation-mode` element are `AUTO`, `CALLBACK` and `NONE`.

The validation mode value specified in `persistence.xml` may be overridden by the configuration property `jakarta.persistence.validation.mode`, which may be passed to `Persistence.createEntityManagerFactory` or `PersistenceProvider.createContainerEntityManagerFactory`, as defined in Section 9.7 and Section 9.1. Expected values for this property are `auto`, `callback` and `none`.

> To ensure deterministic behavior and prevent silent validation omissions, applications should always explicitly set the `validation-mode` to `CALLBACK` rather than relying on the default.

The persistence provider must automatically utilize Jakarta Validation metadata to derive persistent attribute characteristics and DDL constraints whenever a validation provider is present in the environment. This behavior is independent of the runtime validation mode.

3.11.2. Automatic Validation Upon Lifecycle Events

Automatic validation of entities is specified in terms of their lifecycle events defined in Section 3.10.2.

If enabled, automatic validation of entities must occur immediately after the point at which all corresponding lifecycle callback method invocations complete, or immediately after the point at which such lifecycle callback methods would have completed if there were any such callback methods present.

> Automatic validation is triggered as an immediate post-action of a lifecycle callback. When multiple operations are applied to a single entity within the scope of a single transaction, the number of automatic validation events is implementation-dependent and may vary across Jakarta Persistence providers.
>
> For example, it is provider-specific whether a `PreUpdate` callback is invoked if an entity is persisted and subsequently modified, or modified and subsequently removed, within the same transaction Section 3.10.2. Consequently, the automatic validation associated to the pre-update lifecycle callback may or may not occur in these scenarios.
>
> Portable applications must not rely on the specific timing or ordering of automatic validation events.

3.11.2.1. Requirements for Automatic Validation upon Lifecycle Events

Each lifecycle event type has a list of validation groups assigned to it. If the list of validation groups for a particular

event is nonempty, validation is applied; otherwise, no validation is applied during that particular event.

By default, only the lists of validation groups for the pre-insert, pre-update and pre-upsert lifecycle events contain the `jakarta.validation.groups.Default` group (the default Jakarta Validation group). The validation group lists for other lifecycle events are empty.

This default validation behavior can be overridden by specifying the target groups using the following configuration properties in the `persistence.xml` file or by passing these properties in the configuration of the entity manager factory through the `createEntityManagerFactory` method:

- `jakarta.persistence.validation.group.pre-persist`
- `jakarta.persistence.validation.group.pre-merge`
- `jakarta.persistence.validation.group.pre-remove`
- `jakarta.persistence.validation.group.pre-insert`
- `jakarta.persistence.validation.group.pre-update`
- `jakarta.persistence.validation.group.pre-upsert`
- `jakarta.persistence.validation.group.pre-delete`

The value of a configuration property must be a list of the targeted groups. A targeted group must be specified by its fully qualified class name. Each pair of names must be separated by a comma. Optional whitespace characters shall be permitted both preceding and following each group name.

When an entity lifecycle event occurs, the persistence provider must validate the entity using a `Validator` instance obtained from the validator factory in use (see Section 3.11.1.1) and invoke its `validate` method with the targeted groups. If the list of targeted groups is empty, no validation is performed. If the set of `ConstraintViolation` objects returned by the `validate` method is not empty, the persistence provider must throw the `jakarta.validation.ConstraintViolationException` containing a reference to the returned set of `ConstraintViolation` objects, and must mark the transaction for rollback if the persistence context is joined to the transaction.

The validation process must not have any side effects; validation must not result in the loading of any attribute or association which is not already loaded. An entity can be validated at most once per flush cycle.

To achieve this, persistence provider must initialize the validator instance in use with the traversable resolver (`jakarta.validation.TraversableResolver`) that matches these behavior requirements. In particular:

- `TraversableResolver.isReachable(..)` only returns `true` for attributes and associations which are already loaded.
- `TraversableResolver.isCascadable(..)` only returns `true` for attributes that are not associations.

A validator instance with such a traversable resolver can be obtained from a `jakarta.validation.ValidatorFactory` instance by calling `validatorFactory.usingContext().traversableResolver(tr).getValidator()`, where `tr` is the resolver.

3.12. Entity Graphs

An entity graph is a template that captures the path and boundaries of an operation or query. It might be defined in the form of metadata or as an object created at runtime via the programmatic `EntityGraph` API.

Every entity graph has a *root entity*. The root node of the graph represents this entity. The child nodes of the root node represent persistent attributes, embedded objects, or associations to other entities. A node representing an embedded object or an association to an entity might be the root of a subgraph with its own child nodes.

The *attribute nodes* and *subgraphs* of an entity graph control the limits of the graph of associated attributes and entities fetched when an operation which retrieves an instance or instances of the root entity of the graph is executed.

The interface `EntityGraph`, which may be found in Appendix B, represents a graph at runtime. Its child attribute nodes

and subgraphs are represented as instances of `AttributeNode` and `Subgraph`, also found in Appendix B. These interfaces may be used to construct an entity graph programmatically.

The annotations `@NamedEntityGraph`, `@NamedAttributeNode`, and `@NamedSubgraph` described in Section 10.3 are used to statically define entity graphs. The `named-entity-graph` XML element and its subelements may be used to override these annotations or to define additional named entity graphs. As an alternative to nesting `@NamedAttributeNode` and `@NamedSubgraph` within `@NamedEntityGraph`, the annotation `@Fetch`, also found in Section 10.3, may be applied to persistent attributes to specify the nodes and subgraphs of a named entity graph declared via the `@NamedEntityGraph` annotation.

A well-defined entity graph is a tree. The application program is responsible for ensuring that every named entity graph is acyclic. If a named graph contains a cycle, the behavior is undefined. A provider may reject the graph at runtime; it may reject it when the persistence unit is initialized; or it may transform the graph into an acyclic graph according to some vendor-specific algorithm. Portable applications must avoid relying on such behavior.

When the `subgraph` member of `@Fetch` is used, there might be multiple entities with `@NamedEntityGraph` annotations specifying identical values for the `name` member. In this scenario, the root entity of the named graph is identified by it being the unique entity class that is not referenced by the `subgraph` member of any `Fetch` annotation belonging to the named graph. If there is no such entity class, or if it is not unique, the behavior is undefined. An application must ensure that each named entity graph has a unique root entity.

The semantics of entity graphs with regard to find and query operations are described in Section 3.12.1.

3.12.1. Use of Entity Graphs in find and query operations

Use of an entity graph may be requested by passing the instance of `EntityGraph` to:

- the first parameter of the `find(EntityGraph, Object, FindOption…)` method of `EntityHandler`, to control the data fetched by `find()`,
- the second parameter of the `createQuery(String, EntityGraph)` method of `EntityHandler`, or to the `withEntityGraph()` method of `StatementOrTypedQuery`, to control the data fetched by the JPQL query, or
- either `find()` method of `EntityManager` accepting a properties map, or the `setHint()` method of `Query`, via one of the standard properties `jakarta.persistence.fetchgraph` or `jakarta.persistence.loadgraph`.

An entity graph is interpreted as *load graph*, as specified below in Section 3.12.1.2 unless it is passed as a hint via the property `jakarta.persistence.fetchgraph`, in which case it is interpreted as a *fetch graph*, as specified by Section 3.12.1.1.

The *default fetch graph* for an entity or embeddable is defined to consist of the transitive closure of every attribute that is declared as `fetch=FetchType.EAGER` (or defaulted as such). This fetch graph is used whenever `find()` is called with no explicit entity graph. The default fetch graph does not affect JPQL queries. In the absence of an explicit entity graph, the data fetched by a query is determined by the query itself.

The persistence provider is always permitted to fetch additional entity state beyond that specified by a fetch graph or load graph. It is required, however, that the persistence provider fetch all state specified by the fetch or load graph.

3.12.1.1. Fetch Graph Semantics

When an entity graph is interpreted as a fetch graph, that is, when it is specified via the property `jakarta.persistence.fetchgraph`:

- an attribute that is explicitly declared as an attribute node of the entity graph is treated as if it had been mapped with `FetchType.EAGER`, and
- an attribute that is not declared as a node is treated as if had been mapped with `FetchType.LAZY`.

The following rules apply. The rules of this section are applied recursively.

A primary key or version attribute never needs to be specified in an attribute node of a fetch graph. (This applies to composite primary keys as well, including embedded id primary keys.) When an entity is fetched, its primary key and version attributes are always fetched. It is not incorrect, however, to specify primary key attributes or version attributes.

Attributes other than primary key and version attributes are assumed not to be fetched unless the attribute is specified. The following rules apply to the specification of attributes.

- If the attribute is an embedded attribute, and the attribute is specified in an attribute node, but a subgraph is not specified for the attribute, the default fetch graph for the embeddable is fetched. If a subgraph is specified for the attribute, the attributes of the embeddable are fetched according to their specification in the corresponding subgraph.
- If the attribute is an element collection of basic type, and the attribute is specified in an attribute node, the element collection together with its basic elements is fetched.
- If the attribute is an element collection of embeddables, and the attribute is specified in an attribute node, but a subgraph is not specified for the attribute, the element collection together with the default fetch graph of its embeddable elements is fetched. If a subgraph is specified for the attribute, the attributes of the embeddable elements are fetched according to the corresponding subgraph specification.
- If the attribute is a one-to-one or many-to-one relationship, and the attribute is specified in an attribute node, but a subgraph is not specified for the attribute, the default fetch graph of the target entity is fetched. If a subgraph is specified for the attribute, the attributes of the target entity are fetched according to the corresponding subgraph specification.
- If the attribute is a one-to-many or many-to-many relationship, and the attribute is specified in an attribute node, but a subgraph is not specified, the collection is fetched and the default fetch graphs of the referenced entities are fetched. If a subgraph is specified for the attribute, the entities in the collection are fetched according to the corresponding subgraph specification.
- If the key of a map which has been specified in an attribute node is a basic type, it is fetched. If the key of a map which has been specified in an attribute node is an embedded type, the default fetch graph is fetched for the embeddable. Otherwise, if the key of the map is an entity, and a map key subgraph is not specified for the attribute node, the map key is fetched according to its default fetch graph. If a key subgraph is specified for the map key attribute, the map key attribute is fetched according to the map key subgraph specification.

Examples:

```
@NamedEntityGraph // graph name defaults to "PhoneNumber"
@Entity
public class PhoneNumber {
    @Id
    protected String number;

    protected PhoneTypeEnum type;

    // ...
}
```

In the above example, only the `number` attribute would be eagerly fetched.

```
@NamedEntityGraph(name = "employeeWithProjects")
@Entity
public class Employee {
    @Id
    @GeneratedValue
    protected long id;

    @Basic
```

```
    protected String name;

    @Basic
    protected String employeeNumber;

    @OneToMany
    protected List<Dependents> dependents;

    @OneToMany
    // include projects as a node in the graph
    @Fetch(graph = "employeeWithProjects")
    protected List<Project> projects;

    @OneToMany
    protected List<PhoneNumber> phoneNumbers;

    // ...
}

@Entity
@Inheritance
public class Project {
    @Id
    @GeneratedValue
    protected long id;

    String name;

    @OneToOne(fetch=FetchType.EAGER)
    protected Requirements doc;

    // ...
}

@Entity
public class LargeProject extends Project {
    @OneToOne(fetch=FetchType.LAZY)
    protected Employee approver;

    // ...
}

@Entity
public class Requirements {
    @Id
    protected long id;

    @Lob
    protected String description;

    @OneToOne(fetch=FetchType.LAZY)
    protected Approval approval;

    // ...
}
```

In the above example, the `Employee` entity's primary key will be fetched as well as the related `Project` instances, whose default fetch graph (`id`, `name`, and `doc` attributes) will be fetched. The related `Requirements` object will be fetched according to its default fetch graph.

If the `approver` attribute of `LargeProject` were `FetchType.EAGER`, and if any of the projects were instances of `LargeProject`, their `approver` attributes would also be fetched. Since the type of the `approver` attribute is `Employee`, the approver's default fetch graph (`id`, `name`, and `employeeNumber` attributes) would also be fetched.

3.12.1.2. Load Graph Semantics

When an entity graph is interpreted as a load graph:

- an attribute that is explicitly declared as an attribute node of the entity graph is treated as if it had been mapped with `FetchType.EAGER`, and
- an attribute that is not declared as a node is treated according to its specified or defaulted `FetchType` of its default fetch group.

The following rules apply. The rules of this section are applied recursively.

- A primary key or version attribute never needs to be specified in an attribute node of a load graph. (This applies to composite primary keys as well, including embedded id primary keys.) When an entity is fetched, its primary key and version attributes are always fetched. It is not incorrect, however, to specify primary key attributes or version attributes.
- If the attribute is an embedded attribute, and the attribute is specified in an attribute node, but a subgraph is not specified for the attribute, the default fetch graph for the embeddable is fetched. If a subgraph is specified for the attribute, attributes that are specified by the subgraph are also fetched.
- If the attribute is an element collection of basic type, and the attribute is specified in an attribute node, the element collection together with its basic elements is fetched.
- If the attribute is an element collection of embeddables, and the attribute is specified in an attribute node, the element collection together with the default fetch graph of its embeddable elements is fetched. If a subgraph is specified for the attribute, attributes that are specified by the subgraph are also fetched.
- If the attribute is a one-to-one or many-to-one relationship, and the attribute is specified in an attribute node, the default fetch graph of the target entity is fetched. If a subgraph is specified for the attribute, attributes that are specified by the subgraph are also fetched.
- If the attribute is a one-to-many or many-to-many relationship, and the attribute is specified in an attribute node, the collection is fetched and the default fetch graphs of the referenced entities are fetched. If a subgraph is specified for the attribute, attributes that are specified by the subgraph are also fetched.
- If the key of a map which has been specified in an attribute node is a basic type, it is fetched. If the key of a map which has been specified in an attribute node is an embedded type, the default fetch graph is fetched for the embeddable. Otherwise, if the key of the map is an entity, the map key is fetched according to its default fetch graph. If a key subgraph is specified for the map key attribute, additional attributes are fetched as specified in the key subgraph.

Examples:

```
@NamedEntityGraph // graph name defaults to "PhoneNumber"
@Entity
public class PhoneNumber {
    @Id
    protected String number;

    protected PhoneTypeEnum type;

    // ...
}
```

In the above example, the `number` and `type` attributes are fetched.

```
@NamedEntityGraph(name = "employeeWithPhoneNumbers")
@Entity
public class Employee {
    @Id
    @GeneratedValue
```

```
    protected long id;

    @Basic
    protected String name;

    @Basic
    protected String employeeNumber;

    @OneToMany
    protected List<Dependents> dependents;

    @OneToMany
    protected List<Project> projects;

    @OneToMany
    // include phoneNumbers as a node in the graph
    @Fetch(graph = "employeeWithPhoneNumbers")
    protected List<PhoneNumber> phoneNumbers;

    // ...
}

@Entity
@Inheritance
public class Project {
    @Id
    @GeneratedValue
    protected long id;

    String name;

    @OneToOne(fetch=FetchType.EAGER)
    protected Requirements doc;

    // ...
}

@Entity
public class LargeProject extends Project {
    @OneToOne(fetch=FetchType.LAZY)
    protected Employee approver;

    // ...
}

@Entity
public class Requirements {
    @Id
    protected long id;

    @Lob
    protected String description;

    @OneToOne(fetch=FetchType.LAZY)
    protected Approval approval;

    // ...
}
```

In the above example, the default fetch graph (`id`, `name`, `employeeNumber` attributes) of `Employee` is fetched. The default fetch graphs of the related `Project` instances (`id`, `name`, and `doc` attributes) and their `Requirements` instances (`id` and `description` attributes) are also fetched.

3.13. Type Conversion of Basic Attributes

An *attribute converter* is a class whose methods convert between:

- the *target type* of the converter, an arbitrary Java type which may be used as the type of a persistent field or property, and
- a basic type, as defined in Section 2.6, used as an intermediate step in mapping to the database representation.

A converter can be used to convert attributes defined by entity classes, mapped superclasses, or embeddable classes. A converted attribute is considered a basic attribute, since, with the aid of the converter, its values can be represented as instances of a basic type.

Every attribute converter class must implement the interface `AttributeConverter` found in Section B.20 and must be annotated with the `@Converter` annotation defined in Section 10.6 or declared as a converter in the XML descriptor.

Attribute conversion is not supported for:

- the primary key of an entity, as defined in Section 2.4, nor for an attribute of a composite primary key or an attribute involved in a derived identity,
- an entity version, as defined in Section 2.5,
- a relationship attribute, as defined in Section 2.11,
- a basic attribute explicitly annotated as `@Enumerated` or `@Temporal` or designated as such in the XML descriptor.

Attribute conversion is supported for element collections, as defined in Section 2.8, with the conversion applying to the elements of the collection.

3.13.1. Auto-Apply Converters

A converter declared `@Converter(autoApply=true)` is automatically applied to:

- every attribute of an entity class, mapped superclass, or embeddable class whose declared type matches the target type of the converter, unless the converter is overridden or disabled through application of the `@Convert` annotation to the attribute, as specified below, and
- every element collection whose element type matches the target type of the converter, unless the converter is overridden or disabled through application of the `@Convert` annotation to the collection-valued attribute, as specified below.

An automatically applied attribute converter must implement the type `AttributeConverter<X,Y>`, where:

- `X` is the target type of the converter, and
- `Y` is a basic type.

In determining whether such an *auto-apply converter* applies to an attribute, the provider must treat primitive types and wrapper types as equivalent.

An automatically applied converter is not applied to a field or property of an entity for which attribute conversion is not supported according to the rules specified above.

3.13.2. Explicit Converters

Conversion of an individual attribute is controlled by the `@Convert` annotation defined in Section 11.2.11.

- The `@Convert` annotation may be applied directly to an attribute of an entity, mapped superclass, or embeddable class to specify conversion of the attribute, overriding or disabling application of any automatically applied converter.

When property access is used as specified in Section 2.2.3, the `@Convert` annotation must be applied to the getter method.

- The `@Convert` annotation may be applied to an entity that extends a mapped superclass to specify or override the conversion mapping for an inherited basic or embedded attribute.

An explicitly applied attribute converter must implement the type `AttributeConverter<X,Y>`, where:

- `X` is the declared type of the attribute, and
- `Y` is a basic type.

A persistence provider is not required to support the application of the `@Convert` annotation to a field or property of an entity for which attribute conversion is not supported according to the rules specified above.

3.13.3. Converter Invocation

The persistence provider runtime must:

- call `convertToDatabaseColumn()` before writing the value of an entity attribute in the database, and
- call `convertToEntityAttribute()` after reading the value of an entity attribute from the database.

Before executing a JPQL or criteria query, the persistence provider might need to apply a converter to a literal value occurring in the query or to the argument value bound to a parameter of the query. In particular, when a literal or parameter is compared to or assigned to an attribute with a converter, the converter must be applied to the value of the literal or parameter if the type of the literal or parameter matches the unconverted type of the attribute.

The persistence provider might need to apply a converter to a value returned by a JPQL or criteria query. In particular, if the result of a JPQL query or criteria query includes an entity attribute with a converter, the persistence provider must apply the converter to the corresponding value in the query result before returning it to the application.

The persistence provider is not required to support the use of a converted attribute as an argument of a JPQL function or aggregate function.

If an exception is thrown by a conversion method, the persistence provider must wrap the exception in a `PersistenceException` and, if the persistence context is joined to a transaction, mark the transaction for rollback.

3.13.4. Attribute Converters and Dependency Injection

As specified in Section 9.10, an attribute converter class may be defined to make use of dependency injection through the Contexts and Dependency Injection (CDI) API [7] when CDI is available.

Attribute converters that do not make use of CDI injection are stateless. The lifecycle of such attribute converters is unspecified.

3.14. Second-Level Cache

A persistence provider may support the use of a *second-level cache*, that is, it might have a way to store data read in one persistence context for use in subsequent persistence contexts. A second-level cache might enhance performance, but tends to undermine the semantics of transaction processing, possibly exposing the application to stale data or similar anomalies.

Access to the second-level cache, if enabled, is mediated via the persistence context, and is largely transparent to the application. As an exception, the `Cache` interface described below in Section 3.14.3 allows the application to directly evict data from the second-level cache.

The persistence provider is not required to support use of a second-level cache.

3.14.1. The Shared Cache Mode and Cacheable Annotation

Whether a given entity is eligible for storage in the second level cache is determined by:

- the annotations of the entity class, and
- the value specified for the `shared-cache-mode` element of the `persistence.xml` file, by the `sharedCacheMode()` of the `PersistenceConfiguration`, or by the configuration property `jakarta.persistence.sharedCache.mode`.

The value of the property `jakarta.persistence.sharedCache.mode` takes precedence over the value of the `shared-cache-mode` element or `sharedCacheMode()` of the `PersistenceConfiguration`.

The `shared-cache-mode` element takes one of five possible values, which are enumerated by `SharedCacheMode`:

- `ALL` specifies that every entity type associated with the persistence unit and all entity-related state and data is eligible for storage in the second-level cache.
- `NONE` specifies that the second-level cache is disabled for the whole persistence unit and that no entity type is eligible for storage in the second-level cache.
- `ENABLE_SELECTIVE` specifies that an entity type is eligible for storage in the second-level cache if and only if it is annotated `Cacheable` or `Cacheable(true)`, or if the equivalent setting is specified in XML.
- `DISABLE_SELECTIVE` specifies that an entity type is eligible for storage in the second-level cache if and only if it is not annotated `Cacheable(false)`.
- `UNSPECIFIED` selects the provider-specific default behavior.

If neither the `shared-cache-mode` element nor the property `jakarta.persistence.sharedCache.mode` is specified, or if the specified value is `UNSPECIFIED`, the behavior is not defined, and provider-specific defaults may apply. In particular, the semantics of the `Cacheable` annotation (and XML equivalent) is undefined.

If the persistence provider does not support use of a second-level cache, or if a second-level cache is not installed or not enabled, this setting may be ignored and no caching will occur.

A persistence provider may support additional vendor-specific mechanisms for configuring the cache and marking entities eligible (or not) for storage in the second-level cache. However, if a second-level cache is supported, and enabled, the provider must respect the configuration options defined in this section, if specified by the application.

3.14.2. Cache Modes

The *cache retrieve mode* and *cache store mode* control how a given persistence context interacts with the second-level cache.

- The cache retrieve mode may be set by calling `setCacheRetrieveMode()` on `EntityManager`, `EntityAgent`, or `Query`.
- The cache store mode may be set by calling `setCacheStoreMode()` on `EntityManager`, `EntityAgent`, or `Query`.
- A cache store mode or cache retrieve mode, or both, may be passed to the `find()` method of `EntityManager` as a `FindOption`.
- A cache store mode may be passed to the `refresh()` method of `EntityManager` as a `RefreshOption`.

A cache mode specified for a given `Query` instance applies only to executions of that query, but takes precedence over the current cache mode of the `EntityManager` to which the `Query` belongs. A cache mode passed to `find()` or `refresh()` applies only to the method invocation, and takes precedence over the current cache mode of the `EntityManager`.

Alternatively, a cache mode may be specified using the property name `jakarta.persistence.cache.retrieveMode` or `jakarta.persistence.cache.storeMode` by:

- calling the `setProperty()` method of `EntityManager` or `EntityAgent`,
- calling the `setHint()` method of `Query`, or
- passing a map containing one of these properties to `find()` or `refresh()`.

If second-level caching is not enabled (for example, if the `shared-cache-mode` element is set to `NONE`), cache modes must be ignored. Similarly, if a given entity is not eligible for storage in the second-level cache (for example, if the `shared-cache-mode` element is set to `ENABLE_SELECTIVE`, and the entity is not annotated `@Cacheable`), cache modes are ignored for operations applying to that entity.

Cache modes must be respected when caching is enabled, regardless of whether caching is enabled via the configuration options defined by this specification or via provider-specific mechanisms.

Applications which depend on the cache retrieve mode or cache store mode but which do not specify the `shared-cache-mode` element are not portable.

`CacheRetrieveMode` enumerates the cache retrieve modes recognized by this specification. The semantics of each mode is defined by its Javadoc.

```
/**
 * Specifies how the {@link EntityManager} interacts with the
 * second-level cache when data is read from the database via
 * the {@link EntityManager#find} methods and execution of
 * queries.
 * <ul>
 * <li>{@link #USE} indicates that data may be read from the
 *     second-level cache.
 * <li>{@link #BYPASS} indicates that data must not be read
 *     from the second-level cache.
 * </ul>
 *
 * <p>Enumerates legal values of the property
 * {@code jakarta.persistence.cache.retrieveMode}.
 *
 * @see EntityHandler#setCacheRetrieveMode(CacheRetrieveMode)
 * @see TypedQuery#setCacheRetrieveMode(CacheRetrieveMode)
 *
 * @since 2.0
 */
public enum CacheRetrieveMode
        implements FindOption, FetchOption, TypedQuery.Option,
                   EntityManager.Option, EntityAgent.Option {

    /**
     * Specifies that data may be read from the second-level cache
     * instead of from the database.
     * <p>
     * This is the default mode.
     */
    USE,

    /**
     * Specifies that data must not be read from the second-level
     * cache and must be read from the database.
     *
     * @apiNote When used as a {@link FetchOption}, this value is
     *          a hint to the persistence provider, and might be
     *          ignored.
     */
    BYPASS
}
```

`CacheStoreMode` enumerates the cache store modes recognized by this specification. The semantics of each mode is

defined by its Javadoc.

```
/**
 * Specifies how the {@link EntityManager} interacts with the
 * second-level cache when data is read from the database and
 * when data is written to the database.
 * <ul>
 * <li>{@link #USE} indicates that data may be written to the
 *     second-level cache.
 * <li>{@link #BYPASS} indicates that data must not be written
 *     to the second-level cache.
 * <li>{@link #REFRESH} indicates that data must be written
 *     to the second-level cache, even when the data is already
 *     cached.
 * </ul>
 *
 * <p>Enumerates legal values of the property
 * {@code jakarta.persistence.cache.storeMode}.
 *
 * @see EntityHandler#setCacheStoreMode(CacheStoreMode)
 * @see TypedQuery#setCacheStoreMode(CacheStoreMode)
 *
 * @since 2.0
 */
public enum CacheStoreMode
        implements FindOption, RefreshOption, FetchOption, TypedQuery.Option,
                   EntityManager.Option, EntityAgent.Option {

    /**
     * Specifies that entity data may be inserted into the
     * second-level cache when read from the database, and inserted
     * or updated in the second-level cache when written to the
     * database. The persistence provider is not required to refresh
     * already-cached items when reading from the database.
     * <p>
     * This is the default mode.
     */
    USE,

    /**
     * Specifies that entity data must never be inserted into the
     * second-level cache when read from the database, nor when
     * written to the database. The persistence provider is
     * permitted to invalidate cached items when writing to the
     * database.
     *
     * @apiNote When used as a {@link FetchOption}, this value is
     *          a hint to the persistence provider, and might be
     *          ignored.
     */
    BYPASS,

    /**
     * Specifies that entity data must be inserted or updated in
     * the second-level cache when read from the database or when
     * written to the database. The persistence provider is
     * required to refresh already-cached items when reading from
     * the database.
     *
     * @apiNote When used as a {@link FetchOption}, this value is
     *          a hint to the persistence provider, and might be
     *          ignored.
     */
    REFRESH
}
```

3.14.3. Cache Interface

The `Cache` interface found in Section B.8 allows the application to request eviction of entity data from the second-level cache directly and immediately, outside the scope of any persistence context.

3.15. Query APIs

The `Query`, `Statement`, and `TypedQuery` interfaces are used to control parameter binding and execution for both:

- static statements and queries declared in metadata, and
- dynamic statements and queries passed as strings to methods of `EntityHandler`.

The `TypedQuery` interface also allows pagination control and control over locking.

For example:

```
public List<Customer> findWithName(String name) {
    return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .ofType(Customer.class)
        .setParameter("custName", name)
        .setMaxResults(10)
        .getResultList();
}
```

The `StoredProcedureQuery` interface is used to call stored procedures defined in the database.

These interfaces may be found in Appendix B.

3.15.1. Query Execution

A query or statement, whether expressed in the Jakarta Persistence query language, in native SQL, or via the Criteria API, is executed via the `TypedQuery` or `Statement` interface, or, for backward compatibility, via deprecated methods of the `Query` interface.

- A `SELECT` query specified in the Jakarta Persistence query language or using the Criteria API (as an instance of `CriteriaSelect`) is executed by calling `getResultList()`, `getResultStream()`, `getSingleResult()`, or `getSingleResultOrNull()` of `TypedQuery`, or, for backward compatibility, by calling the same methods of `Query`.
- Similarly, any native SQL query that returns a result set is executed by calling `getResultList()`, `getResultStream()`, `getSingleResult()`, or `getSingleResultOrNull()`.
- An `UPDATE` or `DELETE` statement specified in the Jakarta Persistence query language or using the Criteria API (as an instance of `CriteriaUpdate` or `CriteriaDelete`) is executed by calling the method `execute()` of `Statement` or, for backward compatibility by calling the method `executeUpdate()` of `Query`.
- Similarly, any native SQL statement that returns a row count is executed by calling `execute()` or `executeUpdate()`.

Every query executed using `getResultList()`, `getResultStream()`, `getSingleResult()`, or `getSingleResultOrNull()` has a well-defined result type.

- For an instance of `TypedQuery` that represents a criteria query, the result type is determined by the type specified when the `CriteriaQuery` object was created, as described in Section 6.3.1.
- For an instance of `TypedQuery` that represents a `SELECT` query written in the Jakarta Persistence query language, the result type is determined by the second argument to the `createQuery()` or `createNamedQuery()` method, which must be the Java `Class` representing the result type or an `EntityGraph` whose root entity type is the result type. In either case, the specified result type must be compatible with the select list of the query, as specified in Section 3.15.10.
- For an instance of `TypedQuery` that represents a native SQL query that returns a result set, the result type is

determined by the second argument to the `createNativeQuery()` or `createNamedQuery()` method, which must be the Java `Class` representing the result type or a `ResultSetMapping` whose `type` member represents the result type. The specified result type must be compatible with the result set actually returned when the native SQL query is executed, as specified in Section 3.15.13 and Section 3.15.14.

- For an instance of `Query` which is not a `TypedQuery`, but which represents a Jakarta Persistence `SELECT` query, the result type depends on whether there is more than one item in the select list of the query. If there is more than one item in the select list, the result type is `Object[]`. Otherwise, if there is just one item in the select list, the query result type is just the type of that item.
- For an instance of `Query` which is not a `TypedQuery`, but which represents a native SQL query that returns a result set, the result type depends on whether there is more than one item in the result set mapping, if given, or, if there is no result set mapping given, on whether there is more than one column in the result set itself. If there is more than one item or column, respectively, the result type is `Object[]`. Otherwise, if there is exactly one item or column, the query result type is just `Object`.

For a native SQL query that returns a result set, the SQL result set mapping determines the values (entities, scalar values, constructor results) returned in each query result, as specified in Section 3.15.13. If no result set mapping is provided, the default type mappings defined by the JDBC specification are used to map columns of the result set to unmanaged Java types.

The semantics of the methods

```
public <T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass)

public <T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass)
```

may be extended in a future release of this specification to support other result types. Use of other result types, including `Tuple.class`, is not portable.

A stored procedure query that returns at least one result set can be executed by calling one of the `getResultList()`, `getSingleResult()`, `getSingleResultOrNull()`, or `execute()` methods of `StoredProcedureQuery`. A stored procedure that does not return a result set can be executed using the `execute()` or `executeUpdate()` method. Stored procedure query execution is described in detail in Section 3.15.15.3.

An `IllegalArgumentException` is thrown if a parameter instance is specified that does not correspond to a parameter of the query, if a parameter name is specified that does not correspond to a named parameter of the query, if a positional value is specified that does not correspond to a positional parameter of the query, or if the type of the parameter is not valid for the query. This exception may be thrown when the parameter is bound, or the execution of the query may fail. See Section 3.15.6, Section 3.15.7, and Section 3.15.8 for supported parameter usage.

The effect of applying `setMaxResults` or `setFirstResult` to a query involving fetch joins over collections is undefined. The use of `setMaxResults` and `setFirstResult` is not supported for stored procedure queries.

3.15.2. Query Execution and Transactions

Except for `executeUpdate()`, methods of `Query` and `TypedQuery` may be invoked without an active transaction context unless a lock mode other than `LockModeType.NONE` is requested, as specified below in Section 3.15.4. In particular, if no lock is requested, the methods `getResultList()`, `getSingleResult()`, and `getSingleResultOrNull()` may be called when no transaction is active:

- if the `Query` or `TypedQuery` object was obtained from an `EntityAgent` or from an `EntityManager` with a transaction-scoped persistence context, as defined in Section 3.7.1, the entities returned by the query are detached;

- otherwise, if the query object was obtained from an `EntityManager` with an extended or application-managed persistence context, the returned entities are managed.

The `execute()` method of `Statement` must not be called without an active transaction context. The `executeUpdate()` method of `Query` or `TypedQuery` must not be called without an active transaction context.

Whether a `StoredProcedureQuery` can be invoked without an active transaction context is determined by the transactional semantics and the requirements of the stored procedure implementation and the database. In particular, problems may occur if the stored procedure initiates a transaction and a transaction is already active. The state of entities returned by a stored procedure query invocation is determined according to the rules for `Query` or `TypedQuery` as described above.

A query object obtained from an `EntityManager` has an associated persistence context and returns entities belonging to that persistence context. If the persistence context is joined to the transaction, any runtime exception thrown by any method of `Query`, `Statement`, `TypedQuery`, or `StoredProcedureQuery` causes the current transaction to be marked for rollback, unless the type of the exception is explicitly excepted below in Section 3.16 or unless the method is explicitly excepted by the rules given below.

A query object obtained from an `EntityAgent` has no associated persistence context, though the persistence provider is permitted to canonicalize the entity instances returned by the query according to their persistent identities. A runtime exception thrown by a method of a query object obtained from an `EntityAgent` is considered recoverable and does not usually cause the current transaction to be marked for rollback.

On any database platform on which a query timeout causes an immediate transaction rollback, the persistence provider must throw a `PersistenceException` instead of a `QueryTimeoutException`.

Certain methods never cause the current transaction to be marked for rollback:

- a runtime exception thrown by the following methods of the `Query`, `Statement`, `TypedQuery`, and `StoredProcedureQuery` interfaces does not cause the current transaction to be marked for rollback: `getLockMode`, `getParameters`, `getParameter`, `getParameterValue`, `getOutputParameterValue`, and
- a runtime exception thrown by a method of the `Tuple`, `TupleElement`, or `Parameter` interface does not cause the current transaction to be marked for rollback.

3.15.3. Queries and the Flush Mode

The *flush mode* setting introduced above in Section 3.6.6 may affect the result of a query. The possible flush modes are enumerated by `FlushModeType` and by `QueryFlushMode`.

The `FlushModeType` specified via the method `setFlushMode()` of `EntityManager` controls the behavior for a persistence context. Its effect is limited to the persistence context in which `setFlushMode()` is called.

```
 * @see SynchronizationType
 *
 * @since 1.0
 */
public enum FlushModeType implements EntityManager.Option {
    /**
     * Every flush is an explicit operation requested by the
     * application program by calling {@link EntityManager#flush()}.
     * The persistence context is never automatically flushed.
     * Pending modifications to entities held in the persistence
     * context might not be visible to the processing of queries
     * and might never be made persistent.
     *
     * @since 4.0
     */
```

```
    EXPLICIT,

    /**
     * Pending modifications to entities associated with a
     * persistence context joined to the current transaction are
     * automatically flushed to the database when the transaction
     * commits. The provider is permitted to flush at other times,
     * but is not required to. Pending modifications to entities
     * associated with the persistence context might not be visible
     * to the processing of queries.
     */
    COMMIT,

    /**
     * Pending modifications to entities associated with a
     * persistence context joined to the current transaction are
     * automatically flushed to the database when the transaction
     * commits. In addition, the persistence provider must ensure
     * that every modification to the state of every entity
     * associated with the persistence context which could possibly
     * affect the result of a query is visible to the processing
     * of the query. The persistence provider implementation might
     * guarantee this by flushing pending modifications to the
     * database before executing the query. The provider is
     * permitted to flush at other times, but is not required to.
     * <p>
     * This is the default flush mode for a newly created
     * {@link EntityManager}.
     */
    AUTO
}
```

The `QueryFlushMode` specified via the method `setQueryFlushMode()` of `Query`, `Statement`, `TypedQuery`, or `StoredProcedureQuery` allows the behavior to be overridden during execution of a given statement or query. Its effect is limited to executions of the instance of `Query` on which `setQueryFlushMode()` was invoked.

```
 *     provider is required to flush modifications to the database
 *     before executing the query.
 * <li>If {@link QueryFlushMode#NO_FLUSH} is set, the persistence
 *     provider is not required to flush modifications to the
 *     database before executing the query; if, in addition, the
 *     current flush mode of the entity manager is
 *     {@link FlushModeType#EXPLICIT}, the persistence provider
 *     is not permitted to flush modifications to the database.
 * </ul>
 * Otherwise, if {@link QueryFlushMode#DEFAULT} is set:
 * <ul>
 * <li>If {@link FlushModeType#AUTO} is in effect, the provider
 *     must ensure that every modification to the state of every
 *     entity associated with the persistence context which could
 *     possibly affect the result of the query is visible to the
 *     processing of the query. The provider implementation might
 *     guarantee this by flushing pending updates to modified
 *     entities to the database before executing the query.
 * <li>If {@link FlushModeType#EXPLICIT} is in effect, the
 *     provider is not permitted to flush modifications to the
 *     database before executing the query. Pending modifications
 *     to entities in the persistence context might not be visible
 *     to the processing of the query.
 * <li>If {@link FlushModeType#COMMIT} is in effect, it is
 *     unspecified how pending modifications to entities in the
 *     persistence context affect the results of the query.
 * </ul>
 *
 * <p>When there is no transaction active, or if the persistence
```

```
 * context has not been joined to the current transaction, the
 * persistence provider must not flush pending modifications to
 * the database, regardless of the flush mode.
 *
 * <p>An {@link EntityAgent} has no associated persistence context,
 * and so flush modes have no effect on an entity agent, nor on
 * {@code Query} objects created by an entity agent.
 *
 * @since 4.0
 *
 * @see Query#setQueryFlushMode(QueryFlushMode)
 * @see NamedQuery#flush
 * @see NamedNativeQuery#flush
 * @see jakarta.persistence.query.QueryOptions#flush
 * @see FlushModeType
 */
public enum QueryFlushMode
        implements TypedQuery.Option, Statement.Option,
                   StoredProcedureQuery.Option {
    /**
     * The persistence provider is required to flush pending
     * modifications to the database before executing the query.
     * Pending modifications to entities held in the persistence
     * context are always visible to the processing of the query.
     * The {@linkplain EntityManager#getFlushMode current flush
     * mode} of the persistence context is completely overridden.
     */
    FLUSH,

    /**
     * The persistence provider is not required to flush pending
     * modifications to the database before executing the query.
     * Pending modifications to entities held in the persistence
     * context might not be visible to the processing of the query.
     * <ul>
     * <li>If the {@linkplain EntityManager#getFlushMode current
     *     flush mode} of the persistence context is
     *     {@link FlushModeType#EXPLICIT}, the persistence provider
     *     is not permitted to flush pending modifications before
     *     executing the query.
     * <li>Otherwise, this mode is considered a hint to the
     *     persistence provider.
     * </ul>
     */
    NO_FLUSH,

    /**
     * The persistence provider must determine whether to flush
     * before executing the query by considering the current
     * {@linkplain EntityManager#getFlushMode flush mode} of the
     * persistence context.
     */
    DEFAULT
}
```

When a Jakarta Persistence or native SQL query is executed within a transaction via an instance of `Query` obtained from an `EntityManager` joined to the transaction, the effective flush mode is determined by the current flush mode of the `Query` object, if any flush mode has been specified by calling `setFlushMode()` or `setQueryFlushMode()`, or, otherwise, by the current flush mode of the persistence context.

- If `QueryFlushMode.FLUSH` in set, the persistence provider is required to flush modifications to the database before executing the query.
- If `QueryFlushMode.NO_FLUSH` is set, the persistence provider is not required to flush modifications to the database before executing the query; if, in addition, the current flush mode of the entity manager is `FlushModeType.EXPLICIT`, the

persistence provider is not permitted to flush modifications to the database.

Otherwise, if `QueryFlushMode.DEFAULT` is set:

- If `FlushModeType.AUTO` is in effect, the persistence provider must ensure that every modification to the state of every entity associated with the persistence context which could possibly affect the result of the query is visible to the processing of the query. The persistence provider implementation might guarantee this by flushing pending updates to modified entities to the database before executing the query.
- If `FlushModeType.EXPLICIT` is in effect, the persistence provider is not permitted to flush modifications to the database before executing the query. Pending modifications to entities in the persistence context might not be visible to the processing of the query.
- If `FlushModeType.COMMIT` is in effect, it is unspecified how pending modifications to entities in the persistence context affect the results of the query.

When there is no transaction active, or if the persistence context has not been joined to the current transaction, the persistence provider must not flush pending modifications to the database, regardless of the flush mode.

An `EntityAgent` has no associated persistence context, and so flush modes have no effect on an entity agent, nor on `Query` objects created by an entity agent.

3.15.4. Queries and the Lock Mode

Execution of a JPQL query obtains a lock on queried records if the application requests such a lock by:

- calling `setLockMode()`, or
- specifying the `lockMode` member of the `@QueryOptions` or `@NamedQuery` annotation.

If a query is executed when no transaction is active, then the specified lock mode must be `LockModeType.NONE`. If there is no active transaction, or if the entity manager has not been joined to the transaction, and any lock mode other than `LockModeType.NONE` is requested, the method must throw `jakarta.persistence.TransactionRequiredException`.

A lock is obtained for each entity specified in the query result (including entities passed to constructors in the query `SELECT` clause).[24] The scope of a pessimistic lock, as defined above in Section 3.9.3, is controlled by:

- calling `setLockScope()`, or
- specifying the `lockScope` member of the `@QueryOptions` or `@NamedQuery` annotation.

If the lock mode type is `PESSIMISTIC_READ`, `PESSIMISTIC_WRITE`, or `PESSIMISTIC_FORCE_INCREMENT`, and the query returns scalar data (e.g., the values of entity field or properties, including scalar data passed to constructors in the query `SELECT` clause), the underlying database rows will be locked[25], but the version columns (if any) for any entities corresponding to such scalar data are not updated unless the entities themselves are also otherwise retrieved and updated.

If the lock mode type is `OPTIMISTIC` or `OPTIMISTIC_FORCE_INCREMENT`, and the query returns scalar data, any entities returned by the query will be locked, but no locking will occur for scalar data that does not correspond to the state of any entity instance in the query result.

Locking is supported for JPQL queries and criteria queries. If the `setLockMode()` or `getLockMode()` method is called on a query object representing a native SQL query or stored procedure, the persistence provider is permitted to throw `IllegalStateException`. Alternatively, execution of the query might fail.

3.15.5. Query Hints and Options

A *query hint* is a named value which may be:

- passed to the `setHint()` method declared by the `Query` interface found in Appendix B, or
- specified via the `hints` member of any one of the annotations which declares a named query or static query, as specified in Section 10.4.

A query hint specified in annotations may be overridden by calling the `setHint()` method of the `Query` interface.

This standard query hint is defined by this specification:

```
jakarta.persistence.query.timeout // time in milliseconds
```

This query hint may additionally be:

- passed as a property to the `createEntityManagerFactory()` method of the `Persistence` class, as defined in Section 9.7, or
- used in the `properties` element of the `persistence.xml` file, as defined in Section 8.2.1.12.

The timeout specified by calling the `createEntityManagerFactory()` method, via the `persistence.xml` file, or in annotations, serves as a default value which can be selectively overridden by calling the `setHint()` method of the `Query` interface.

Portable applications should not rely on query hints. Depending on the persistence provider and database in use, the hint may or may not be observed.

A persistence provider is permitted to support the use of additional, vendor-specific hints. Vendor-specific hints must not have names in the `jakarta.persistence` namespace. A vendor-specific hint must be ignored if it is not understood by the provider.

Alternatively, a *query option* is a type safe object carrying the same sort of information which might be provided in a query hint. The interfaces `Statement`, `TypedQuery`, and `StoredProcedureQuery` define inner `Option` interfaces, and this specification provides a number of built-in concrete option types implementing these option interfaces. Providers are encouraged to define additional custom option types controlling vendor-specific behavior.

> Custom query options are type safe and result in more elegant code. However, their use results in a dependency on vendor-specific APIs. Application programs desiring portability between persistence providers might choose to use string-based query hints in preference to custom query options.

3.15.6. Parameter Objects

`Parameter` objects can be used for criteria queries and for Jakarta Persistence query language queries.

Implementations may support the use of `Parameter` objects for native queries, however support for `Parameter` objects with native queries is not required by this specification. The use of `Parameter` objects for native queries is not portable. The mixing of parameter objects with named or positional parameters is invalid.

A portable application should not attempt to reuse a `Parameter` object obtained from one instance of `Query` by passing it to a method of a different instance of `Query`.

3.15.7. Named Parameters

Named parameters can be used for Jakarta Persistence query language queries, for criteria queries (although use of `Parameter` objects is to be preferred), and for stored procedure queries that support named parameters.

Named parameters follow the rules for identifiers defined in Section 4.4.1. Named parameters are case-sensitive. The mixing of named and positional parameters is invalid.

A named parameter of a JPQL query is an identifier prefixed by the `:` symbol. Parameter names passed as arguments to

the `setParameter` methods of the `Query` interface do not include this prefix.

3.15.8. Positional Parameters

Only positional parameter binding and positional access to result items may be portably used for native queries, except for stored procedure queries for which named parameters have been defined. When binding the values of positional parameters, the numbering starts at `1`. It is assumed that for native queries the parameters themselves use the standard JDBC syntax: `?` instead of `?1`.

The use of positional parameters is not supported for criteria queries.

3.15.9. Arguments to Query Parameters

Arguments are assigned to query parameters by calling `Query.setParameter()`. The first parameter of `setParameter()` identifies the named or positional parameter of the query.

An argument may be assigned to a single-valued parameter of a JPQL or native SQL query by passing the argument to the second parameter of `setParameter()`.

```
query.setParameter("name", name)
```

A list of arguments may be assigned to a collection-valued parameter of a JPQL query by packaging the arguments in a non-null instance of `java.util.List` and passing the list as an argument to the second parameter of `setParameter()`. The list should contain at least one element. If the list is empty the behavior is undefined. Portable applications should not pass an empty list to a collection-valued parameter.

```
query.setParameter("names", List.of(name1, name2, name3))
```

3.15.10. Result Classes for Jakarta Persistence Queries

The `createQuery()` method of `EntityHandler` accepts, among other possibilities, a JPQL query string and a Java class object specifying a *result class*.

```
<T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass);
```

The `TypedQuery` created by this method returns instances of the given result class. Either:

- the select list of the query contains only a single item, which must be assignable to the given result class,
- the result class is `Object[].class` or `Tuple.class`, or
- the result class is a non-`abstract` class or record type with a constructor with the same number of parameters as the query has items in its select list, and the constructor parameter types exactly match the types of the corresponding items in the select list.

In the first case, each query result is returned directly to the caller. In the second case, each query result is packaged in an array with the array elements corresponding by position with the items of the query select list or in an instance of `Tuple` with the indices and aliases of the tuple elements determined by the positions and aliases of the corresponding items in the select list. In the third case, each query result is automatically packaged in a new instance of the result class by calling the matching constructor.

3.15.11. Named Queries and Named Statements

A *named query* is:

- a static query expressed in metadata using a `@NamedQuery` or `@NamedNativeQuery` annotation, as defined in Chapter 10, a `@JakartaQuery` or `@NativeQuery` annotation, as defined in Appendix F, or using a `<named-query>` or `<named-native-query>` XML element, as specified in Section 12.2.2.7, or
- a query instantiated at runtime and registered by calling the `addNamedQuery()` method of `EntityManagerFactory`.

A named query may be expressed in the Jakarta Persistence query language or in SQL. Query names are scoped to the persistence unit and must be unique within a given unit. A named query should be a JPQL `select` query or a native SQL statement that returns a result set.

> For backward compatibility, a named query is permitted to be a JPQL `update` or `delete` statement or a SQL statement that returns a row count, but this usage is now discouraged.

Similarly, a *named statement* is:

- a static statement expressed in metadata using a `@NamedStatement` or `@NamedNativeStatement` annotation, as defined in Chapter 10, a `@JakartaQuery` or `@NativeQuery` annotation, as defined in Appendix F, or using a `<named-statement>` or `<named-native-statement>` XML element, as specified in Section 12.2.2.8, or
- a statement instantiated at runtime and registered by calling the `addNamedStatement()` method of `EntityManagerFactory`.

A named statement may be expressed in the Jakarta Persistence query language or in SQL. Statement names are scoped to the persistence unit and must be unique within a given unit. A named statement must be a JPQL `update` or `delete` statement or a native SQL statement that returns a row count.

A named query or named statement may be declared by annotating any class or interface belonging to the persistence unit, as specified in Section 8.2.1.8.

The following named query is written in JPQL and statically declared using an annotation:

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
@Entity class Customer { ... }
```

The following is an example of the use of this named query:

```
@PersistenceContext
public EntityManager em;
    // ...

    customers = em.createNamedQuery("findAllCustomersWithName")
        .setParameter("custName", "Smith")
        .getResultList();
```

Named queries and statements expressed in annotations have representations in the canonical static metamodel, as specified in Section 5.2.

```
@PersistenceContext
public EntityManager em;
    // ...

    customers = em.createQuery(Customer_._findAllCustomersWithName_)
        .setParameter("custName", "Smith")
        .getResultList();
```

3.15.12. Polymorphic Queries

By default, all queries are polymorphic. That is, the `FROM` clause of a query designates not only instances of the specific entity class(es) to which it explicitly refers, but subclasses as well. The instances returned by a query include instances of the subclasses that satisfy the query conditions.

For example, the following query returns the average salary of all employees, including subtypes of `Employee`, such as `Manager` and `Exempt`.

```
select avg(e.salary) from Employee e where e.salary > 80000
```

Entity type expressions, described in Section 4.7.12, as well as the use of downcasting, described in Section 4.4.9, can be used to restrict query polymorphism.

3.15.13. SQL Queries

Queries may be expressed in native SQL. The result of a native SQL query may consist of entities, unmanaged instances created via constructors, scalar values, or some combination of these.

> The SQL query facility is intended to provide support for those cases where it is necessary to use the native SQL of the target database in use (and/or where the Jakarta Persistence query language cannot be used). Native SQL queries are not expected to be portable across databases.

3.15.13.1. Returning Managed Entities from Native Queries

The persistence provider is responsible for performing the mapping between the values returned by the SQL query and entity attributes in accordance with the object/relational mapping metadata for the entity or entities. In particular, the names of the columns in the SQL result are used to map to the entity attributes as defined by this metadata. This mapping includes the mapping of the attributes of any embeddable classes that are part of the non-collection-valued entity state and attributes corresponding to foreign keys contained as part of the entity state[26].

When an entity is to be returned from a native query, the SQL statement should select all of the columns that are mapped to the entity object. This should include foreign key columns to related entities. The results obtained when insufficient data is available are undefined.

In the simplest case—i.e., when the results of the query are limited to entities of a single entity class and the mapping information can be derived from the columns of the SQL result and the object/relational mapping metadata—it is sufficient to specify only the expected class of the entity result.

The following example illustrates the case where a native SQL query is created dynamically using the `createNativeQuery` method and the entity class that specifies the type of the result is passed in as an argument.

```
Query q = em.createNativeQuery(
    "SELECT o.id, o.quantity, o.item " +
        "FROM Order o, Item i " +
        "WHERE (o.item = i.id) AND (i.name = 'widget')",
    com.acme.Order.class);
```

When executed, this query will return a collection of all `Order` entities for items named “widget”.

The `SqlResultSetMapping` metadata annotation—which is designed to handle more complex cases—can be used as an alternative here. See Section 10.4.6 for the definition of the `SqlResultSetMapping` metadata annotation and related annotations.

For the query shown above, the `SqlResultSetMapping` metadata for the query result type might be specified as follows:

```
@SqlResultSetMapping(
    name="WidgetOrderResults",
    entities=@EntityResult(entityClass=com.acme.Order.class))
```

The same results as produced by the query above can then obtained by the following:

```
Query q = em.createNativeQuery(
    "SELECT o.id, o.quantity, o.item " +
        "FROM Order o, Item i " +
        "WHERE (o.item = i.id) AND (i.name = 'widget')",
    "WidgetOrderResults");
```

When multiple entities are returned by a SQL query or when the column names of the SQL result do not correspond to those of the object/relational mapping metadata, a `SqlResultSetMapping` metadata definition must be provided to specify the entity mapping.

The following query and `SqlResultSetMapping` metadata illustrates the return of multiple entity types. It assumes default metadata and column name defaults.

```
Query q = em.createNativeQuery(
    "SELECT o.id, o.quantity, o.item, i.id, i.name, i.description " +
        "FROM Order o, Item i " +
        "WHERE (o.quantity > 25) AND (o.item = i.id)",
    "OrderItemResults");

@SqlResultSetMapping(name="OrderItemResults", entities={
    @EntityResult(entityClass=com.acme.Order.class),
    @EntityResult(entityClass=com.acme.Item.class)
})
```

When the column names of the SQL result do not correspond to those of the object/relational mapping metadata or introduce a conflict in mapping column defaults as in the example code above, more explicit SQL result mapping metadata must be provided to enable the persistence provider runtime to map the JDBC results into the expected objects. This might arise, for example, when column aliases must be used in the SQL SELECT clause when the SQL result would otherwise contain multiple columns of the same name or when columns in the SQL result are the results of operators or functions. The `FieldResult` annotation element within the `EntityResult` annotation is used to specify the mapping of such columns to entity attributes.

The following example combining multiple entity types includes aliases in the SQL statement. This requires that the column names be explicitly mapped to the entity fields corresponding to those columns. The `FieldResult` annotation is used for this purpose.

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item AS order_item, " +
        "i.id, i.name, i.description " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item = i.id)",
    "OrderItemResults");

@SqlResultSetMapping(name="OrderItemResults", entities={
    @EntityResult(entityClass=com.acme.Order.class, fields={
        @FieldResult(name="id", column="order_id"),
        @FieldResult(name="quantity", column="order_quantity"),
        @FieldResult(name="item", column="order_item")}),
    @EntityResult(entityClass=com.acme.Item.class)
```

```
})
```

When the returned entity type contains an embeddable class, the `FieldResult` element must use a dot (“ . ”) notation to indicate which column maps to which field or property of the contained embeddable.

Example:

```
Query q = em.createNativeQuery(
    "SELECT c.id AS customer_id, " +
        "c.street AS customer_street, " +
        "c.city AS customer_city, " +
        "c.state AS customer_state, " +
        "c.status AS customer_status " +
        "FROM Customer c " +
        "WHERE c.status = 'GOLD' ",
    "CustomerResults");

@SqlResultSetMapping(name="CustomerResults", entities={
    @EntityResult(entityClass=com.acme.Customer.class, fields={
        @FieldResult(name="id", column="customer_id"),
        @FieldResult(name="address.street", column="customer_street"),
        @FieldResult(name="address.city", column="customer_city"),
        @FieldResult(name="address.state", column="customer_state"),
        @FieldResult(name="status", column="customer_status")
    })
})
```

When the returned entity type is the owner of a single-valued relationship and the foreign key is a composite foreign key (composed of multiple columns), a `FieldResult` element should be used for each of the foreign key columns. The `FieldResult` element must use the dot (“ . ”) notation form to indicate the column that maps to each property or field of the target entity primary key.

If the target entity has a primary key of type `IdClass`, this specification takes the form of the name of the field or property for the relationship, followed by a dot (“ . ”), followed by the name of the field or property of the primary key in the target entity. The latter will be annotated with `Id`, as specified in Section 11.2.25.

Example:

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item_id AS order_item_id, " +
        "o.item_name AS order_item_name, " +
        "i.id, i.name, i.description " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item_id = i.id) " +
        "AND (order_item_name = i.name)",
    "OrderItemResults");

@SqlResultSetMapping(name="OrderItemResults", entities={
    @EntityResult(entityClass=com.acme.Order.class, fields={
        @FieldResult(name="id", column="order_id"),
        @FieldResult(name="quantity", column="order_quantity"),
        @FieldResult(name="item.id", column="order_item_id")}),
        @FieldResult(name="item.name", column="order_item_name")}),
    @EntityResult(entityClass=com.acme.Item.class)
})
```

If the target entity has a primary key of type `EmbeddedId`, this specification is composed of the name of the field or property for the relationship, followed by a dot (“ . ”), followed by the name or the field or property of the primary key (i.e., the name of the field or property annotated as `EmbeddedId`), followed by the name of the corresponding field or

property of the embedded primary key class.

Example:

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item_id AS order_item_id, " +
        "o.item_name AS order_item_name, " +
        "i.id, i.name, i.description " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item_id = i.id) AND (order_item_name = i.name)",
    "OrderItemResults");

@SqlResultSetMapping(name="OrderItemResults", entities={
    @EntityResult(entityClass=com.acme.Order.class, fields={
        @FieldResult(name="id", column="order_id"),
        @FieldResult(name="quantity", column="order_quantity"),
        @FieldResult(name="item.itemPk.id", column="order_item_id")}),
        @FieldResult(name="item.itemPk.name", column="order_item_name")}),
    @EntityResult(entityClass=com.acme.Item.class)
})
```

The `FieldResult` elements for the composite foreign key are combined to form the primary key `EmbeddedId` class for the target entity. This may then be used to subsequently retrieve the entity if the relationship is to be eagerly loaded.

The dot-notation form is not required to be supported for any usage other than for embeddables, composite foreign keys, or composite primary keys.

3.15.13.2. Returning Unmanaged Instances

Instances of other classes (including non-managed entity instances) as well as scalar results can be returned by a native query. These can be used singly, or in combination, including with entity results.

Scalar Results

Scalar results can be included in the query result by specifying the `ColumnResult` annotation element of the `SqlResultSetMapping` annotation. The intended type of the result can be specified using the `type` element of the `ColumnResult` annotation. The specified type must be one of the basic types listed in Section 2.6.

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item AS order_item, " +
        "i.name AS item_name, " +
        "i.availabilityDate AS item_shipdate " +
        "FROM Order o, Item i " +
        "WHERE (order_quantity > 25) AND (order_item = i.id)",
    "OrderResults");

@SqlResultSetMapping(
    name="OrderResults",
    entities={
        @EntityResult(entityClass=com.acme.Order.class, fields={
            @FieldResult(name="id", column="order_id"),
            @FieldResult(name="quantity", column="order_quantity"),
            @FieldResult(name="item", column="order_item")}
        )},
    columns={
        @ColumnResult(name="item_name"),
        @ColumnResult(name="item_shipdate", type=java.util.Date.class)
```

```
})
```

Constructor Results

The mapping to constructors is specified using the `ConstructorResult` annotation element of the `SqlResultSetMapping` annotation. The `targetClass` element of the `ConstructorResult` annotation specifies the class whose constructor corresponds to the specified result mappings. If the `targetClass` is an entity class, any entities returned as constructor results are in either the new or the detached state, depending on whether a primary key is retrieved for the constructed object.

- Constructor arguments mapped directly to column results must be specified using the `columns` element of the `ConstructorResult` annotation in the same order as the corresponding arguments of the constructor, before constructor arguments mapped to entity results.
- Constructor arguments mapped to entity results must be specified using the `entities` element of the `ConstructorResult` annotation, again in the same order as the corresponding arguments of the constructor.

Example:

```
Query q = em.createNativeQuery(
    "SELECT c.id, c.name, COUNT(o) as orderCount, AVG(o.price) AS avgOrder " +
        "FROM Customer c, Orders o " +
        "WHERE o.cid = c.id " +
        "GROUP BY c.id, c.name",
    "CustomerDetailsResult");

@SqlResultSetMapping(name="CustomerDetailsResult", classes={
    @ConstructorResult(targetClass=com.acme.CustomerDetails.class, columns={
        @ColumnResult(name="id"),
        @ColumnResult(name="name"),
        @ColumnResult(name="orderCount"),
        @ColumnResult(name="avgOrder", type=Double.class)})
})
```

3.15.13.3. Annotated Result Set Mappings

The `SqlResultSetMapping` annotation defined in Section 10.4.6 allows columns of a SQL result set to be explicitly mapped to an entity class, as described above in Section 3.15.13.1, a constructor, as described above in Section 3.15.13.2.2, or a basic type, as described in Section 3.15.13.2.1.

- When a `SqlResultSetMapping` specifies exactly one `EntityResult`, `ConstructorResult`, or `ColumnResult`, each row of the SQL result set maps to a query result whose type is the `entityClass` specified by the `EntityResult`, `targetClass` specified by the `ConstructorResult`, or `type` specified by (or inferred from) the `ColumnResult`.
- Otherwise, when a `SqlResultSetMapping` specifies more than one result mapping, each row in the SQL result set maps to a query result of type `Object[]`, whose elements are, in order:
  - entity results, in the order in which they are declared by the `entities` element;
  - constructor results, in the order in which they are declared by the `classes` element; and then
  - scalar results, in the order in which they are declared by the `columns` element.

If the SQL result set has a column with no result mapping explicitly specified by the `SqlResultSetMapping` annotation, the column is ignored.

3.15.13.4. Programmatic Result Set Mappings

As an alternative to the annotations specified above, the API found in Appendix G defines a programmatic way to

specify the mapping of a SQL result set.

In the programmatic API:

- `EntityMapping`, found in Section G.6, is an alternative to `@EntityResult`, and has the semantics specified above in Section 3.15.13.1,
- `FieldMapping`, found in Section G.5, is an alternative to `@FieldResult`,
- `ConstructorMapping`, found in Section G.9, is an alternative to `@ConstructorResult`, with semantics specified in Section 3.15.13.2.2,
- `TupleMapping`, found in Section G.10, is similar to `ConstructorMapping` but specifies that each query result should be packaged as an instance of `Tuple`, with the indexes and aliases of the tuple elements determined by the positions and aliases of the children of the `TupleMapping`,
- `ColumnMapping`, found in Section G.4 is an alternative to `@ColumnResult`, as specified in Section 3.15.13.2.1, and
- `EmbeddedMapping`, found in Section G.7 is an alternative to the use of dot notation to map embedded objects within an `EntityMapping`.

Instances of these objects are usually constructed using static factory methods of the `ResultSetMapping` interface. These methods accept instances of the metamodel defined in Chapter 5, which may be obtained from the static metamodel specified in Section 5.1 for type safety. Mapped column names are supplied as strings.

The following code constructs a mapping for the `Author` entity with an embedded field of type `Name`.

```
import static jakarta.persistence.sql.ResultSetMapping.*;
...
    var entityMapping =
            entity(Author.class,
                    field(Author_.ssn, "auth_ssn"),
                    embedded(Author_.name,
                            field(Name_.first, "auth_first_name"),
                            field(Name_.last, "auth_last_name")));
```

This code constructs a mapping to the constructor of the unmanaged `Summary` class with three parameters of type `String`.

```
import static jakarta.persistence.sql.ResultSetMapping.*;
...
    var constructorMapping =
            constructor(Summary.class,
                    column("isbn", String.class),
                    column("title", String.class),
                    column("author", String.class));
```

This example shows a mixed mapping to two entities, a scalar result, and the constructor of an unmanaged class.

```
import static jakarta.persistence.sql.ResultSetMapping.*;
...
    var compoundMapping =
            compound(
                    entity(Author.class),
                    entity(Book.class, field(Book_.isbn, "isbn")),
                    column("sales", BigDecimal.class),
                    constructor(Summary.class, column("isbn"), column("title"))
            );
```

A programmatically constructed result set mapping may be passed as argument to the second parameter of the `createNativeQuery()` method of `EntityManager` for execution.

An instance of `ResultSetMapping` representing a result set mapping declared using the `@SqlResultSetMapping` annotation

may be obtained from the `EntityManagerFactory` by calling `getResultSetMappings()`, passing the Java class if the query results as determined by the rules specified above in Section 3.15.13.3.

3.15.14. Result Classes for SQL Queries

The `createNativeQuery()` method of `EntityHandler` accepts, among other possibilities, a SQL query string and a Java class object specifying a *result class*.

```
<T> TypedQuery<T> createNativeQuery(String sqlString, Class<T> resultClass);
```

The `TypedQuery` created by this method returns instances of the given result class. Either:

- the result class is an entity class and is interpreted as a managed entity result with implicit field mappings determined by the names of the columns in the result set and the object/relational mapping of the entity, as specified above in Section 3.15.13.1,
- the result class is the class of a basic type listed in Section 2.6 and the result set must have a single column which is interpreted as a scalar result as specified in Section 3.15.13.2.1,
- the result class is a non-`abstract` class or record type with a constructor with the same number of parameters as the result set has columns, and is interpreted as a constructor result including all the columns of the result set as specified in Section 3.15.13.2.2, or
- the result class is `Object[].class` and each query result is packaged in an array of type `Object[]`, with the array elements corresponding by position with the columns of the select list and column values obtained according to the default type mappings defined by the JDBC specification.

3.15.14.1. Restrictions

When an entity is being returned, the SQL statement should select all of the columns that are mapped to the entity object. This should include foreign key columns to related entities. The results obtained when insufficient data is available are undefined. A SQL result set mapping must not be used to map results to the non-persistent state of an entity.

The use of named parameters is not defined for native SQL queries. Only positional parameter binding for SQL queries may be used by portable applications.

3.15.15. Stored Procedures

The `StoredProcedureQuery` interface supports the use of database stored procedures.

Stored procedures can be specified either by means of the `NamedStoredProcedureQuery` annotation or dynamically. Annotations for the specification of stored procedures are described in Section 10.4.5.

3.15.15.1. Named Stored Procedure Queries

Unlike in the case of a named native query, the `NamedStoredProcedureQuery` annotation names a stored procedure that exists in the database rather than providing a stored procedure definition. The `NamedStoredProcedureQuery` annotation specifies the types of all parameters to the stored procedure, their corresponding parameter modes (`IN`, `OUT`, `INOUT`, `REF_CURSOR`[27]), and how result sets, if any, are to be mapped. The name that is assigned to the stored procedure in the `NamedStoredProcedureQuery` annotation is passed as an argument to the `createNamedStoredProcedureQuery` method to create an executable `StoredProcedureQuery` object.

A stored procedure may return more than one result set. As with native queries, the mapping of result sets can be specified either in terms of a `resultClasses` or as a `resultSetMappings` annotation element. If there are multiple result sets,

it is assumed that they will be mapped using the same mechanism — e.g., all via a set of result class mappings or all via a set of result set mappings. The order of the specification of these mappings must be the same as the order in which the result sets will be returned by the stored procedure invocation. If the stored procedure returns one or more result sets and no `resultClasses` or `resultSetMappings` element has been specified, any result set will be returned as a list of type `Object[]`. The combining of different strategies for the mapping of stored procedure result sets is undefined.

Result set mappings and result classes are interpreted according to the rules defined above in Section 3.15.13 for native SQL queries.

`StoredProcedureParameter` metadata needs to be provided for all parameters. Parameters must be specified in the order in which they occur in the parameter list of the stored procedure. If parameter names are used, the parameter name is used to bind the parameter value and to extract the output value (if the parameter is an INOUT or OUT parameter). If parameter names are not specified, it is assumed that positional parameters are used. The mixing of named and positional parameters is invalid.

3.15.15.2. Dynamically-specified Stored Procedure Queries

If the stored procedure is not defined using metadata, parameter and result set information must be provided dynamically.

All parameters of a dynamically-specified stored procedure query must be registered using the `registerStoredProcedureParameter` method of the `StoredProcedureQuery` interface.

Result set mapping information can be provided by means of the `createStoredProcedureQuery` method.

3.15.15.3. Stored Procedure Query Execution

Stored procedure query execution can be controlled as described below.

The `setParameter` methods are used to set the values of all required IN and INOUT parameters. It is not required to set the values of stored procedure parameters for which default values have been defined by the stored procedure.

When `getResultList`, `getSingleResult`, and `getSingleResultOrNull` are called on a `StoredProcedureQuery` object, the persistence provider will call `execute` on an unexecuted stored procedure query before processing `getResultList`, `getSingleResult` or `getSingleResultOrNull`.

When `executeUpdate` is called on a `StoredProcedureQuery` object, the persistence provider will call `execute` on an unexecuted stored procedure query followed by `getUpdateCount`. The results of `executeUpdate` will be those of `getUpdateCount`.

The `execute` method supports both the simple case where scalar results are passed back only via INOUT and OUT parameters as well as the most general case (multiple result sets and/or update counts, possibly also in combination with output parameter values).

The `execute` method returns `true` if the first result is a result set, and `false` if it is an update count or there are no results other than through INOUT and OUT parameters, if any.

If the `execute` method returns `true`, the pending result set can be obtained by calling `getResultList`, `getSingleResult`, or `getSingleResultOrNull`. The `hasMoreResults` method can then be used to test for further results.

If `execute` or `hasMoreResults` returns `false`, the `getUpdateCount` method can be called to obtain the pending result if it is an update count. The `getUpdateCount` method will return either the update count (zero or greater) or -1 if there is no update count (i.e., either the next result is a result set or there is no next update count).

For portability, results that correspond to JDBC result sets and update counts need to be processed before the values of any INOUT or OUT parameters are extracted.

After results returned through `getResultList` and `getUpdateCount` have been exhausted, results returned through INOUT and OUT parameters can be retrieved.

The `getOutputParameterValue` methods are used to retrieve the values passed back from the procedure through INOUT and OUT parameters.

When using `REF_CURSOR` parameters for result sets, the update counts should be exhausted before calling `getResultList` to retrieve the result set. Alternatively, the `REF_CURSOR` result set can be retrieved through `getOutputParameterValue`. Result set mappings will be applied to results corresponding to `REF_CURSOR` parameters in the order the `REF_CURSOR` parameters were registered with the query.

In the simplest case, where results are returned only via INOUT and OUT parameters, `execute` can be followed immediately by calls to `getOutputParameterValue`.

3.16. Summary of Exceptions

The following exception types are defined by this specification:

PersistenceException

Thrown by the persistence provider when a problem occurs. Every other exception type defined by this specification is a subclass of `PersistenceException`.

Any instance of `PersistenceException` thrown by an `EntityManager` causes the current active transaction to be marked for rollback if the persistence context is joined to the transaction, unless the exception is also an instance of `NoResultException`, `NonUniqueResultException`, `LockTimeoutException`, or `QueryTimeoutException`.

TransactionRequiredException

Thrown by the persistence provider when a transaction is required but is not active.

RollbackException

Thrown by the persistence provider when a resource-local transaction cannot be committed, that is, when the `commit()` method of `EntityTransaction` fails.

OptimisticLockException

Thrown by the persistence provider when an optimistic locking conflict occurs. This exception may be thrown by an API call, during flush, or when the transaction commits.

If thrown by an `EntityManager` with a persistence context joined to an active transaction, the transaction is marked for rollback when this exception is thrown.

PessimisticLockException

Thrown by the persistence provider when a pessimistic locking conflict occurs. This exception may be thrown by an API call, during flush, or when the transaction commits.

If thrown by an `EntityManager` with a persistence context joined to an active transaction, the transaction is marked for rollback when this exception is thrown.

EntityExistsException

Thrown by the persistence provider when:

- a detached instance of an entity type is passed to `persist()`, or
- any kind of uniqueness constraint is violated when an entity is inserted in the database.

The persistence provider is always permitted to postpone an `EntityExistsException` until execution of the

resulting database insert operation. If a detached instance is passed to the persist operation, either:

- an `EntityExistsException` is immediately thrown by `persist()`, or
- an `EntityExistsException` or some other subtype of `PersistenceException` is thrown when the persistence context is flushed or when the transaction commits.

If thrown by an `EntityManager` with a persistence context joined to an active transaction, the transaction is marked for rollback when this exception is thrown.

EntityNotFoundException

Thrown by the persistence provider when:

- `get()` cannot return an instance of the requested entity type because there is no matching record in the database,
- an entity reference obtained by calling `getReference()` is accessed but the reference has no corresponding record in the database,
- an entity is passed to `refresh()` and its record no longer exists in the database,
- an entity is passed to `lock()`, a pessimistic lock mode is requested, and the record no longer exists in the database.

If thrown by an `EntityManager` with a persistence context joined to an active transaction, the transaction is marked for rollback when this exception is thrown.

NoResultException

Thrown by the persistence provider when `getSingleResult()` is executed and there is no result to return.

This exception type is considered recoverable. When this exception type is thrown, the current transaction is not automatically marked for rollback.

NonUniqueResultException

Thrown by the persistence provider when `getSingleResult()` is executed and there is more than one result from the query.

This exception type is considered recoverable. When this exception type is thrown, the current transaction is not automatically marked for rollback.

LockTimeoutException

Thrown by the persistence provider when a pessimistic locking conflict occurs that does not result in transaction rollback. This exception may be thrown by an API call, during flush, or when the transaction commits.[28]

This exception type is considered recoverable. When this exception type is thrown, the current transaction is not automatically marked for rollback.

QueryTimeoutException

Thrown by the persistence provider when a query times out and only the statement is rolled back.

This exception type is considered recoverable. When this exception type is thrown, the current transaction is not automatically marked for rollback.

In addition to the exception types listed above, certain methods of `EntityManager`, `EntityHandler`, `Query`, `Statement`, `TypedQuery`, and `StoredProdureQuery` throw `IllegalArgumentException` to indicate that an argument of the method is not an entity, is not an entity class, is not the name of a declared named query or entity graph, is not a legal JPQL query, and so on. Since Jakarta Persistence 4.0, such exceptions are now considered recoverable, and the persistence provider is not required to automatically mark the current transaction for rollback when a method of one of these interfaces throws

`IllegalArgumentException`.[29]

[1] Actually, when mapping a legacy database schema, it is often perfectly acceptable to map the primary key of an entity to any unique key of the primary table.

[2] Alternatively, the context is a map keyed by persistent identity.

[3] This is accomplished via indirection at the container level. Actually, every transaction-scoped persistence context is associated with a distinct implementation of `EntityManager` supplied by the persistence provider.

[4] If the parent is detached or new, or was previously removed before the orphan was associated with it, the remove operation is not applied to the entity being orphaned.

[5] This includes, for example, modifications to elements of a `char[]` or `byte[]` array.

[6] The `refresh()` method may be used to refresh entity state from the database, as described in Section 3.6.7.

[7] This might be an issue if unique constraints (such as those described for the default mappings in Section 2.12.3.1 and Section 2.12.5.1) were not applied in the definition of the object/relational mapping.

[8] When a new transaction begins, the managed objects belonging to an extended persistence context are not reloaded from the database.

[9] Applications may require that database isolation levels higher than read-committed be in effect. The configuration of the setting database isolation levels, however, is outside the scope of this specification.

[10] Such alternative mechanisms might be standardized by a future release of this specification.

[11] This includes owned relationships maintained in join tables.

[12] Typically, by incrementing the version number, or by replacing the previous timestamp with a timestamp representing the current time.

[13] Ideally, version verification and update happen in a single atomic operation against the datastore, for example, in a single SQL update statement.

[14] Implementations are permitted to use database mechanisms other than locking to achieve the semantic effects described here, for example, multiversion concurrency control mechanisms.

[15] This is achieved by using a lock with `LockModeType.PESSIMISTIC_WRITE` or `LockModeType.PESSIMISTIC_FORCE_INCREMENT` as described in Section 3.9.4.

[16] For example, a persistence provider may use an underlying database platform's SELECT FOR UPDATE statements to implement pessimistic locking if that construct provides appropriate semantics, or the provider may use an isolation level of repeatable read.

[17] The lock mode type NONE may be specified as a method argument and also provides a default value for annotations.

[18] Databases concurrency control mechanisms that provide comparable semantics, e.g., multiversion concurrency control, can be used by the provider.

[19] The persistence provider is not required to flush the entity to the database immediately.

[20] Note that this caution applies also to the actions of objects that might be injected into an entity listener

[21] Excluded listeners may be reintroduced on an entity class by listing them explicitly in the `EntityListeners` annotation or XML `entity-listeners` element.

[22] If a method overrides an inherited callback method but specifies a different lifecycle event or is not a lifecycle callback method, the overridden method is not invoked.

[23] For example, if a transaction commit occurs as a result of the normal termination of a session bean business method with transaction attribute `RequiresNew`, the `PostPersist` and `PostRemove` callbacks are executed in the naming context, the transaction context, and the security context of that component.

[24] Note that the setLockMode() method may be called more than once (with different values) on a `Query` or `TypedQuery` object.

[25] Note that locking does not occur for data passed to aggregate functions. Further, queries involving aggregates with pessimistic locking may not be supported on all database platforms.

[26] Support for joins is currently limited to single-valued relationships that are mapped directly—i.e., not via join tables.

[27] Note that `REF_CURSOR` parameters are used by some databases to return result sets from stored procedures.

[28] Typically this occurs because the database uses statement-level rollback when a pessimistic lock cannot be granted and there is no deadlock.

[29] There are many possible conditions leading to an `IllegalArgumentException` and some such conditions do lead to the transaction being marked for rollback.

Chapter 4. Query Language

The Jakarta Persistence query language is a string-based query language used to define queries over entities and their persistent state. It enables the application developer to specify the semantics of queries in a portable way, independent of the particular database schema in use in an enterprise environment. The full range of the language may be used in both static and dynamic queries.

This chapter provides the full definition of the Jakarta Persistence query language.

4.1. Overview

The Jakarta Persistence query language is a query specification language for string-based dynamic queries and static queries expressed through metadata. It is used to define queries over the persistent entities defined by this specification and their persistent state and relationships.

The Jakarta Persistence query language can be compiled to a target language, such as SQL, of a database or other persistent store. This allows the execution of queries to be shifted to the native language facilities provided by the database, instead of requiring queries to be executed on the runtime representation of the entity state. As a result, query methods can be optimizable as well as portable.

The query language uses the abstract persistence schema of entities, including their embedded objects and relationships, for its data model, and it defines operators and expressions based on this data model. It uses a SQL-like syntax to select objects or values based on abstract schema types and relationships. It is possible to parse and validate queries before entities are deployed.

> The term abstract persistence schema refers to the persistent schema abstraction (persistent entities, their state, and their relationships) over which Jakarta Persistence queries operate. Queries over this persistent schema abstraction are translated into queries that are executed over the database schema to which entities are mapped.

Queries may be defined in metadata annotations or the XML descriptor. The abstract schema types of a set of entities can be used in a query if the entities are defined in the same persistence unit as the query. Path expressions allow for navigation over relationships defined in the persistence unit.

> A persistence unit defines the set of all classes that are related or grouped by the application and which must be colocated in their mapping to a single database.

4.2. Statement Types

A Jakarta Persistence query language statement may be either a select statement, an update statement, or a delete statement.

> This chapter refers to all such statements as “queries”. Where it is important to distinguish among statement types, the specific statement type is referenced.

In BNF syntax, a query language statement is defined as:

```
QL_statement ::= select_statement | update_statement | delete_statement
```

Any Jakarta Persistence query language statement may be constructed dynamically or may be statically defined in a metadata annotation or XML descriptor element.

All statement types may have parameters.

4.2.1. Select Statements

A select query is a string with the following clauses:

- a `SELECT` clause, which determines the type of the objects or values to be selected.
- a `FROM` clause, which provides declarations that designate the domain to which the expressions specified in the other clauses of the query apply.
- an optional `WHERE` clause, which may be used to restrict the results that are returned by the query.
- an optional `GROUP BY` clause, which allows query results to be aggregated in terms of groups.
- an optional `HAVING` clause, which allows filtering over aggregated groups.
- an optional `ORDER BY` clause, which may be used to order the results that are returned by the query.

In BNF syntax, a select query is defined by:

```
select_query ::= [select_clause] from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]
```

Every select statement has a `FROM` clause. The square brackets `[]` in the BNF indicate that the other clauses are optional.

4.2.1.1. Set Operators in Select Statements

A select statement may be a single select query, or it may combine multiple select queries using the binary left-associative operators `UNION`, `UNION ALL`, `INTERSECT`, `INTERSECT ALL`, `EXCEPT`, and `EXCEPT ALL`. The semantics of these operators are identical to SQL. [1]

The full syntax for a select statement is defined by:

```
select_statement ::= union
union ::= intersection | union {UNION [ALL] | EXCEPT [ALL]} intersection
intersection ::= query_expression | intersection INTERSECT [ALL] query_expression
query_expression ::= select_query | (union)
```

A provider is only required to support select statements where every constituent select query has the same number of items in the select clause, and where corresponding items in the select clauses of the constituent select queries either:

- have exactly the same type, as defined by Section 4.9.1, or
- are entity types which inherit a common entity type, as defined by Section 2.13.

4.2.2. Update and Delete Statements

Update and delete statements provide bulk operations over sets of entities.

In BNF syntax, these operations are defined by:

```
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
```

The update and delete clauses determine the type of the entities to be updated or deleted. The `WHERE` clause may be used to restrict the scope of the update or delete operation.

Update and delete statements are described further in Section 4.11.

4.3. Abstract Schema Types and Query Domains

The Jakarta Persistence query language is a typed language, and every expression has a type. The type of an expression is derived from the structure of the expression, the abstract schema types of the identification variable declarations, the types to which the persistent attributes evaluate, and the types of literals.

The abstract schema type of an entity or embeddable is derived from its class and the metadata information provided by Java language annotations or in the XML descriptor.

Informally, the abstract schema type of an entity or embeddable can be characterized as follows:

- For every non-relationship persistent field or get accessor method (for a persistent property) of the class, there is a field ("state field") whose abstract schema type corresponds to that of the field or the result type of the accessor method.
- For every persistent relationship field or get accessor method (for a persistent relationship property) of the class, there is a field ("association field") whose type is the abstract schema type of the related entity (or, if the relationship is a one-to-many or many-to-many, a collection of such).

Abstract schema types are specific to the query language data model. The persistence provider is not required to implement or otherwise materialize an abstract schema type.

The domain of a query consists of the abstract schema types of all entities and embeddables that are defined in the same persistence unit.

The domain of a query may be restricted by the *navigability* of the relationships of the entity and associated embeddable classes on which it is based. The association fields of an entity's or embeddable's abstract schema type determine navigability. Using the association fields and their values, a query can select related entities and use their abstract schema types in the query.

4.3.1. Naming

An entity is designated in a query by its *entity name*. The entity name is specified by the `name` element of the `@Entity` annotation, as defined in Section 10.1, or by the `entity-name` XML descriptor element, and defaults to the unqualified name of the entity class.[2]

Entity names are scoped within the persistence unit and must be unique within the persistence unit.

4.3.2. Example

This example assumes that the application developer provides several entity classes, representing orders, products, and line items, and an embeddable address class representing shipping addresses and billing addresses. The abstract schema types for the entities are `Order`, `Product`, and `LineItem` respectively. There is a one-to-many relationship between `Order` and `LineItem`. The entity `LineItem` is related to `Product` in a many-to-one relationship. The classes are logically in the same persistence unit, as shown in Figure 1.

Queries to select orders can be defined by navigating over the association fields and state fields defined by `Order` and `LineItem`. A query to find all orders with pending line items might be written as follows:

```
SELECT DISTINCT o
FROM Order AS o JOIN o.lineItems AS l
WHERE l.shipped = FALSE
```

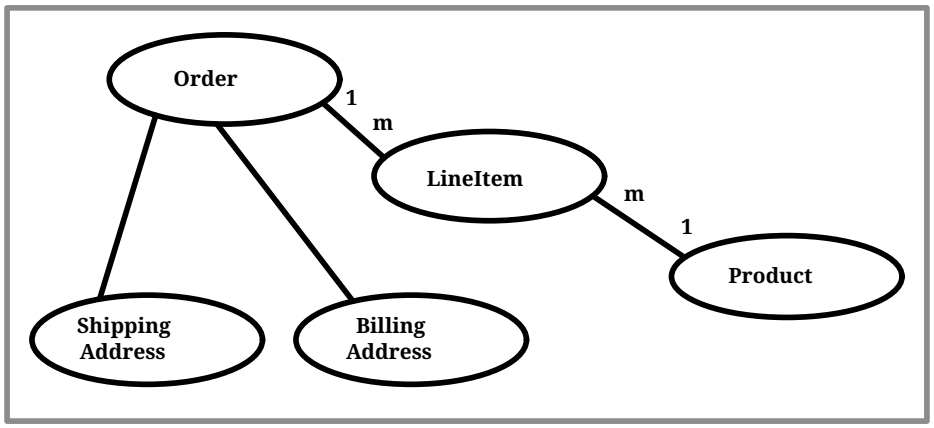

Figure 1. Abstract persistence schema of several entities defined in the same persistence unit.

This query navigates over the association field `lineItems` of the abstract schema type `Order` to find line items, and uses the state field `shipped` of `LineItem` to select those orders that have at least one line item that has not yet shipped. (Note that this query does not select orders that have no line items.)

Although reserved identifiers, such as `DISTINCT`, `FROM`, `AS`, `JOIN`, `WHERE`, and `FALSE` appear in upper case in this example, reserved identifiers are case insensitive.[3]

The `SELECT` clause of this example designates the return type of this query to be of type `Order`.

Because the same persistence unit defines the abstract persistence schema of the related entities, the developer can also specify a query over orders that utilizes the abstract schema type for products, and hence the state fields and association fields of both the abstract schema types `Order` and `Product`. For example, if the abstract schema type `Product` has a state field named `productType`, a query over orders can be specified using this state field. Such a query might be to find all orders for products with product type office supplies. A query for this might be as follows.

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType = 'office_supplies'
```

Because `Order` is related to `Product` by means of the relationships between `Order` and `LineItem` and between `LineItem` and `Product`, navigation using the association fields `lineItems` and `product` is used to express the query. This query is specified by using the entity name `Order`, which designates the abstract schema type over which the query ranges. The basis for the navigation is provided by the association fields `lineItems` and `product` of the abstract schema types `Order` and `LineItem` respectively.

4.4. The FROM Clause and Navigational Declarations

The `FROM` clause of a query defines the *domain* of the query:

- one or more named entity abstract schema types, as specified below in Section 4.4.3, together with
- zero or more joined associations and collections, as specified below in Section 4.4.5.

An *identification variable* is an identifier declared in the `FROM` clause of a query. Each identification variable is assigned an abstract schema type. Each element of the domain may declare an identification variable.

- If the domain has exactly one named entity abstract schema type and no joins, then the named entity does not require an explicit identification variable, and its identification variable defaults to the *implicit identification variable*, `this`.
- Otherwise, every element of the `FROM` clause—that is, every named entity abstract schema types and every join—

must declare an identification variable.

```
from_clause ::=
    FROM {this_implicit_variable | identification_variable_declarations}

this_implicit_variable ::= entity_name

identification_variable_declarations ::=
    identification_variable_declaration
    {, {identification_variable_declaration | collection_member_declaration}}*

identification_variable_declaration ::= range_variable_declaration {join | fetch_join}*

range_variable_declaration ::= entity_name [AS] identification_variable

join ::= range_join | path_join

range_join ::= join_spec range_variable_declaration [join_condition]

path_join ::=
    join_spec join_association_path_expression [AS] identification_variable [join_condition]

fetch_join ::= join_spec FETCH join_association_path_expression

join_spec ::= [INNER | LEFT [OUTER]] JOIN

join_association_path_expression ::=
    join_collection_valued_path_expression |
    join_single_valued_path_expression |
    TREAT(join_collection_valued_path_expression AS subtype) |
    TREAT(join_single_valued_path_expression AS subtype)

join_collection_valued_path_expression ::=
[identification_variable.]{single_valued_embeddable_object_field.}*collection_valued_field

join_single_valued_path_expression ::=
[identification_variable.]{single_valued_embeddable_object_field.}*single_valued_object_field

join_condition ::= ON conditional_expression

collection_member_declaration ::= IN (collection_valued_path_expression) [AS] identification_variable
```

The following subsections discuss the constructs used in the `FROM` clause.

4.4.1. Identifiers

An identifier is a character sequence of unlimited length. The character sequence must begin with a Java identifier start character, and all other characters must be Java identifier part characters. An identifier start character is any character for which the method `Character.isJavaIdentifierStart` returns true. This includes the underscore (_) character and the dollar sign ($) character. An identifier part character is any character for which the method `Character.isJavaIdentifierPart` returns true. The question mark (?) character is reserved for use by the Jakarta Persistence query language.

The following[4] are reserved identifiers: `ABS`, `ALL`, `AND`, `ANY`, `AS`, `ASC`, `AVG`, `BETWEEN`, `BIT_LENGTH`, `BOTH`, `BY`, `CASE`, `CEILING`, `CHAR_LENGTH`, `CHARACTER_LENGTH`, `CLASS`, `COALESCE`, `CONCAT`, `COUNT`, `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `DELETE`, `DESC`, `DISTINCT`, `ELSE`, `EMPTY`, `END`, `ENTRY`, `ESCAPE`, `EXISTS`, `EXP`, `EXTRACT`, `FALSE`, `FETCH`, `FIRST`, `FLOOR`, `FROM`, `FUNCTION`, `GROUP`, `HAVING`, `IN`, `INDEX`, `INNER`, `IS`, `JOIN`, `KEY`, `LEADING`, `LAST`, `LEFT`, `LENGTH`, `LIKE`, `LOCAL`, `LN`, `LOCATE`, `LOWER`, `MAX`, `MEMBER`, `MIN`, `MOD`, `NEW`, `NOT`, `NULL`, `NULLS`, `NULLIF`, `OBJECT`, `OF`, `ON`, `OR`, `ORDER`, `OUTER`, `POSITION`, `POWER`, `REPLACE`, `RIGHT`, `ROUND`, `SELECT`, `SET`, `SIGN`, `SIZE`, `SOME`, `SQRT`, `SUBSTRING`, `SUM`, `THEN`, `TRAILING`, `TREAT`, `TRIM`, `TRUE`, `TYPE`, `UNKNOWN`, `UPDATE`, `UPPER`, `VALUE`, `WHEN`, `WHERE`.

Reserved identifiers are case-insensitive. Reserved identifiers must not be used as identification variables or result variables (see Section 4.9).

> It is recommended that SQL keywords other than those listed above not be used as identification variables in queries because they may be used as reserved identifiers in future releases of this specification.

4.4.2. Identification Variables

An identification variable is a valid identifier declared in the `FROM` clause of a query.

Every identification variable must be declared in the `FROM` clause, except for the implicit identification variable `this`. Identification variables are never declared in other clauses.

An identification variable must not be a reserved identifier.

An identification variable may have the same name as an entity.

Identification variables are case-insensitive.

An identification variable evaluates to a value of the type of the expression used in declaring the variable. For example, consider the previous query:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType = 'office_supplies'
```

In the `FROM` clause declaration `o.lineItems l`, the identification variable `l` evaluates to any `LineItem` value directly reachable from `Order`. The association field `lineItems` is a collection of instances of the abstract schema type LineItem and the identification variable `l` refers to an element of this collection. The type of `l` is the abstract schema type of `LineItem`.

An identification variable can range over an entity, embeddable, or basic abstract schema type. An identification variable designates an instance of an abstract schema type or an element of a collection of abstract schema type instances.

Note that for identification variables referring to an instance of an association or collection represented as a `java.util.Map`, the identification variable is of the abstract schema type of the map `value`.

An identification variable always designates a reference to a single value. It is declared in one of three ways: in a range variable declaration, in a join clause, or in a collection member declaration. The identification variable declarations are evaluated from left to right in the `FROM` clause, and an identification variable declaration can use the result of a preceding identification variable declaration of the query string.

All identification variables used in the `SELECT`, `WHERE`, `ORDER BY`, `GROUP BY`, or `HAVING` clause of a `SELECT` or `DELETE` statement must be declared in the `FROM` clause. The identification variables used in the `WHERE` clause of an `UPDATE` statement must be declared in the `UPDATE` clause.

Identification variables are existentially quantified in these clauses. This means that an identification variable represents a member of a collection or an instance of an entity's abstract schema type. An identification variable never designates a collection in its entirety.

An identification variable is scoped to the query (or subquery) in which it is defined and is also visible to any subqueries within that query scope that do not define an identification variable of the same name.

4.4.3. Range Variable Declarations

A range variable declaration introduces a query domain element ranging over a given named entity abstract schema type, with an associated identification variable.

The syntax for declaring an identification variable as a range variable is similar to that of SQL; optionally, it may use the `AS` keyword. A range variable declaration designates an entity abstract schema type by its entity name, as defined above in Section 4.3.1.[5]

```
range_variable_declaration ::= entity_name [AS] identification_variable
```

The entity name in a range variable declaration is case-sensitive.

Range variable declarations allow the developer to designate a “root” for objects which may not be reachable by navigation.

In order to select values by comparing more than one instance of an entity abstract schema type, more than one identification variable ranging over the abstract schema type is needed in the `FROM` clause.

The following query returns orders whose quantity is greater than the order quantity for John Smith. This example illustrates the use of two different identification variables in the `FROM` clause, both of the abstract schema type `Order`. The `SELECT` clause of this query determines that it is the orders with quantities larger than John Smith’s that are returned.

```
SELECT DISTINCT o1
FROM Order o1, Order o2
WHERE o1.quantity > o2.quantity AND
 o2.customer.lastname = 'Smith' AND
 o2.customer.firstname= 'John'
```

If the query domain is a single entity abstract schema type, the range variable declaration is optional. These queries are equivalent:

```
SELECT quantity
FROM Order
WHERE customer.lastname = 'Smith'
  AND customer.firstname= 'John'


SELECT this.quantity
FROM Order
WHERE this.customer.lastname = 'Smith'
  AND this.customer.firstname= 'John'


SELECT ord.quantity
FROM Order AS ord
WHERE ord.customer.lastname = 'Smith'
  AND ord.customer.firstname= 'John'
```

Otherwise, if the query domain has more than one element, each named entity abstract schema type listed in the `FROM` clause must be a range variable declaration, and the implicit identification variable is not implicitly assigned an abstract schema type.

4.4.4. Path Expressions

A path expression is a sequence of identifiers uniquely identifying a state field or association field of an element of the query domain.

A path expression may begin with a reference to an identification variable, followed by the navigation operator (`.`). If the first element of a path expression is not an identification variable, then the path expression is interpreted exactly as if it began with the implicit identification variable `this`.

The remaining elements of the path expression are interpreted as references to state fields or association fields in the context of the abstract schema type assigned to the identification variable—or to `this`, if the path expression does not begin with an identification variable.

A reference to a state field or association field in a path expression is case-sensitive.

The type of the path expression is the type computed as the result of navigation; that is, the type of the state field or association field to which the expression navigates. The type of a path expression that navigates to an association field may be specified as a subtype of the declared type of the association field by means of the `TREAT` operator. See Section 4.4.9.

An identification variable qualified by the `KEY`, `VALUE`, or `ENTRY` operator is a path expression. The `KEY`, `VALUE`, and `ENTRY` operators may only be applied to identification variables that correspond to map-valued associations or map-valued element collections. The type of the path expression is the type computed as the result of the operation; that is, the abstract schema type of the field that is the value of the `KEY`, `VALUE`, or `ENTRY` operator (the map key, map value, or map entry respectively).[6]

In the following query, photos is a map from photo label to filename.

```
SELECT i.name, VALUE(p)
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE '%egret'
```

In the above query the identification variable `p` designates an abstract schema type corresponding to the map `value`. The results of `VALUE(p)` and `KEY(p)` are the map value and the map key associated with `p`, respectively. The following query is equivalent:

```
SELECT i.name, p
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE '%egret'
```

A path expression using the `KEY` or `VALUE` operator can be further composed. A path expression using the `ENTRY` operator is terminal. It cannot be further composed and can only appear in the `SELECT` list of a query.

The syntax for qualified identification variables is as follows.

```
qualified_identification_variable ::=
    map_field_identification_variable |
    ENTRY(identification_variable)

map_field_identification_variable ::=
    KEY(identification_variable) |
    VALUE(identification_variable)
```

Depending on navigability, a path expression that leads to an association field or to a field whose type is an embeddable class may be further composed. Path expressions can be composed from other path expressions if the original path expression evaluates to a single-valued type (not a collection).

In the following example, simple data model with `Employee`, `ContactInfo`, `Address` and `Phone` classes is used:

```
@Entity
public class Employee {
    @Id int id;
```

```
    @Embedded
    private ContactInfo contactInfo;
}

@Entity
public class Phone {
    @Id
    private int id;
    private String vendor;
}

@Embeddable
public class ContactInfo {
    @Embedded
    private Address address;
    @ManyToMany
    private List<Phone> phones;
}

@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;
    private String zipcode;
}
```

The `contactInfo` field denotes an embeddable class consisting of an address and set of phones.

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo.phones p
WHERE e.contactInfo.address.zipcode = '95054'
```

Path expression navigability is composed using “inner join” semantics. That is, if the value of a non-terminal field in the path expression is null, the path is considered to have no value, and does not participate in the determination of the result.

The following query is equivalent to the query above:

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo c JOIN c.phones p
WHERE e.contactInfo.address.zipcode = '95054'
```

4.4.4.1. Path Expression Syntax

The syntax for single-valued path expressions and collection-valued path expressions is as follows.

An identification variable used in a `single_valued_object_path_expression` or in a `collection_valued_path_expression` may be an unqualified identification variable or an identification variable to which the `KEY` or `VALUE` function has been applied.

```
general_identification_variable ::=
    identification_variable |
    map_field_identification_variable
```

The type of an entity-valued path expression or an entity-valued subpath of a path expression used in a `WHERE` clause may be specified as a subtype of the corresponding declared type by means of the `TREAT` operator. See Section 4.4.9.

```
general_subpath ::= simple_subpath | treated_subpath{.single_valued_object_field}*

simple_subpath ::=
    general_identification_variable |
```

```
    general_identification_variable{.single_valued_object_field}*

treated_subpath ::= TREAT(general_subpath AS subtype)

single_valued_path_expression ::=
    qualified_identification_variable |
    TREAT(qualified_identification_variable AS subtype) |
    state_field_path_expression |
    single_valued_object_path_expression

state_field_path_expression ::= [general_subpath.]state_field

state_valued_path_expression ::= state_field_path_expression | general_identification_variable

single_valued_object_path_expression ::= general_subpath.single_valued_object_field

collection_valued_path_expression ::= general_subpath.collection_valued_field
```

A `single_valued_object_field` is designated by the name of an association field in a one-to-one or many-to-one relationship or a field of embeddable class type. The type of a `single_valued_object_field` is the abstract schema type of the related entity or embeddable class.

A `single_valued_embeddable_object_field` is designated by the name of a field of embeddable class type.

A `state_field` is designated by the name of an entity or embeddable class state field that corresponds to a basic type.

A `collection_valued_field` is designated by the name of an association field in a one-to-many or a many-to-many relationship or by the name of an element collection field. The type of a `collection_valued_field` is a collection of values of the abstract schema type of the related entity or element type.

It is syntactically illegal to compose a path expression from a path expression that evaluates to a collection. For example, if `o` designates `Order`, the path expression `o.lineItems.product` is illegal since navigation to `lineItems` results in a collection. This case should produce an error when the query string is verified. To handle such a navigation, an identification variable must be declared in the `FROM` clause to range over the elements of the `lineItems` collection. Another path expression must be used to navigate over each such element in the `WHERE` clause of the query, as in the following:

```
SELECT DISTINCT l.product
FROM Order AS o JOIN o.lineItems l
```

A `collection_valued_path_expression` may only occur in:

- the `FROM` clause of a query,
- an `empty_collection_comparison_expression`,
- a `collection_member_expression`, or
- as an argument to the `SIZE` operator.

See Section 4.6.8, Section 4.6.9, and Section 4.7.7.2.

4.4.5. Joins

JPQL defines the following varieties of join:

- inner joins, and
- left outer joins.[7]

The semantics of each variety of join is identical to SQL, and the syntax is borrowed from ANSI SQL.

Every join has a target, either:

- an entity-valued path expression, or
- an entity type (that is, range variable declaration, as already specified in Section 4.4.3).

An inner join may be implicitly specified by the use of a cartesian product in the `FROM` clause and a join condition in the `WHERE` clause. In the absence of a join condition, this reduces to the cartesian product.

The main use case for this generalized style of join is when a join condition does not involve a foreign key relationship mapped to an association between entities.

Example:

```
SELECT c FROM Customer c, Employee e WHERE c.hatsize = e.shoesize
```

This style of inner join (sometimes called a "theta" join) is less typical than explicitly defined joins over relationships.

The syntax for explicit join operations is given by:

```
join ::= range_join | path_join

range_join ::= join_spec range_variable_declaration [join_condition]

path_join ::=
    join_spec join_association_path_expression [AS] identification_variable [join_condition]

fetch_join ::= join_spec FETCH join_association_path_expression

join_spec ::= [INNER | LEFT [OUTER]] JOIN

join_association_path_expression ::=
    join_collection_valued_path_expression |
    join_single_valued_path_expression |
    TREAT(join_collection_valued_path_expression AS subtype) |
    TREAT(join_single_valued_path_expression AS subtype)

join_collection_valued_path_expression ::=
    [identification_variable.]{single_valued_embeddable_object_field.}*collection_valued_field

join_single_valued_path_expression ::=
    [identification_variable.]{single_valued_embeddable_object_field.}*single_valued_object_field

join_condition ::= ON conditional_expression
```

The inner and outer join operation types described in Section 4.4.5.1, Section 4.4.5.2, and Section 4.4.5.3 are supported.

4.4.5.1. Inner Joins

The syntax for an inner join to an entity type is given by:

```
[INNER] JOIN range_variable_declaration [join_condition]
```

The keyword `INNER` is optional and does not affect the semantics of the query.

```
SELECT c
FROM Customer c
    JOIN Order o ON o.customer.id = c.id
WHERE c.status = 1
```

Or, equivalently:

```
SELECT c
FROM Customer c
    INNER JOIN Order o ON o.customer.id = c.id
WHERE c.status = 1
```

These queries are equivalent to the following query involving an implicit "theta" join:

```
SELECT c
FROM Customer c, Order o
WHERE o.customer.id = c.id AND c.status = 1
```

The syntax for an inner join over an association is given by:

```
[INNER] JOIN join_association_path_expression [AS] identification_variable [join_condition]
```

For example, the query below joins over the relationship between customers and orders. This type of join typically equates to a join over a foreign key relationship in the database.

```
SELECT c
FROM Customer c
   JOIN c.orders o
WHERE c.status = 1
```

Equivalently:

```
SELECT c
FROM Customer c
    INNER JOIN c.orders o
WHERE c.status = 1
```

This is equivalent to the following query using the earlier `IN` construct, defined in [4]. It selects those customers of status 1 for which at least one order exists:

```
SELECT OBJECT(c)
FROM Customer c, IN(c.orders) o
WHERE c.status = 1
```

The query below joins over `Employee`, `ContactInfo` and `Phone`. `ContactInfo` is an embeddable class that consists of an address and set of phones. `Phone` is an entity.

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo c JOIN c.phones p
WHERE c.address.zipcode = '95054'
```

A join condition may be specified for an inner join. This is equivalent to specification of the same condition in the `WHERE` clause.

4.4.5.2. Outer Joins

The syntax for an outer join to an entity type is given by:

```
LEFT [OUTER] JOIN range_variable_declaration [join_condition]
```

The keyword `OUTER` is optional and does not affect the semantics of the query.

```
SELECT c
```

```
FROM Customer c
    LEFT JOIN Order o ON o.customer.id = c.id
WHERE c.status = 1
```

Or, equivalently:

```
SELECT c
FROM Customer c
    LEFT OUTER JOIN Order o ON o.customer.id = c.id
WHERE c.status = 1
```

Outer joins enable the retrieval of a set of entities where matching values in the join condition may be absent. For example, the queries above return `Customer` instances with no matching `Order`.

The syntax for an outer join over an association is given by:

```
LEFT [OUTER] JOIN join_association_path_expression [AS] identification_variable [join_condition]
```

An association outer join without no explicit `join_condition` has an implicit join condition inferred from the foreign key relationship mapped by the `join_association_path_expression`. Typically, a JPQL join of this form is translated to a SQL outer join with an `ON` condition specifying the foreign key relationship, as in the following examples.

Jakarta Persistence query language:

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
GROUP BY s.name
```

SQL:

```
SELECT s.name, COUNT(p.id)
FROM Suppliers s LEFT JOIN Products p
    ON s.id = p.supplierId
GROUP By s.name
```

An explicit `join_condition` (that is, an `ON` condition in the `JOIN`) results in an additional restriction in the `ON` condition of the generated SQL.

Jakarta Persistence query language:

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
    ON p.status = 'inStock'
GROUP BY s.name
```

SQL:

```
SELECT s.name, COUNT(p.id)
FROM Suppliers s LEFT JOIN Products p
    ON s.id = p.supplierId AND p.status = 'inStock'
GROUP BY s.name
```

Note that the result of this query will be different from that of the following query:

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
WHERE p.status = 'inStock'
GROUP BY s.name
```

The result of the latter query will exclude suppliers who have no products in stock whereas the former query will include them.

An important use case for `LEFT JOIN` is in enabling the prefetching of related data items as a side effect of a query. This is accomplished by specifying the `LEFT JOIN` as a fetch join, that is, `LEFT JOIN FETCH`, as described below.

4.4.5.3. Fetch Joins

A *fetch join* clause in a query results in eager fetching of an association or element collection as a side effect of execution of the query.

The syntax for a fetch join is given by:

```
fetch_join ::= [LEFT [OUTER] | INNER] JOIN FETCH join_association_path_expression
```

A fetch join must be an `INNER` or `LEFT` (`OUTER`) join. A fetch join does not have an explicit join condition or identification variable.

The association referenced by the right side of the fetch join clause must be an association or element collection that is referenced from an entity or embeddable that is returned as a result of the query. It is not permitted to specify an identification variable for the objects referenced by the right side of the fetch join clause, and hence references to the implicitly fetched entities or elements cannot appear elsewhere in the query.

The following query returns a set of departments. As a side effect, the associated employees for those departments are also retrieved, even though they are not part of the explicit query result. The initialization of the persistent state or relationship fields or properties of the objects that are retrieved as a result of a fetch join is determined by the metadata for that class—in this example, the `Employee` entity class.

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

A fetch join has the same join semantics as the corresponding inner or outer join, except that the related objects specified on the right-hand side of the join operation are not returned in the query result or otherwise referenced in the query. Hence, for example, if department 1 has five employees, the above query returns five references to the department 1 entity.

The fetch join construct must not be used in the `FROM` clause of a subquery.

4.4.6. Collection Member Declarations

An identification variable declared by a `collection_member_declaration` ranges over values of a collection obtained by navigation using a path expression.

An identification variable of a collection member declaration is declared using a special operator, the reserved identifier `IN`. The argument to the `IN` operator is a collection-valued path expression. The path expression evaluates to a collection type specified as a result of navigation to a collection-valued association field of an entity or embeddable class abstract schema type.

The syntax for declaring a collection member identification variable is as follows:

```
collection_member_declaration ::=
    IN (collection_valued_path_expression) [AS] identification_variable
```

For example, the query

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l
WHERE l.product.productType = 'office_supplies'
```

can equivalently be expressed as follows, using the `IN` operator:

```
SELECT DISTINCT o
FROM Order o, IN(o.lineItems) l
WHERE l.product.productType = 'office_supplies'
```

In this example, `lineItems` is the name of an association field whose value is a collection of instances of the abstract schema type `LineItem`. The identification variable `l` designates a member of this collection, a single `LineItem` abstract schema type instance. In this example, `o` is an identification variable of the abstract schema type `Order`.

4.4.7. FROM Clause and SQL

The Jakarta Persistence query language treats the `FROM` clause similarly to SQL in that the declared identification variables affect the results of the query even if they are not used in the `WHERE` clause. Application developers should use caution in defining identification variables because the domain of the query can depend on whether there are any values of the declared type.

For example, the `FROM` clause below defines a query over all orders that have line items and existing products. If there are no `Product` instances in the database, the domain of the query is empty and no order is selected.

```
SELECT o
FROM Order AS o JOIN o.lineItems l JOIN l.product p
```

4.4.8. Polymorphism

Jakarta Persistence queries are automatically polymorphic. The `FROM` clause of a query designates not only instances of the specific entity class(es) to which it explicitly refers but instances of subclasses of those classes as well. The instances returned by a query thus include instances of the subclasses that satisfy the query criteria.

Non-polymorphic queries or queries whose polymorphism is restricted can be specified using entity type expressions in the `WHERE` clause to restrict the domain of the query. See Section 4.7.12.

4.4.9. Downcasting

The use of the `TREAT` operator is supported for downcasting within path expressions in the `FROM` and `WHERE` clauses. Use of the `TREAT` operator allows access to subclass-specific state.

If during query execution the first argument to the `TREAT` operator is not a subtype (proper or improper) of the target type, the path is considered to have no value, and does not participate in the determination of the result. That is, in the case of a join, the referenced object does not participate in the result, and in the case of a restriction, the associated predicate is false. Use of the `TREAT` operator therefore also has the effect of filtering on the specified type (and its subtypes) as well as performing the downcast. If the target type is not a subtype (proper or improper) of the static type of the first argument, the query is invalid.

Examples:

```
SELECT b.name, b.ISBN
FROM Order o JOIN TREAT(o.product AS Book) b

SELECT e FROM Employee e JOIN TREAT(e.projects AS LargeProject) lp
WHERE lp.budget > 1000
```

```
SELECT e FROM Employee e JOIN e.projects p
WHERE TREAT(p AS LargeProject).budget > 1000
    OR TREAT(p AS SmallProject).name LIKE 'Persist%'
    OR p.description LIKE "cost overrun"

SELECT e FROM Employee e
WHERE TREAT(e AS Exempt).vacationDays > 10
    OR TREAT(e AS Contractor).hours > 100
```

4.5. WHERE Clause

The `WHERE` clause of a query consists of a conditional expression used to select objects or values that satisfy the expression. The `WHERE` clause restricts the result of a select statement or the scope of an update or delete operation.

A `WHERE` clause is defined as follows:

```
where_clause ::= WHERE conditional_expression
```

The `GROUP BY` construct enables the aggregation of values according to the properties of an entity class. The `HAVING` construct enables conditions to be specified that further restrict the query result as restrictions upon the groups.

The syntax of the `HAVING` clause is as follows:

```
having_clause ::= HAVING conditional_expression
```

The `GROUP BY` and `HAVING` constructs are further discussed in Section 4.8.

4.6. Conditional Expressions

The following sections describe language constructs that can be used in a conditional expression of the `WHERE` clause, the `HAVING` clause, or in an `ON` condition.

State fields that are mapped in serialized form or as lobs cannot be portably used in conditional [8].

4.6.1. Conditional Expression Composition

Conditional expressions are composed of other conditional expressions, comparison operations, logical operations, path expressions that evaluate to boolean values, boolean literals, and boolean input parameters.

The scalar expressions described in Section 4.7 can be used in conditional expressions.

Aggregate functions can only be used in conditional expressions in a `HAVING` clause. See Section 4.8.

Standard bracketing `()` for ordering expression evaluation is supported.

Conditional expressions are defined as follows:

```
conditional_expression ::= conditional_term | conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::=
    comparison_expression |
    between_expression |
    in_expression |
    like_expression |
    null_comparison_expression |
```

```
    empty_collection_comparison_expression |
    collection_member_expression |
    exists_expression
```

4.6.2. Operators and Operator Precedence

The operators are listed below in order of decreasing precedence.

- Navigation operator (`.`)
- Arithmetic operators:
  - `+`, `-` unary
  - `*`, `/` multiplication and division
  - `+`, `-` addition and subtraction
- String concatenation (`||`)
- Comparison operators: `=`, `>`, `>=`, `<` , `<=`, `<>` (not equal), `[NOT] BETWEEN`, `[NOT] LIKE`, `[NOT] IN`, `IS [NOT] NULL`, `IS [NOT] EMPTY`, `[NOT] MEMBER [OF]`, `[NOT] EXISTS`
- Logical operators:
  - `NOT`
  - `AND`
  - `OR`

The following sections describe operators used in specific expressions.

4.6.3. Comparison Expressions

The syntax for the use of comparison expressions in a conditional expression is as follows[9]:

```
comparison_expression ::=
    string_expression comparison_operator {string_expression | all_or_any_expression} |
    boolean_expression {= | <>} {boolean_expression | all_or_any_expression} |
    enum_expression {= | <>} {enum_expression | all_or_any_expression} |
    datetime_expression comparison_operator
        {datetime_expression | all_or_any_expression} |
    entity_expression {= | <>} {entity_expression | all_or_any_expression} |
    arithmetic_expression comparison_operator
        {arithmetic_expression | all_or_any_expression} |
    entity_id_or_version_function {= | <>} input_parameter |
    entity_type_expression {= | <>} entity_type_expression}

comparison_operator ::= = | > | >= | < | <= | <>
```

Examples:

```
item.cost * 1.08 <= 100.00
CONCAT(person.lastName, ', ', person.firstName)) = 'Jones, Sam'
TYPE(e) = ExemptEmployee
```

4.6.4. Between Expressions

The syntax for the use of the comparison operator `[NOT] BETWEEN` in a conditional expression is as follows:

```
between_expression ::=
    arithmetic_expression [NOT] BETWEEN arithmetic_expression AND arithmetic_expression |
    string_expression [NOT] BETWEEN string_expression AND string_expression |
    datetime_expression [NOT] BETWEEN datetime_expression AND datetime_expression
```

The `BETWEEN` expression

```
x BETWEEN y AND z
```

is semantically equivalent to:

```
y <= x AND x <= z
```

The rules for unknown and `NULL` values in comparison operations apply. See Section 4.6.13.

Examples:

- `p.age BETWEEN 15 and 19` is equivalent to `p.age >= 15 AND p.age <= 19`
- `p.age NOT BETWEEN 15 and 19` is equivalent to `p.age < 15 OR p.age > 19`

In the following example, `transactionHistory` is a list of credit card transactions defined using an order column.

```
SELECT t
FROM CreditCard c JOIN c.transactionHistory t
WHERE c.holder.name = 'John Doe' AND INDEX(t) BETWEEN 0 AND 9
```

4.6.5. In Expressions

The syntax for the use of the comparison operator `[NOT] IN` in a conditional expression is as follows:

```
in_expression ::=
    {state_valued_path_expression | type_discriminator} [NOT] IN
        {(in_item {, in_item}*) | (subquery) | collection_valued_input_parameter}
in_item ::= literal | single_valued_input_parameter
```

The `state_valued_path_expression` must have a string, numeric, date, time, timestamp, or enum value.

The literal and/or input parameter values must be *like* the abstract schema type of the `state_valued_path_expression` in type. (See Section 4.6.14.)

The results of the subquery must be *like* the abstract schema type of the `state_valued_path_expression` in type. Subqueries are discussed in Section 4.6.12.

Example 1:

```
o.country IN ('UK', 'US', 'France')
```

is true for `UK` and false for `Peru`, and is equivalent to the expression

```
(o.country = 'UK') OR (o.country = 'US') OR (o.country = 'France')
```

Example 2:

```
o.country NOT IN ('UK', 'US', 'France')
```

is false for `UK` and true for `Peru`, and is equivalent to the expression

```
NOT ((o.country = 'UK') OR (o.country = 'US') OR (o.country = 'France'))
```

If an `IN` or `NOT IN` expression has a list of `in_item` expressions, there must be at least one item in the list. The value of such expressions is determined according to the following rules:

- If the `state_valued_path_expression` in an `IN` or `NOT IN` expression evaluates to `NULL` or unknown, then the whole `IN` or `NOT IN` expression evaluates to `NULL` or unknown.
- Otherwise, if the `state_valued_path_expression` and at least one `in_item` evaluate to the same value, the whole `IN` or `NOT IN` expression evaluates to true.
- Otherwise, if the value of a `state_valued_path_expression` evaluates to a value distinct from the value of every `in_item` expression, the whole `IN` or `NOT IN` expression evaluates to:
  - false, if every `in_item` expression evaluates to a non-null value, or
  - `NULL` or unknown if at least one `in_item` expression evaluates to null.

The list of values may be parameterized by a collection-valued input parameter. [10] (See Section 4.7.4.)

```
o.country NOT IN :countries
```

4.6.6. Like Expressions

The syntax for the use of the comparison operator `[NOT] LIKE` in a conditional expression is as follows:

```
like_expression ::=
    string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]
```

The `string_expression` must have a string value. The `pattern_value` is a string literal or a string-valued input parameter in which an underscore (_) stands for any single character, a percent (%) character stands for any sequence of characters (including the empty sequence), and all other characters stand for themselves. The optional `escape_character` is a single-character string literal or a character-valued input parameter (i.e., `char` or `Character`) and is used to escape the special meaning of the underscore and percent characters in `pattern_value`. [11]

Examples:

- `address.phone LIKE '12%3'` is true for `'123'`, `'12993'` and false for `'1234'`
- `asentence.word LIKE 'l_se'` is true for `'lose'` and false for `'loose'`
- `aword.underscored LIKE '_%' ESCAPE '\'` is true for `'_foo'` and false for `'bar'`
- `address.phone NOT LIKE '12%3'` is false for `'123'` and `'12993'` and true for `'1234'`

If the value of the `string_expression` or `pattern_value` is `NULL` or unknown, the value of the `LIKE` expression is unknown. If the `escape_character` is specified and is `NULL`, the value of the `LIKE` expression is unknown.

4.6.7. Null Comparison Expressions

The syntax for the use of the comparison operator IS `NULL` in a conditional expression is as follows:

```
null_comparison_expression ::=
    {single_valued_path_expression | input_parameter} IS [NOT] NULL
```

A null comparison expression tests whether or not the single-valued path expression or input parameter is a `NULL` value.

Null comparisons over instances of embeddable class types are not supported. Support for comparisons over embeddables may be added in a future release of this specification.

4.6.8. Empty Collection Comparison Expressions

The syntax for the use of the comparison operator `IS EMPTY` in an `empty_collection_comparison_expression` is as follows:

```
empty_collection_comparison_expression ::=
```

```
    collection_valued_path_expression IS [NOT] EMPTY
```

This expression tests whether or not the collection designated by the collection-valued path expression is empty (i.e, has no elements).

Example:

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

If the value of the collection-valued path expression in an empty collection comparison expression is unknown, the value of the empty comparison expression is unknown.

4.6.9. Collection Member Expressions

The syntax for the use of the comparison operator `MEMBER OF` [12] in a `collection_member_expression` is as follows:

```
collection_member_expression ::=
    entity_or_value_expression [NOT] MEMBER [OF] collection_valued_path_expression
entity_or_value_expression ::=
    single_valued_object_path_expression |
    state_valued_path_expression |
    simple_entity_or_value_expression
simple_entity_or_value_expression ::=
    identification_variable |
    input_parameter |
    literal
```

This expression tests whether the designated value is a member of the collection specified by the collection-valued path expression.

Expressions that evaluate to embeddable types are not supported in collection member expressions. Support for use of embeddables in collection member expressions may be added in a future release of this specification.

If the collection valued path expression designates an empty collection, the value of the `MEMBER OF` expression is `FALSE` and the value of the `NOT MEMBER OF` expression is `TRUE`. Otherwise, if the value of the `collection_valued_path_expression` or `entity_or_value_expression` in the collection member expression is `NULL` or unknown, the value of the collection member expression is unknown.

Example:

```
SELECT p
FROM Person p
WHERE 'Joe' MEMBER OF p.nicknames
```

4.6.10. Exists Expressions

An `EXISTS` expression is a predicate that is true only if the result of the subquery consists of one or more values and that is false otherwise.

The syntax of an exists expression is

```
exists_expression ::= [NOT] EXISTS (subquery)
```

Example:

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
    SELECT spouseEmp
    FROM Employee spouseEmp
        WHERE spouseEmp = emp.spouse)
```

The result of this query consists of all employees whose spouses are also employees.

4.6.11. All or Any Expressions

An `ALL` conditional expression is a predicate over a subquery that is true if the comparison operation is true for all values in the result of the subquery or the result of the subquery is empty. An `ALL` conditional expression is false if the result of the comparison is false for at least one value of the result of the subquery, and is unknown if neither true nor false.

An `ANY` conditional expression is a predicate over a subquery that is true if the comparison operation is true for some value in the result of the subquery. An `ANY` conditional expression is false if the result of the subquery is empty or if the comparison operation is false for every value in the result of the subquery, and is unknown if neither true nor false. The keyword `SOME` is synonymous with `ANY`.

The comparison operators used with `ALL` or `ANY` conditional expressions are =, <, <=, >, >=, <>. The result of the subquery must be like that of the other argument to the comparison operator in type. See Section 4.6.14.

The syntax of an `ALL` or `ANY` expression is specified as follows:

```
all_or_any_expression ::= {ALL | ANY | SOME} (subquery)
```

Example:

```
SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
    SELECT m.salary
    FROM Manager m
    WHERE m.department = emp.department)
```

The result of this query consists of all employees whose salaries exceed the salaries of all managers in their department.

4.6.12. Subqueries

Subqueries may be used in the `WHERE` or `HAVING` clause.[13]

The syntax for subqueries is as follows:

```
subquery ::= simple_select_clause subquery_from_clause [where_clause]
    [groupby_clause] [having_clause]
simple_select_clause ::= SELECT [DISTINCT] simple_select_expression
subquery_from_clause ::=
    FROM subselect_identification_variable_declaration
        {, subselect_identification_variable_declaration |
            collection_member_declaration}*
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    derived_path_expression [AS] identification_variable {join}* |
    derived_collection_member_declaration
simple_select_expression ::=
```

```
        single_valued_path_expression |
        scalar_expression |
        aggregate_expression |
        identification_variable
    derived_path_expression ::=
        general_derived_path.single_valued_object_field |
        general_derived_path.collection_valued_field
    general_derived_path ::=
        simple_derived_path |
        treated_derived_path{.single_valued_object_field}*
    simple_derived_path ::= superquery_identification_variable{.single_valued_object_field}*
    treated_derived_path ::= TREAT(general_derived_path AS subtype)
    derived_collection_member_declaration ::=
        IN superquery_identification_variable.{single_valued_object_field.}*collection_valued_field
```

Examples:

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
    SELECT spouseEmp
    FROM Employee spouseEmp
    WHERE spouseEmp = emp.spouse)
```

Note that some contexts in which a subquery can be used require that the subquery be a scalar subquery (i.e., produce a single result). This is illustrated in the following examples using numeric comparisons.

```
SELECT c
FROM Customer c
WHERE (SELECT AVG(o.price) FROM c.orders o) > 100

SELECT goodCustomer
FROM Customer goodCustomer
WHERE goodCustomer.balanceOwed < (
    SELECT AVG(c.balanceOwed)/2.0 FROM Customer c)
```

4.6.13. Null Values

When the target of a reference does not exist in the database, its value is regarded as `NULL`. SQL `NULL` semantics [2] defines the evaluation of conditional expressions containing `NULL` values.

The following is a brief description of these semantics:

- Comparison or arithmetic operations with a `NULL` value always yield an unknown value.
- Two `NULL` values are not considered to be equal, the comparison yields an unknown value.
- Comparison or arithmetic operations with an unknown value always yield an unknown value.
- The `IS NULL` and `IS NOT NULL` operators convert a `NULL` state field or single-valued object field value into the respective `TRUE` or `FALSE` value.
- Boolean operators use three valued logic, defined by Table 1, Table 2, and Table 3.

Table 1. Definition of the AND Operator

AND	T	F	U
T	T	F	U
F	F	F	F

AND	T	F	U
U	U	F	U

Table 2. Definition of the OR Operator

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Table 3. Definition of the NOT Operator

NOT	
T	F
F	T
U	U

The Jakarta Persistence query language defines the empty string, `''`, as a string with length zero, which is not equal to a `NULL` value. However, `NULL` values and empty strings may not always be distinguished when queries are mapped to some databases. Application developers should therefore not rely on the semantics of query comparisons involving the empty string and `NULL` value.

4.6.14. Equality and Comparison Semantics

Only the values of *like* types are permitted to be compared. A type is *like* another type if they correspond to the same Java language type, or if one is a primitive Java language type and the other is the wrapped Java class type equivalent (e.g., `int` and `Integer` are like types in this sense). There is one exception to this rule: it is valid to compare numeric values for which the rules of numeric promotion apply. Conditional expressions attempting to compare non-like type values are disallowed except for this numeric case.

Note that the arithmetic operators, the string concatenation operator, and comparison operators are permitted to be applied to state fields and input parameters of the wrapped Java class equivalents to the primitive numeric Java types.

Two entities of the same abstract schema type are equal if and only if they have the same primary key value.

Only equality/inequality comparisons over enums are required to be supported.

Comparisons over instances of embeddable class or map entry types are not supported.

The following examples illustrate the syntax and semantics of the Jakarta Persistence query language. These examples are based on the example presented in Section 4.3.2.

Find all orders:

```
SELECT o
```

```
FROM Order o
```

Find all orders that need to be shipped to California:

```
SELECT o
FROM Order o
WHERE o.shippingAddress.state = 'CA'
```

Find all states for which there are orders:

```
SELECT DISTINCT o.shippingAddress.state
FROM Order o
```

Find all orders that have line items:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l
```

Note that the result of this query does not include orders with no associated line items. This query can also be written as:

```
SELECT o
FROM Order o
WHERE o.lineItems IS NOT EMPTY
```

Find all orders that have no line items:

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

Find all pending orders:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l
WHERE l.shipped = FALSE
```

Find all orders in which the shipping address differs from the billing address. This example assumes that the application developer uses two distinct `entity` types to designate shipping and billing addresses.

```
SELECT o
FROM Order o
WHERE
    NOT (o.shippingAddress.state = o.billingAddress.state AND
         o.shippingAddress.city = o.billingAddress.city AND
         o.shippingAddress.street = o.billingAddress.street)
```

If the application developer uses a single `entity` type in two different relationships for both the shipping address and the billing address, the above expression can be simplified based on the equality rules defined in Section 4.6.14. The query can then be written as:

```
SELECT o
FROM Order o
WHERE o.shippingAddress <> o.billingAddress
```

The query checks whether the same entity abstract schema type instance (identified by its primary key) is related to an order through two distinct relationships.

4.6.14.1. Queries Using Input Parameters

The following query finds the orders for a product whose name is designated by an input parameter:

```
SELECT DISTINCT o
FROM Order o JOIN o.lineItems l
WHERE l.product.name = ?1
```

For this query, the input parameter must be of the type of the state field name, i.e., a string.

4.7. Scalar Expressions

Numeric, string, datetime, case, and entity type expressions result in scalar values.

Scalar expressions may be used in the `SELECT` clause of a query as well as in the `WHERE` [14] and `HAVING` clauses.

```
scalar_expression ::=
    arithmetic_expression |
    string_expression |
    enum_expression |
    datetime_expression |
    boolean_expression |
    case_expression |
    entity_type_expression |
    entity_id_or_version_function
```

4.7.1. Literals

A string literal is enclosed in single quotes—for example: `'literal'`. A string literal that includes a single quote is represented by two single quotes—for example: `'literal''s'`. String literals in queries, like Java `String` literals, use unicode character encoding. The use of Java escape notation is not supported in query string literals.

A numeric literal may be either:

- a decimal Java integer (int or long) literal
- a Java floating point (float or double) literal, or
- a literal `BigInteger` or `BigDecimal`.

A suffix `L`, `D`, or `F` may be used to indicate the specific numeric type, in accordance with the Java Language Specification. The suffix is not case-sensitive. The literal numeric value preceding the suffix must conform to the rules for Java numeric literals established by the Java Language Specification.

A suffix `BI` or `BD` may be used to indicate a literal `BigInteger` or `BigDecimal`, respectively. The literal numeric value preceding the suffix must be an exact or approximate SQL numeric literal. For a `BigInteger` literal, the numeric value must be an exact integer literal.

Just as in Java, when a numeric literal has no suffix:

- an integer literal is interpreted as a Java `int`, and
- a floating point literal is interpreted as a Java `double`.

Support for hexadecimal and octal numeric literals is not required by this specification.

Enum literals support the use of Java enum literal syntax. The fully qualified enum class name must be specified.

The JDBC escape syntax may be used for the specification of date, time, and timestamp literals. For example:

```
SELECT o
FROM Customer c JOIN c.orders o
WHERE c.name = 'Smith'
    AND o.submissionDate < {d '2008-12-31'}
```

The portability of this syntax for date, time, and timestamp literals is dependent upon the JDBC driver in use. Persistence providers are not required to translate from this syntax into the native syntax of the database or driver.

The boolean literals are `TRUE` and `FALSE`.

Entity type literals are specified by entity names—for example: `Customer`.

Although reserved literals appear in uppercase, they are case-insensitive.

4.7.2. Identification Variables

All identification variables used in the `WHERE` or `HAVING` clause of a `SELECT` or `DELETE` statement must be declared in the `FROM` clause, as described in Section 4.4.2. The identification variables used in the `WHERE` clause of an `UPDATE` statement must be declared in the `UPDATE` clause.

Identification variables are existentially quantified in the `WHERE` and `HAVING` clause. This means that an identification variable represents a member of a collection or an instance of an entity's abstract schema type. An identification variable never designates a collection in its entirety.

4.7.3. Path Expressions

It is illegal to use a `collection_valued_path_expression` within a `WHERE` or `HAVING` clause as part of a conditional expression except in an `empty_collection_comparison_expression`, in a `collection_member_expression`, or as an argument to the `SIZE` operator.

4.7.4. Input Parameters

An input parameter allows a value in the Java program to be safely interpolated into the text of the parameterized query.

In a given query, either positional or named parameters may be used. Positional and named parameters must not be mixed in a single query.

The persistence provider is required to support input parameters which occur in the `WHERE` clause or `HAVING` clause of a query, or as the new value for an update item in the `SET` clause of an update statement.

> Note that if an input parameter value is null, comparison operations or arithmetic operations involving the input parameter will result in an unknown value. See Section 4.6.13.

An input parameter might be *single-valued* or *collection-valued*. An input parameter which occurs directly to the right of the `IN` keyword in an `IN` predicate, as defined in Section 4.6.5, is collection-valued. Every other input parameter is single-valued.

The API for the binding concrete arguments to query parameters is described in Section 3.15.9.

4.7.4.1. Positional Parameters

The following rules apply to positional input parameters.

- A positional parameter is designated by an integer, and prefixed with a `?` symbol (question mark) in the text of the query string. For example: `?1`.
- Input parameters are numbered starting from 1.
- A given positional parameter may occur more than once in the query string.
- The ordering of the use of parameters within the text of the query string need not match the numbering of the positional parameters.

4.7.4.2. Named Parameters

A named parameter is denoted by an identifier, and prefixed by the `:` symbol (colon) in the text of the query string. The identifier name must follow the usual rules for identifiers specified in Section 4.4.1. Named parameters are case-sensitive.

Example:

```
SELECT c
FROM Customer c
WHERE c.status = :stat
```

A given named parameter may occur more than once in the query string.

4.7.5. Arithmetic Expressions

The arithmetic operators are:

- +, - unary
- *, / multiplication and division
- +, - addition and subtraction

Arithmetic operations use numeric promotion.

Arithmetic functions are described in Section 4.7.7.2.

4.7.6. String concatenation operator

The binary concatenation operator is `||`. Its operands must be string expressions.

4.7.7. Built-in String, Arithmetic, and Datetime Functional Expressions

The Jakarta Persistence query language includes the built-in functions described in Section 4.7.7.1, Section 4.7.7.2, Section 4.7.7.3, which may be used in the `SELECT`, `WHERE` or `HAVING` clause of a query. The invocation of predefined database functions and user-defined database functions is described in Section 4.7.9.

If the value of any argument to a functional expression is null or unknown, the value of the functional expression is unknown.

4.7.7.1. String Functions

```
functions_returning_strings ::=
    CONCAT(string_expression, string_expression {, string_expression}*) |
    SUBSTRING(string_expression,
              arithmetic_expression [, arithmetic_expression]) |
    TRIM([[trim_specification] [trim_character] FROM] string_expression) |
    LOWER(string_expression) |
```

```
        UPPER(string_expression) |
        REPLACE(string_expression, string_expression, string_expression) |
        LEFT(string_expression, arithmetic_expression) |
        RIGHT(string_expression, arithmetic_expression)
    trim_specification ::= LEADING | TRAILING | BOTH

    functions_returning_numerics ::=
        LENGTH(string_expression) |
        LOCATE(string_expression, string_expression[, arithmetic_expression])
```

The `CONCAT` function returns a string that is a concatenation of its arguments.

The second and third arguments of the `SUBSTRING` function denote the starting position and length of the substring to be returned. These arguments are integers. The third argument is optional. If it is not specified, the substring from the start position to the end of the string is returned. The first position of a string is denoted by 1. The `SUBSTRING` function returns a string.

The `TRIM` function trims the specified character from a string. If the character to be trimmed is not specified, it will be assumed to be space (or blank). The optional `trim_character` is a single-character string literal or a character-valued input parameter (i.e., `char` or `Character`) [15]. If a trim specification is not provided, it defaults to `BOTH`. The `TRIM` function returns the trimmed string.

The `LOWER` and `UPPER` functions convert a string to lower and upper case, respectively, with regard to the locale of the database. They return a string.

The `LEFT` and `RIGHT` functions return the leftmost or rightmost substring, respectively, of the first argument whose length is given by the second argument.

The `REPLACE` function replaces all occurrences within the first argument string of the second argument string with the third argument string.

The `LOCATE` function returns the position at which one string occurs within a second string, optionally ignoring any occurrences that begin before a specified character position in the second string. It returns the first character position within the second string (after the specified character position, if any) at which the first string occurs, as an integer, where the first character of the second string is denoted by 1. That is, the first argument is the string to be searched for; the second argument is the string to be searched in; the optional third argument is an integer representing the character position at which the search starts (by default, 1, the first character of the second string). If the first string does not occur within the second string, 0 is returned.[16]

The `LENGTH` function returns the length of the string in characters as an integer.

4.7.7.2. Arithmetic Functions

```
    functions_returning_numerics ::=
        ABS(arithmetic_expression) |
        CEILING(arithmetic_expression) |
        EXP(arithmetic_expression) |
        FLOOR(arithmetic_expression) |
        LN(arithmetic_expression) |
        MOD(arithmetic_expression, arithmetic_expression) |
        POWER(arithmetic_expression, arithmetic_expression) |
        ROUND(arithmetic_expression, arithmetic_expression) |
        SIGN(arithmetic_expression) |
        SQRT(arithmetic_expression) |
        SIZE(collection_valued_path_expression) |
        INDEX(identification_variable) |
        extract_datetime_field
```

The `ABS`, `CEILING`, and `FLOOR` functions accept a numeric argument and return a number (integer, float, or double) of the same type as the argument.

The `SIGN` function accepts a numeric argument and returns an integer.

The `SQRT`, `EXP`, and `LN` functions accept a numeric argument and return a double.

The `MOD` function accepts two integer arguments and returns an integer.

The `ROUND` function accepts a numeric argument and an integer argument and returns a number of the same type as the first argument.

The `POWER` function accepts two numeric arguments and returns a double.

Numeric arguments to these functions may correspond to the numeric Java object types as well as the primitive numeric types.

The `SIZE` function returns an integer value, the number of elements of the collection. If the collection is empty, the `SIZE` function evaluates to zero.

The `INDEX` function returns an integer value corresponding to the position of its argument in an ordered list. The `INDEX` function can only be applied to identification variables denoting types for which an order column has been specified.

In the following example, `studentWaitlist` is a list of students for which an order column has been specified:

```
SELECT w.name
FROM Course c JOIN c.studentWaitlist w
WHERE c.name = 'Calculus'
AND INDEX(w) = 0
```

4.7.7.3. Datetime Functions

```
functions_returning_datetime :=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP |
    LOCAL DATE |
    LOCAL TIME |
    LOCAL DATETIME |
    extract_datetime_part
```

The functions `LOCAL DATE`, `LOCAL TIME`, and `LOCAL DATETIME` return the value of the current date, time, or timestamp on the database server, respectively. Their types are `java.time.LocalDate`, `java.time.LocalTime`, and `java.time.LocalDateTime` respectively.

The functions `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP` return the value of the current date, time, or timestamp on the database server, respectively. Their types are `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp` respectively.

The `EXTRACT` function takes a datetime argument and one of the following field type identifiers: `YEAR`, `QUARTER`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `DATE`, `TIME`.

`EXTRACT` returns the value of the corresponding field or part of the datetime.

```
extract_datetime_field :=
    EXTRACT(datetime_field FROM datetime_expression)

datetime_field := identification_variable
```

For the following field type identifiers, `EXTRACT` returns an integer value:

- `YEAR` means the calendar year.
- `QUARTER` means the calendar quarter, numbered from 1 to 4.
- `MONTH` means the calendar month of the year, numbered from 1.
- `WEEK` means the ISO-8601 week number.
- `DAY` means the calendar day of the month, numbered from 1.
- `HOUR` means the hour of the day in 24-hour time, numbered from 0 to 23.
- `MINUTE` means the minute of the hour, numbered from 0 to 59.

For the `SECOND` field type identifier, `EXTRACT` returns a floating point value:

- `SECOND` means the second of the minute, numbered from 0 to 59, including a fractional part representing fractions of a second.

It is illegal to pass a datetime argument which does not have the given field type to `EXTRACT`.

```
extract_datetime_part :=
    EXTRACT(datetime_part FROM datetime_expression)

datetime_part := identification_variable
```

For the following field type identifiers, `EXTRACT` returns a part of the datetime value:

- `DATE` means the date part of a datetime.
- `TIME` means the time part of a datetime.

It is illegal to pass a datetime argument which does not have the given part to `EXTRACT`.

```
FROM Course c WHERE c.year = EXTRACT(YEAR FROM LOCAL DATE)
```

4.7.8. Typecasts

The `CAST` function converts an expression of one type to an expression of a different type.

```
string_cast_function::=
    CAST(scalar_expression AS STRING)
arithmetic_cast_function::=
    CAST(string_expression AS {INTEGER | LONG | FLOAT | DOUBLE})
```

The persistence provider is required to accept typecasts of the following forms:

- any scalar expression to `STRING`
- any string expression to `INTEGER`, `LONG`, `FLOAT`, or `DOUBLE`

Typecast expressions are evaluated by the database, with semantics that vary somewhat between different databases.

When a typecast occurs as a select expression, the result type of the select expression is:

- `java.lang.String` for a cast to `STRING`
- `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, or `java.lang.Double` for a cast to `INTEGER`, `LONG`, `FLOAT`, or `DOUBLE`, respectively

4.7.9. Invocation of Predefined and User-defined Database Functions

The invocation of functions other than the built-in functions of the Jakarta Persistence query language is supported by means of the `function_invocation` syntax. This includes the invocation of predefined database functions and user-

defined database functions.

```
function_invocation ::= FUNCTION(function_name {, function_arg}*)

function_arg ::=
    literal |
    state_valued_path_expression |
    input_parameter |
    scalar_expression
```

The `function_name` argument is a string that denotes the database function that is to be invoked. The arguments must be suitable for the database function that is to be invoked. The result of the function must be suitable for the invocation context.

The function may be a database-defined function or a user-defined function. The function may be a scalar function or an aggregate function.

Applications that use the `function_invocation` syntax are not portable across databases.

Example:

```
SELECT c
FROM Customer c
WHERE FUNCTION('hasGoodCredit', c.balance, c.creditLimit)
```

4.7.10. Case Expressions

The following forms of case expressions are supported: general case expressions, simple case expressions, coalesce expressions, and nullif expressions.[17]

```
case_expression ::=
    general_case_expression |
    simple_case_expression |
    coalesce_expression |
    nullif_expression

general_case_expression ::=
    CASE when_clause {when_clause}* {ELSE scalar_expression} END
when_clause ::= WHEN conditional_expression THEN scalar_expression

simple_case_expression ::=
    CASE case_operand simple_when_clause {simple_when_clause}*
    {ELSE scalar_expression}
    END
case_operand ::= state_valued_path_expression | type_discriminator
simple_when_clause ::= WHEN scalar_expression THEN scalar_expression

coalesce_expression ::= COALESCE(scalar_expression {, scalar_expression}+)

nullif_expression ::= NULLIF(scalar_expression, scalar_expression)
```

The `ELSE` branch of a case expression is optional. A case expression without an `ELSE` branch is interpreted as if it had the `ELSE` branch `ELSE NULL`.

Examples:

```
UPDATE Employee e
SET e.salary =
    CASE WHEN e.rating = 1 THEN e.salary * 1.1
         WHEN e.rating = 2 THEN e.salary * 1.05
         ELSE e.salary * 1.01
```

```
    END

UPDATE Employee e
SET e.salary =
    CASE e.rating WHEN 1 THEN e.salary * 1.1
                  WHEN 2 THEN e.salary * 1.05
                  ELSE e.salary * 1.01
    END

SELECT e.name,
    CASE TYPE(e) WHEN Exempt THEN 'Exempt'
                 WHEN Contractor THEN 'Contractor'
                 WHEN Intern THEN 'Intern'
                 ELSE 'NonExempt'
    END
FROM Employee e
WHERE e.dept.name = 'Engineering'

SELECT e.name,
       f.name,
       CONCAT(CASE WHEN f.annualMiles > 50000 THEN 'Platinum '
                   WHEN f.annualMiles > 25000 THEN 'Gold '
                   ELSE ''
              END,
       'Frequent Flyer')
FROM Employee e JOIN e.frequentFlierPlan f
```

4.7.11. Identifier and Version Functions

The `ID` and `VERSION` functions evaluate to the primary key or version, respectively, of their argument, which must be an identification variable assigned an entity abstract schema type or a path expression resolving to a one-to-one or many-to-one relationship field. For example, if `Person` has a primary key field named `ssn`, then `ID(person)` is a synonym for `person.ssn`.

```
entity_id_or_version_function ::= id_function | version_function
id_function ::=
    ID(general_identification_variable |
       single_valued_object_path_expression)
version_function ::=
    VERSION(general_identification_variable |
            single_valued_object_path_expression)
```

The result type of an `ID` or `VERSION` function expression is the primary key type or version type of the argument entity, respectively.

The result may be compared to an input parameter:

```
DELETE from Employee
WHERE id(this) = :id
  AND version(this) = :version
```

A persistence provider is not required to support the use of the `ID` function for entities with composite primary keys.

4.7.12. Entity Type Expressions and Literal Entity Types

An *entity type expression* can be used to restrict query polymorphism. The syntax of an entity type expression is as follows:

```
entity_type_expression ::=
    type_discriminator |
```

```
        entity_type_literal |
        input_parameter
    type_discriminator ::=
        TYPE(general_identification_variable |
             single_valued_object_path expression |
             input_parameter)
```

The `TYPE` operator returns the exact type of its argument, which must be an identification variable assigned an entity abstract schema type, a path expression resolving to a one-to-one or many-to-one relationship field, or an input parameter.

An `entity_type_literal` specifies a literal entity type by its entity name defined above in Section 4.3.1.

For an input parameter, the entity type must be specified by calling `Query.setParameter()` with the `java.lang.Class` object representing the entity class.

Examples:

```
SELECT e
FROM Employee e
WHERE TYPE(e) IN (Exempt, Contractor)

SELECT e
FROM Employee e
WHERE TYPE(e) IN (:empType1, :empType2)

SELECT e
FROM Employee e
WHERE TYPE(e) IN :empTypes

SELECT TYPE(e)
FROM Employee e
WHERE TYPE(e) <> Exempt
```

4.7.13. Numeric Expressions and Type Promotion

Every numeric expression in a query is assigned a Java numeric type according to the following rules:

- An expression that corresponds to a persistent state field is of the same type as that persistent state field.
- An expression that corresponds to one of arithmetic functions described in Section 4.7.7.2 is of the type defined by Section 4.7.7.2.
- An expression that corresponds to one of an aggregate functions described in Section 4.9.5 is of the type defined by Section 4.9.5.

For a `CASE` expression, `COALESCE` expression, `NULLIF` expression, or arithmetic operator expression (+, -, *, /), the numeric type is determined by its operand types, and by the following rules[18].

- If there is an operand of type `Double` or `double`, the expression is of type `Double`;
- otherwise, if there is an operand of type `Float` or `float`, the expression is of type `Float`;
- otherwise, if there is an operand of type `BigDecimal`, the expression is of type `BigDecimal`;
- otherwise, if there is an operand of type `BigInteger`, the expression is of type `BigInteger`, unless the operator is `/` (division), in which case the expression type is not defined here;
- otherwise, if there is an operand of type `Long` or `long`, the expression is of type `Long`, unless the operator is `/` (division), in which case the expression type is not defined here;
- otherwise, if there is an operand of integral type, the expression is of type `Integer`, unless the operator is `/` (division), in which case the expression type is not defined here.

Users should note that the semantics of the SQL division operation are not standard across databases. In particular, when both operands are of integral types, the result of the division operation will be an integral type in some databases, and an non-integral type in others. Portable applications should not assume a particular result type.

For numeric expressions occurring in the `SELECT` clause, these rules determine the Java object type returned in the query result list.

4.8. GROUP BY, HAVING

The `GROUP BY` construct enables the aggregation of result values according to a set of properties. The `HAVING` construct enables conditions to be specified that further restrict the query result. Such conditions are restrictions upon the groups.

The syntax of the `GROUP BY` and `HAVING` clauses is as follows:

```
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression | identification_variable

having_clause ::= HAVING conditional_expression
```

If a query contains both a `WHERE` clause and a `GROUP BY` clause, the effect is that of first applying the where clause, and then forming the groups and filtering them according to the `HAVING` clause. The `HAVING` clause causes those groups to be retained that satisfy the condition of the `HAVING` clause.

The requirements for the `SELECT` clause when `GROUP BY` is used follow those of SQL: namely, any item that appears in the `SELECT` clause (other than as an aggregate function or as an argument to an aggregate function) must also appear in the `GROUP BY` clause. In forming the groups, null values are treated as the same for grouping purposes.

Grouping by an entity is permitted. In this case, the entity must contain no serialized state fields or lob-valued state fields that are eagerly fetched. Grouping by an entity that contains serialized state fields or lob-valued state fields is not portable, since the implementation is permitted to eagerly fetch fields or properties that have been specified as `LAZY`.

Grouping by embeddables is not supported.

The `HAVING` clause is used to filter over the groups, and can contain aggregate functions over attributes included in the groups and/or functions or other query language operators over the attributes that are used for grouping. It is not required that an aggregate function used in the `HAVING` clause also be used in the `SELECT` clause.

If there is no `GROUP BY` clause and the `HAVING` clause is used, the result is treated as a single group, and the select list can only consist of aggregate functions. The use of `HAVING` in the absence of `GROUP BY` is not required to be supported by an implementation of this specification. Portable applications should not rely on `HAVING` without the use of `GROUP BY`.

Examples:

```
SELECT c.status, AVG(c.filledOrderCount), COUNT(c)
FROM Customer c
GROUP BY c.status
HAVING c.status IN (1, 2)

SELECT c.country, COUNT(c)
FROM Customer c
GROUP BY c.country
HAVING COUNT(c) > 30

SELECT c, COUNT(o)
```

```
FROM Customer c JOIN c.orders o
GROUP BY c
HAVING COUNT(o) >= 5
```

4.9. SELECT Clause

The `SELECT` clause specifies the query result, as a list of items to be returned by the query.

The `SELECT` clause can contain one or more of the following elements:

- an identification variable that ranges over an abstract schema type,
- a single-valued path expression,
- a scalar expression,
- an aggregate expression,
- a constructor expression.

The `SELECT` clause has the following syntax:

```
select_clause ::= SELECT [DISTINCT] select_item {, select_item}*
select_item ::= select_expression [[AS] result_variable]
select_expression ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable |
    OBJECT(identification_variable) |
    constructor_expression
constructor_expression ::=
    NEW constructor_name (constructor_item {, constructor_item}*)
constructor_item ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable
aggregate_expression ::=
    {AVG | MAX | MIN | SUM} ([DISTINCT] state_valued_path_expression) |
     COUNT ([DISTINCT] identification_variable | state_valued_path_expression |
         single_valued_object_path_expression) |
     function_invocation
```

For example:

```
SELECT c.id, c.status
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

In the following example, `videoInventory` is a Map from the entity `Movie` to the number of copies in stock:

```
SELECT v.location.street, KEY(i).title, VALUE(i)
FROM VideoStore v JOIN v.videoInventory i
WHERE v.location.zipcode = '94301' AND VALUE(i) > 0
```

Note that the `SELECT` clause must be specified to return only single-valued expressions. The query below is therefore not valid:

```
SELECT o.lineItems FROM Order AS o
```

The `DISTINCT` keyword specifies that duplicate values must be eliminated from the query result.

If `DISTINCT` is not specified, duplicate values are not eliminated.

The result of `DISTINCT` over embeddable objects or map `entry` results is undefined.

Standalone identification variables in the `SELECT` clause may optionally be qualified by the `OBJECT` operator.[19] The `SELECT` clause must not use the `OBJECT` operator to qualify path expressions.

A `result_variable` assigns a name to a `select_item` in the query result. The `result variable` must be a valid identifier, as defined in Section 4.4.1, must not be a reserved identifier, and must not collide with any identification variable declared in the `FROM` clause. A result variable may be used to refer to an element of the select clause from an item in the `ORDER BY` clause, as specified in Section 4.10. Like identification variables, result variables are case-insensitive.

Example:

```
SELECT c, COUNT(l) AS itemCount
FROM Customer c JOIN c.orders o JOIN o.lineItems l
WHERE c.address.state = 'CA'
GROUP BY c
ORDER BY itemCount
```

The `SELECT` clause is optional. A query with a missing `SELECT` clause is interpreted as if it had the following single-item `SELECT` clause: `select this`, where `this` is the implicit identification variable.

Thus, the following queries are equivalent:

```
FROM Order
WHERE customer.lastname = 'Smith'
  AND customer.firstname= 'John'
```

```
SELECT this
FROM Order
WHERE this.customer.lastname = 'Smith'
  AND this.customer.firstname= 'John'
```

```
SELECT ord
FROM Order AS ord
WHERE ord.customer.lastname = 'Smith'
  AND ord.customer.firstname= 'John'
```

If the implicit identification variable has not been assigned an abstract schema type, the `SELECT` clause is required.

4.9.1. Result Type of the SELECT Clause

The type of the query result specified by the `SELECT` clause of a query is an entity abstract schema type, a state field type, the result of a scalar expression, the result of an aggregate function, the result of a construction operation, or some sequence of these.

The result type of the `SELECT` clause is defined by the result types of the select expressions contained in it. When multiple select expressions are used in the `SELECT` clause, the elements in this result correspond in order to the order of their specification in the `SELECT` clause and in type to the result types of each of the select expressions.

The type of the result of a `select_expression` is as follows:

- The result type of an `identification_variable` is the type of the entity object or embeddable object to which the identification variable corresponds. The type of an `identification_variable` that refers to an entity abstract schema type is the type of the entity to which that identification variable corresponds or a subtype as determined by the object/relational mapping.

- The result type of a `single_valued_path_expression` that is a `state_field_path_expression` is the same type as the corresponding state field of the entity or embeddable class. If the state field of the entity is a primitive type, the result type is the corresponding object type.
- The result type of a `single_valued_path_expression` that is a `single_valued_object_path_expression` is the type of the entity object or embeddable object to which the path expression corresponds. A `single_valued_object_path_expression` that results in an entity object will result in an entity of the type of the relationship field or the subtype of the relationship field of the entity object as determined by the object/relational mapping.
- The result type of a `single_valued_path_expression` that is an `identification_variable` to which the `KEY` or `VALUE` function has been applied is determined by the type of the map key or value respectively, as defined by the above rules.
- The result type of a `single_valued_path_expression` that is an `identification_variable` to which the `ENTRY` function has been applied is `java.util.Map.Entry`, where the key and value types of the map entry are determined by the above rules as applied to the map key and map value respectively.
- The result type of a `scalar_expression` is the type of the scalar value to which the expression evaluates. The result type of a numeric `scalar_expression` is defined in Section 4.7.13.
- The result type of an `entity_type_expression` scalar expression is the Java class to which the resulting abstract schema type corresponds.
- The result type of `aggregate_expression` is defined in Section 4.9.5.
- The result type of a `constructor_expression` is the type of the class for which the constructor is defined. The types of the arguments to the constructor are defined by the above rules.

4.9.2. Constructor Expressions in the SELECT Clause

A constructor may be used in the `SELECT` list to return an instance of a Java class. The specified class is not required to be an entity or to be mapped to the database. The constructor name must be fully qualified.

If an entity class name is specified as the constructor name in the `SELECT NEW` clause, the resulting entity instances will be in either the new or the detached state, depending on whether a primary key is retrieved for the constructed object.

If a `single_valued_path_expression` or `identification_variable` that is an argument to the constructor references an entity, the resulting entity instance referenced by that `single_valued_path_expression` or `identification_variable` will be in the managed state.

For example,

```
SELECT NEW com.acme.example.CustomerDetails(c.id, c.status, o.count)
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

4.9.3. Null Values in the Query Result

If the result of a query corresponds to an association field or state field whose value is null, that null value is returned in the result of the query method. The `IS NOT NULL` construct can be used to eliminate such null values from the result set of the query.

Note, however, that state field types defined in terms of Java numeric primitive types cannot produce `NULL` values in the query result. A query that returns such a state field type as a result type must not return a null value.

4.9.4. Embeddables in the Query Result

If the result of a query corresponds to an identification variable or state field whose value is an embeddable, the embeddable instance returned by the query is not in the managed state (i.e., it is not part of the state of any managed

entity).

In the following example, the `Address` instances returned by the query will reference `Phone` instances. While the `Phone` instances will be managed, the `Address` instances referenced by the `addr` result variable will not be. Modifications to these embeddable instances will have no effect on persistent state.

```
@Entity
public class Employee {
    @Id
    int id;

    Address address;

    // ...
}

@Embeddable
public class Address {
    String street;

    // ...

    @OneToOne
    Phone phone; // fetch=EAGER
}

@Entity
public class Phone {
    @Id
    int id;

    // ...

    @OneToOne(mappedBy="address.phone")
    Employee emp; // fetch=EAGER
}
```

```
SELECT e.address AS addr
FROM Employee e
```

4.9.5. Aggregate Functions in the SELECT Clause

The result of a query may be the result of an aggregate function applied to a path expression.

The following aggregate functions can be used in the `SELECT` clause of a query: `AVG`, `COUNT`, `MAX`, `MIN`, `SUM`, aggregate functions defined in the database.

For all aggregate functions except `COUNT`, the path expression that is the argument to the aggregate function must terminate in a state field. The path expression argument to `COUNT` may terminate in either a state field or a association field, or the argument to `COUNT` may be an identification variable.

Arguments to the functions `SUM` and `AVG` must be numeric. Arguments to the functions `MAX` and `MIN` must correspond to orderable state field types (i.e., numeric types, string types, character types, or date types).

The Java type that is contained in the result of a query using an aggregate function is determined as follows:

- `COUNT` returns `Long`.
- `MAX`, `MIN` return the type of the state field to which they are applied.
- `AVG` returns `Double`.

- `SUM` returns `Long` when applied to state fields of integral types (other than `BigInteger`); `Double` when applied to state fields of floating point types; `BigInteger` when applied to state fields of type `BigInteger`; and `BigDecimal` when applied to state fields of type `BigDecimal`.

Null values are eliminated before the aggregate function is applied, regardless of whether the keyword `DISTINCT` is specified.

If `SUM`, `AVG`, `MAX`, or `MIN` is used, and there are no values to which the aggregate function can be applied, the result of the aggregate function is `NULL`.

If `COUNT` is used, and there are no values to which `COUNT` can be applied, the result of the aggregate function is `0`.

The argument to an aggregate function may be preceded by the keyword `DISTINCT` to specify that duplicate values are to be eliminated before the aggregate function is applied.[20]

The use of `DISTINCT` with `COUNT` is not supported for arguments of embeddable types or map entry types.

The invocation of aggregate database functions, including user defined functions, is supported by means of the `FUNCTION` operator. See Section 4.7.9.

The following query returns the average order quantity:

```
SELECT AVG(o.quantity) FROM Order o
```

The following query returns the total cost of the items that John Smith has ordered.

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Smith' AND c.firstname = 'John'
```

The following query returns the total number of orders.

```
SELECT COUNT(o) FROM Order o
```

The following query counts the number of items in John Smith's order for which prices have been specified.

```
SELECT COUNT(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Smith' AND c.firstname = 'John'
```

Note that this is equivalent to:

```
SELECT COUNT(l)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Smith' AND c.firstname = 'John' AND l.price IS NOT NULL
```

4.10. ORDER BY Clause

The `ORDER BY` clause specifies how the results of a query should be sorted. The syntax of the `ORDER BY` clause is:

```
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= orderby_expression [ASC | DESC] [NULLS {FIRST | LAST}]
orderby_expression ::=
    state_field_path_expression |
    general_identification_variable |
    result_variable |
    scalar_expression
```

The `ORDER BY` clause specifies a list of items. Each `orderby_expression` must be one of the following:

1. A `state_field_path_expression` evaluating to an orderable state field of an entity or embeddable class abstract schema type designated in the `SELECT` clause by either:
   - a `general_identification_variable`, or
   - a `single_valued_object_path_expression`.
2. A `state_field_path_expression` evaluating to the same state field of the same entity or embeddable abstract schema type as a `state_field_path_expression` in the `SELECT` clause.
3. A `general_identification_variable` evaluating to the same map field of the same entity or embeddable abstract schema type as a `general_identification_variable` in the `SELECT` clause.
4. A reference to a `result_variable` declared by an orderable item in the `SELECT` clause. The orderable item must be an `aggregate_expression`, a `scalar_expression`, or a `state_field_path_expression`.
5. Any `scalar_expression` involving only `state_field_path_expressions` which would be allowed according to items 1 or 2 above.

Depending on the database, arbitrary scalar expressions may not be allowed in the `ORDER BY` clause. Therefore, applications which require portability between databases should not depend on the use of a scalar expression in `ORDER BY` if it is only permitted by item 5.

For example, the four queries below are legal.

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity DESC, o.totalcost

SELECT o.quantity, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, a.zipcode

SELECT o.quantity, o.cost*1.08 AS taxedCost, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA' AND a.county = 'Santa Clara'
ORDER BY o.quantity, taxedCost, a.zipcode

SELECT AVG(o.quantity) as q, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
GROUP BY a.zipcode
ORDER BY q DESC
```

The following query is legal, but might not be supported on every database.

```
SELECT c, o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY UPPER(c.lastname), UPPER(c.firstname)
```

The following two queries are *not* legal because the `orderby_item` is not reflected in the `SELECT` clause of the query.

```
SELECT p.product_name
FROM Order o JOIN o.lineItems l JOIN l.product p JOIN o.customer c
WHERE c.lastname = 'Smith' AND c.firstname = 'John'
ORDER BY p.price

SELECT p.product_name
FROM Order o, IN(o.lineItems) l JOIN o.customer c
```

```
WHERE c.lastname = 'Smith' AND c.firstname = 'John'
ORDER BY o.quantity
```

The keyword `ASC` specifies that ascending ordering is used for the associated `orderby_item`; the keyword `DESC` specifies that descending ordering is used. If neither keyword is explicitly specified, ascending ordering is the default.

The interpretation of ascending or descending order is determined by the database, but, in general:

- ascending order for numeric values means smaller values first, while descending order means larger values first, and
- strings are sorted lexicographically, using a database-dependent collation.

The keyword `NULLS` specifies the ordering of null values, either `FIRST` or `LAST`.

- `FIRST` means that results are sorted so that all null values occur before all non-null values.
- `LAST` means that results are sorted so that all null values occur after all non-null values.

If `NULLS` is not specified, the database determines whether null values occur first or last.

Items occurring earlier in the `ORDER BY` clause take precedence. That is, an item occurring later in the `ORDER BY` clause is only used to resolve "ties" between results which cannot be unambiguously ordered using only earlier items.

The order of query results must be preserved in the result list or stream returned by a query execution method when an `ORDER BY` clause is specified.

4.11. Bulk Update and Delete Operations

Bulk update and delete operations apply to entities of a single entity class (together with its subclasses, if any). Only one entity abstract schema type may be specified in the `FROM` or `UPDATE` clause.

The syntax of these operations is as follows:

```
update_statement ::= update_clause [where_clause]
update_clause ::= UPDATE entity_name [[AS] identification_variable]
                  SET update_item {, update_item}*
update_item ::= [identification_variable.]{single_valued_embeddable_object_field.}*
    {state_field | single_valued_object_field} = new_value
new_value ::=
    scalar_expression |
    simple_entity_expression |
    NULL

delete_statement ::= delete_clause [where_clause]
delete_clause ::= DELETE FROM entity_name [[AS] identification_variable]
```

The syntax of the `WHERE` clause is described in Section 4.5.

A delete operation only applies to entities of the specified class and its subclasses. It does not cascade to related entities.

The `new_value` specified for an update operation must be compatible in type with the field to which it is assigned.

Bulk update maps directly to a database update operation, bypassing optimistic locking checks. Portable applications must manually update the value of the version column, if desired, and/or manually validate the value of the version column.

After execution of a bulk update or delete operation, the persistence provider is not required to resynchronize state held in memory with the effects of the operation on data held in the database.

> Caution should be used when executing bulk update or delete operations because they may result in inconsistencies between the database and the entities in the active persistence context. In general, bulk update and delete operations should only be performed within a transaction in a new persistence context or before fetching or accessing entities whose state might be affected by such operations.

However, the usual rules specified in Section 3.15.3 apply to the processing of a bulk update or delete operation executed within a transaction via a persistence context joined to the transaction. In particular, when `FlushMode.AUTO` is in effect, the persistence provider must ensure that every modification to the state of every entity associated with the persistence context which could possibly alter the effects of the bulk update or delete operation is visible to the processing of the operation.

Examples:

```
DELETE
FROM Customer c
WHERE c.status = 'inactive'

DELETE
FROM Customer c
WHERE c.status = 'inactive'
    AND c.orders IS EMPTY

UPDATE Customer c
SET c.status = 'outstanding'
WHERE c.balance < 10000

UPDATE Employee e
SET e.address.building = 22
WHERE e.address.building = 14
    AND e.address.city = 'Santa Clara'
    AND e.project = 'Jakarta EE'
```

4.12. BNF

BNF notation summary:

- `{ ··· }` grouping
- `[ ··· ]` optional constructs
- `*` zero or more
- `+` one or more
- `|` alternates

The following is the BNF for the Jakarta Persistence query language.

```
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= union
union ::= intersection | union {UNION [ALL] | EXCEPT [ALL]} intersection
intersection ::= query_expression | intersection INTERSECT [ALL] query_expression
query_expression ::= select_query | (union)
select_query ::= [select_clause] from_clause [where_clause] [groupby_clause]
    [having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::=
    FROM {this_implicit_variable | identification_variable_declarations}
this_implicit_variable ::= entity_name
identification_variable_declarations ::=
    identification_variable_declaration
```

```
    {, {identification_variable_declaration | collection_member_declaration}}*
identification_variable_declaration ::= range_variable_declaration {join | fetch_join}*
range_variable_declaration ::= entity_name [AS] identification_variable
join ::= range_join | path_join
range_join ::= join_spec range_variable_declaration [join_condition]
path_join ::=
    join_spec join_association_path_expression [AS] identification_variable [join_condition]
fetch_join ::= join_spec FETCH join_association_path_expression
join_spec ::= [INNER | LEFT [OUTER]] JOIN
join_condition ::= ON conditional_expression
join_association_path_expression ::=
    join_collection_valued_path_expression |
    join_single_valued_path_expression |
    TREAT(join_collection_valued_path_expression AS subtype) |
    TREAT(join_single_valued_path_expression AS subtype)
join_collection_valued_path_expression ::=
    [identification_variable.]{single_valued_embeddable_object_field.}* collection_valued_field
join_single_valued_path_expression ::=
    [identification_variable.]{single_valued_embeddable_object_field.}* single_valued_object_field
collection_member_declaration ::=
    IN (collection_valued_path_expression) [AS] identification_variable
qualified_identification_variable ::=
    map_field_identification_variable |
    ENTRY(identification_variable)
map_field_identification_variable ::=
    KEY(identification_variable) |
    VALUE(identification_variable)
single_valued_path_expression ::=
    qualified_identification_variable |
    TREAT(qualified_identification_variable AS subtype) |
    state_field_path_expression |
    single_valued_object_path_expression
general_identification_variable ::=
    identification_variable |
    map_field_identification_variable
general_subpath ::= simple_subpath | treated_subpath{.single_valued_object_field}*
simple_subpath ::=
    general_identification_variable |
    general_identification_variable{.single_valued_object_field}*
treated_subpath ::= TREAT(general_subpath AS subtype)
state_field_path_expression ::= [general_subpath.]state_field
state_valued_path_expression ::=
    state_field_path_expression | general_identification_variable
single_valued_object_path_expression ::=
    general_subpath.single_valued_object_field
collection_valued_path_expression ::= general_subpath.{collection_valued_field}
update_clause ::= UPDATE entity_name [[AS] identification_variable]
    SET update_item {, update_item}*
update_item ::= [identification_variable.]{single_valued_embeddable_object_field.}*
    {state_field | single_valued_object_field} = new_value
new_value ::=
    scalar_expression |
    simple_entity_expression |
    NULL
delete_clause ::= DELETE FROM entity_name [[AS] identification_variable]
select_clause ::= SELECT [DISTINCT] select_item {, select_item}*
select_item ::= select_expression [[AS] result_variable]
select_expression ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable |
    OBJECT(identification_variable) |
    constructor_expression
constructor_expression ::=
    NEW constructor_name (constructor_item {, constructor_item}*)
constructor_item ::=
```

```
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable
aggregate_expression ::=
    {AVG | MAX | MIN | SUM} ([DISTINCT] state_valued_path_expression) |
    COUNT ([DISTINCT] identification_variable | state_valued_path_expression |
        single_valued_object_path_expression) |
    function_invocation
where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression | identification_variable
having_clause ::= HAVING conditional_expression
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= orderby_expression [ASC | DESC] [NULLS {FIRST | LAST}]
orderby_expression ::=
    state_field_path_expression |
    general_identification_variable |
    result_variable |
    scalar_expression
subquery ::= simple_select_clause subquery_from_clause [where_clause]
    [groupby_clause] [having_clause]
subquery_from_clause ::=
    FROM subselect_identification_variable_declaration
        {, subselect_identification_variable_declaration | collection_member_declaration}*
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    derived_path_expression [AS] identification_variable {join}* |
    derived_collection_member_declaration
derived_path_expression ::=
    general_derived_path.single_valued_object_field |
    general_derived_path.collection_valued_field
general_derived_path ::=
    simple_derived_path |
    treated_derived_path{.single_valued_object_field}*
simple_derived_path ::= superquery_identification_variable{.single_valued_object_field}*
treated_derived_path ::= TREAT(general_derived_path AS subtype)
derived_collection_member_declaration ::=
    IN superquery_identification_variable.{single_valued_object_field.}*collection_valued_field
simple_select_clause ::= SELECT [DISTINCT] simple_select_expression
simple_select_expression::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable
scalar_expression ::=
    arithmetic_expression |
    string_expression |
    enum_expression |
    datetime_expression |
    boolean_expression |
    case_expression |
    entity_type_expression |
    entity_id_or_version_function
conditional_expression ::= conditional_term | conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::=
    comparison_expression |
    between_expression |
    in_expression |
    like_expression |
    null_comparison_expression |
    empty_collection_comparison_expression |
    collection_member_expression |
    exists_expression
```

```
between_expression ::=
    arithmetic_expression [NOT] BETWEEN
        arithmetic_expression AND arithmetic_expression |
    string_expression [NOT] BETWEEN string_expression AND string_expression |
    datetime_expression [NOT] BETWEEN datetime_expression AND datetime_expression
in_expression ::=
    {state_valued_path_expression | type_discriminator} [NOT] IN
        {(in_item{, in_item}*) | (subquery) | collection_valued_input_parameter}
in_item ::= literal | single_valued_input_parameter
like_expression ::=
    string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]
null_comparison_expression ::=
    {single_valued_path_expression | input_parameter} IS [NOT] NULL
empty_collection_comparison_expression ::=
    collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_or_value_expression
    [NOT] MEMBER [OF] collection_valued_path_expression
entity_or_value_expression ::=
    single_valued_object_path_expression |
    state_field_path_expression |
    simple_entity_or_value_expression
simple_entity_or_value_expression ::=
    identification_variable |
    input_parameter |
    literal
exists_expression ::= [NOT] EXISTS (subquery)
all_or_any_expression ::= {ALL | ANY | SOME} (subquery)
comparison_expression ::=
    string_expression comparison_operator {string_expression | all_or_any_expression} |
    boolean_expression {= | <>} {boolean_expression | all_or_any_expression} |
    enum_expression {= | <>} {enum_expression | all_or_any_expression} |
    datetime_expression comparison_operator
        {datetime_expression | all_or_any_expression} |
    entity_expression {= | <>} {entity_expression | all_or_any_expression} |
    arithmetic_expression comparison_operator {arithmetic_expression | all_or_any_expression} |
    entity_id_or_version_function {= | <>} input_parameter |
    entity_type_expression {= | <>} entity_type_expression}
comparison_operator ::= = | > | >= | < | <= | <>
arithmetic_expression ::=
    arithmetic_term | arithmetic_expression {+ | -} arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term {* | /} arithmetic_factor
arithmetic_factor ::= [{+ | -}] arithmetic_primary
arithmetic_primary ::=
    state_valued_path_expression |
    numeric_literal |
    (arithmetic_expression) |
    input_parameter |
    functions_returning_numerics |
    aggregate_expression |
    case_expression |
    function_invocation |
    arithmetic_cast_function |
    (subquery)
string_expression ::=
    state_valued_path_expression |
    string_literal |
    input_parameter |
    functions_returning_strings |
    aggregate_expression |
    case_expression |
    function_invocation |
    string_cast_function |
    string_expression || string_expression |
    (subquery)
datetime_expression ::=
    state_valued_path_expression |
    input_parameter |
```

```
    functions_returning_datetime |
    aggregate_expression |
    case_expression |
    function_invocation |
    date_time_timestamp_literal |
    (subquery)
boolean_expression ::=
    state_valued_path_expression |
    boolean_literal |
    input_parameter |
    case_expression |
    function_invocation |
    (subquery)
enum_expression ::=
    state_valued_path_expression |
    enum_literal |
    input_parameter |
    case_expression |
    (subquery)
entity_expression ::= single_valued_object_path_expression | simple_entity_expression
simple_entity_expression ::= identification_variable | input_parameter
entity_type_expression ::=
    type_discriminator |
    entity_type_literal |
    input_parameter
type_discriminator ::=
    TYPE(general_identification_variable |
        single_valued_object_path_expression |
        input_parameter)
arithmetic_cast_function::=
    CAST(string_expression AS {INTEGER | LONG | FLOAT | DOUBLE})
functions_returning_numerics ::=
    LENGTH(string_expression) |
    LOCATE(string_expression, string_expression[, arithmetic_expression]) |
    ABS(arithmetic_expression) |
    CEILING(arithmetic_expression) |
    EXP(arithmetic_expression) |
    FLOOR(arithmetic_expression) |
    LN(arithmetic_expression) |
    SIGN(arithmetic_expression) |
    SQRT(arithmetic_expression) |
    MOD(arithmetic_expression, arithmetic_expression) |
    POWER(arithmetic_expression, arithmetic_expression) |
    ROUND(arithmetic_expression, arithmetic_expression) |
    SIZE(collection_valued_path_expression) |
    INDEX(identification_variable) |
    extract_datetime_field
functions_returning_datetime ::=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP |
    LOCAL DATE |
    LOCAL TIME |
    LOCAL DATETIME |
    extract_datetime_part
string_cast_function::=
    CAST(scalar_expression AS STRING)
functions_returning_strings ::=
    CONCAT(string_expression, string_expression{, string_expression}*) |
    SUBSTRING(string_expression, arithmetic_expression[, arithmetic_expression]) |
    TRIM([[trim_specification] [trim_character] FROM] string_expression) |
    LOWER(string_expression) |
    UPPER(string_expression)
trim_specification ::= LEADING | TRAILING | BOTH
function_invocation ::= FUNCTION(function_name{, function_arg}*)
extract_datetime_field :=
    EXTRACT(datetime_field FROM datetime_expression)
```

```
datetime_field := identification_variable
extract_datetime_part :=
    EXTRACT(datetime_part FROM datetime_expression)
datetime_part := identification_variable
function_arg ::=
    literal |
    state_valued_path_expression |
    input_parameter |
    scalar_expression
entity_id_or_version_function ::= id_function | version_function
id_function ::=
    ID(general_identification_variable |
       single_valued_object_path_expression)
version_function ::=
    VERSION(general_identification_variable |
            single_valued_object_path_expression)
case_expression ::=
    general_case_expression |
    simple_case_expression |
    coalesce_expression |
    nullif_expression
general_case_expression::= CASE when_clause {when_clause}* {ELSE scalar_expression} END
when_clause ::= WHEN conditional_expression THEN scalar_expression
simple_case_expression ::=
    CASE case_operand simple_when_clause {simple_when_clause}*
    {ELSE scalar_expression}
    END
case_operand ::= state_valued_path_expression | type_discriminator
simple_when_clause ::= WHEN scalar_expression THEN scalar_expression
coalesce_expression ::= COALESCE(scalar_expression{, scalar_expression}+)
nullif_expression::= NULLIF(scalar_expression, scalar_expression)
```

[1] As in SQL, the `INTERSECT` and `INTERSECT ALL` operations have higher precedence than `UNION`, `UNION ALL`, `EXCEPT`, and `EXCEPT ALL`.

[2] The unqualified name of a nested entity class `org.example.Outer.Inner` is just `Inner`.

[3] This chapter uses the convention that reserved identifiers appear in upper case in the examples and BNF for the language.

[4] `BIT_LENGTH`, `CHAR_LENGTH`, `CHARACTER_LENGTH`, `POSITION`, and `UNKNOWN` are not currently used: they are reserved for future use.

[5] A range variable never designates an embeddable class abstract schema type.

[6] Note that use of `VALUE` is optional, as an identification variable referring to an association of type `java.util.Map` is of the abstract schema type of the map value. (See Section 4.4.2.)

[7] Support for right outer joins and full outer joins is under consideration for inclusion in a future version of this specification.

[8] The implementation is not expected to perform such query operations involving such fields in memory rather than in the database.

[9] Note that queries that contain subqueries on both sides of a comparison operation are not portable across all databases.

[10] Note that use of a collection-valued input parameter might prevent precompilation of the query.

[11] Refer to [2] for a more precise characterization of these rules.

[12] The use of the reserved word `OF` is optional in this expression.

[13] Subqueries are restricted to the `WHERE` and `HAVING` clauses in this release. Support for subqueries in the `FROM` clause will be considered in a later release of this specification.

[14] Note that expressions involving aggregate operators must not be used in the `WHERE` clause.

[15] Note that not all databases support the use of a trim character other than the space character; use of this argument may result in queries that are not portable.

[16] Note that not all databases support the use of the third argument to `LOCATE`; use of this argument may result in queries that are not portable.

[17] Note that not all databases support the use of SQL case expressions. The use of case expressions may result in queries that are not portable to such databases.

[18] For a general or simple `CASE` expression, the operands are the scalar expressions in the `THEN` and `ELSE` clauses.

[19] Note that the keyword `OBJECT` is not required. It is preferred that it be omitted for new queries.

[20] It is legal to specify `DISTINCT` with `MAX` or `MIN`, but it does not affect the result.

Chapter 5. Metamodel API

This specification provides a set of interfaces for dynamically accessing a metamodel representing the managed classes of a persistence unit. The metamodel API is found in Appendix D. Instances of metamodel types may be obtained either:

- via programmatic lookup using an instance of the interface `Metamodel` (found in Section D.1) obtained from the `EntityManagerFactory` or `EntityHandler` by calling `getMetamodel()`, or
- in a typesafe way, using *static metamodel classes*.

A static metamodel class is a class with static members providing direct typesafe access to metamodel objects representing the persistent members of a given managed class.

5.1. Static Metamodel Classes

A set of static metamodel classes corresponding to the managed classes of a persistence unit can be generated using an annotation processor or may be created by the application developer.

In the typical case, an annotation processor is used to generate static metamodel classes corresponding to the entities, mapped superclasses, and embeddable classes in the persistence unit, along with other types eligible for metamodel generation as specified below. A static metamodel for a managed class models the persistent state and relationships of the managed class and provides typesafe access to its metamodel objects.

For portability, an annotation processor should generate a *canonical metamodel* as specified in the next section.

> Implementations of this specification are not required to support the use of metamodel classes which do not adhere to this canonical form. An application that uses non-canonical metamodel classes is not portable.

5.1.1. Bootstrapping the Static Metamodel

When the entity manager factory for a persistence unit is created, it is the responsibility of the persistence provider to initialize the state of the static metamodel classes representing managed classes belonging to the persistence unit. Any generated metamodel classes must be accessible on the classpath.

An application must not access members of a static metamodel class before creation of an entity manager factory for a persistence unit to which the corresponding managed class belongs. If the application accesses a member of a static metamodel class before the entity manager factory is created, the behavior is undefined.

If a managed class belongs to multiple persistence units, and XML metadata specifies conflicting information about the entity class, use of the static metamodel for that managed class results in undefined behavior.

5.2. Canonical Metamodel

The canonical metamodel classes must be generated by the annotation processor with the standard structure specified below. The annotation processor might generate additional program elements, but it must at minimum generate the program elements specified here. Applications which require portability between different annotation processors should avoid the use of generated program elements other than those defined by this specification.

A class is considered *eligible* for generation of a metamodel class if it belongs to the persistence unit and:

- it is annotated with a *discoverable annotation type*, as defined in Section 8.2.1.8.1, or

- it has a method annotated with one of the two static query annotations, `@JakartaQuery` or `@NativeQuery`, as defined in Appendix F.

This specification does not require that the annotation processor generate a canonical metamodel class for a type which is declared as a managed type only in XML and which is not annotated with a discoverable annotation type.

5.2.1. Canonical Metamodel Classes

For every eligible class in the persistence unit, a corresponding metamodel class is generated as follows:

- For each eligible class `X` in package `p`, a metamodel class `X_` is created in the package `p`.[1]
- The name of the metamodel class is derived from the name of the eligible class by appending "_" to the name of the eligible class.
- The metamodel class `X_` must be annotated with the `StaticMetamodel` annotation found in Section D.2.[2]

> Implementations of this specification are not required to resolve naming collisions resulting from the faithful application of the rules specified here or below when generating canonical metamodel classes.

5.2.2. Canonical Metamodel for Managed Classes

The metamodel class for a managed class `X` is generated as follows:

- If a managed class `X` extends another class `S`, where `S` is the most derived managed class extended by `X`, then the metamodel class `X_` must extend the metamodel class `S_` created for `S`.
- The metamodel class for a managed class must contain a field declaration as follows:

  ```
  public static volatile jakarta.persistence.metamodel.T<X> class_;
  ```

  where `T` is `EntityType`, `EmbeddableType`, or `MappedSuperclassType` depending on whether `X` is an entity, embeddable, or mapped superclass.

- For every persistent attribute `y` declared by a managed class `X`, the metamodel class must contain a field declaration as follows:

  ```
  public static final String Y = "y";
  ```

  where the field name `Y` is obtained by transforming each lowercase character in the attribute name `y` to uppercase, inserting an underscore if the character following the transformed character is uppercase, and then replacing each character which is not a legal Java identifier character with an underscore.

- For every persistent non-collection-valued attribute `y` declared by `X`, where the type of `y` is `Y`, the metamodel class must contain a declaration as follows:

  ```
  public static volatile SingularAttribute<X, Y> y;
  ```

- For every persistent collection-valued attribute `z` declared by class `X`, where the element type of `z` is `Z`, the metamodel class must contain a declaration as follows:
  - if the collection type of `z` is `java.util.Collection`, then

    ```
    public static volatile CollectionAttribute<X, Z> z;
    ```

  - if the collection type of `z` is `java.util.Set`, then

```
public static volatile SetAttribute<X, Z> z;
```

- if the collection type of `z` is `java.util.List`, then

```
public static volatile ListAttribute<X, Z> z;
```

- if the collection type of `z` is `java.util.Map`, then

```
public static volatile MapAttribute<X, K, Z> z;
```

where `K` is the type of the key of the map in class `X`

5.2.3. Canonical Metamodel for Eligible Classes

The metamodel class for any eligible class `X` is generated as follows:

- For every method `m` of non-primitive return type `R` and with parameters `p1`, `p2`, etc., of types `T1`, `T2`, etc., annotated with a static query annotation, the metamodel class must contain a declaration as follows:

```
public static TypedQueryReference<R> m(T1 p1, T2 p2, ...) {
    return new StaticTypedQueryReference(
            "X.m",
            X.class,
            "m",
            R.class,
            List.of(T1.class, T2.class, ... ),
            List.of("p1", "p2", ... ),
            List.of(p1, p2, ... ),
            g,
            Map.of("h1", v1, "h2", v2, ...),
            o1, o2, o3, ...
    );
}
```

  where (`h1`, `v1`), (`h2`, `v2`), etc., are `hints` and `o1`, `o2`, etc., are options specified by any `@QueryOptions` annotation of `m`, and g is the name of an entity graph specified by `QueryOptions.entityGraph`. If the annotated method has more parameters than the query specified by the annotation `value` member, the annotation processor is permitted to ignore the method parameters which do not correspond to parameters of the query, but this behavior is not required and is not portable between different processors.[3]

- For every method `m` of return type `void`, `int`, or `long`, and with parameters `p1`, `p2`, etc., of types `T1`, `T2`, etc., annotated with a static query annotation, the metamodel class must contain a declaration as follows:

```
public static StatementReference m(T1 p1, T2 p2, ...) {
    return new StaticStatementReference(
            "X.m",
            X.class,
            "m",
            List.of(T1.class, T2.class, ... ),
            List.of("p1", "p2", ... ),
            List.of(p1, p2, ... ),
            Map.of("h1", v1, "h2", v2, ...),
            o1, o2, o3, ...
    );
}
```

  where (`h1`, `v1`), (`h2`, `v2`), etc., are `hints` and `o1`, `o2`, etc., are options specified by any `@QueryOptions` annotation of `m`. If the annotated method has more parameters than the query specified by the annotation `value` member, the annotation

processor is permitted to ignore the method parameters which do not correspond to parameters of the query, but this behavior is not required and is not portable between different processors.

- For every named query, named entity graph, or SQL result set mapping with name "n" declared by annotations of the class X, the metamodel class must contain a declaration as follows:

  ```
  public static final String T_N = "n";
  ```

  where the prefix T is the string QUERY, GRAPH, or MAPPING, as appropriate, depending on the annotation type, and the suffix N is obtained by transforming each lowercase character in the name n to uppercase, inserting an underscore if the character following the transformed character is uppercase, and then replacing each character which is not a legal Java identifier character with an underscore.

- For every named query with name "n" and query result class R declared by annotations of the class X, the metamodel class must contain a declaration as follows:

  ```
  public static volatile TypedQueryReference<R> _n_;
  ```

  where n is the name "n" with every character which is not a legal Java identifier character replaced with an underscore.

- For every named statement with name "n" declared by annotations of the class X, the metamodel class must contain a declaration as follows:

  ```
  public static volatile StatementReference _n_;
  ```

  where n is the name "n" with every character which is not a legal Java identifier character replaced with an underscore.

- For every named entity graph with name "n" declared by annotations of the class X, the metamodel class must contain a declaration as follows:

  ```
  public static volatile EntityGraph<X> _n;
  ```

  where n is the name "n" with every character which is not a legal Java identifier character replaced with an underscore.

- For every SQL result set mapping with name "n" declared by an annotation of the class X, the metamodel class must contain a declaration as follows:

  ```
  public static volatile ResultSetMapping<R> _n;
  ```

  where n is the name "n" with every character which is not a legal Java identifier character replaced with an underscore and R is the type of the mapped result set, as given by the type, entityClass, or targetClass member of the @ColumnResult, @EntityResult, or @ConstructorResult annotation, respectively. When there is more than one such annotation, the type of the mapped result set is Object[], as specified in Section 3.15.13.3.

Import statements must be included for the needed jakarta.persistence and jakarta.persistence.metamodel types as appropriate and for all classes X, Y, Z, R, and K.

5.2.4. Example Canonical Metamodel

Assume the Order entity below.

```java
package com.example;

import java.util.Set;
import java.math.BigDecimal;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;

@Entity
public class Order {
    @Id
    Integer orderId;

    @ManyToOne
    Customer customer;

    @OneToMany
    Set<Item> lineItems;

    Address shippingAddress;

    BigDecimal totalCost;

    // ...
}
```

The corresponding canonical metamodel class, `Order_`, is as follows:

```java
package com.example;

import java.math.BigDecimal;
import jakarta.persistence.metamodel.EntityType;
import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.SetAttribute;
import jakarta.persistence.metamodel.StaticMetamodel;

@StaticMetamodel(Order.class)
public class Order_ {
    public static volatile EntityType<Order> class_;

    public static volatile SingularAttribute<Order, Integer> orderId;
    public static volatile SingularAttribute<Order, Customer> customer;
    public static volatile SetAttribute<Order, Item> lineItems;
    public static volatile SingularAttribute<Order, Address> shippingAddress;
    public static volatile SingularAttribute<Order, BigDecimal> totalCost;

    public static final String LINE_ITEMS = "lineItems";
    public static final String ORDER_ID = "orderId";
    public static final String SHIPPING_ADDRESS = "shippingAddress";
    public static final String TOTAL_COST = "totalCost";
    public static final String CUSTOMER = "customer";
}
```

[1] We might provide the option of a different package in a future release of this specification.

[2] If the class was generated, it should also be annotated with either `javax.annotation.processing.Generated` or `jakarta.annotation.Generated`. The occurrence of any other annotations on static metamodel classes is undefined.

[3] A future version of this specification might define how an annotation processor treats a method parameter which does not correspond to a parameter of the query.

Chapter 6. Criteria API

The Jakarta Persistence Criteria API is used to define queries through the construction of object-based query definition objects, rather than use of the string-based approach of the Jakarta Persistence query language described in Chapter 4.

This chapter provides the full definition of the Criteria API.

6.1. Overview

The Jakarta Persistence Criteria API, like the Jakarta Persistence query language is based on the abstract persistence schema of entities, their embedded objects, and their relationships as its data model. This abstract persistence schema is materialized in the form of metamodel objects over which the Criteria API operates. The semantics of criteria queries are designed to reflect those of Jakarta Persistence query language queries.

The complete criteria query API may be found in Appendix C.

The syntax of the Criteria API is designed to allow the construction of an object-based query "graph", whose nodes correspond to the semantic query elements.

Java language variables can be used to reference individual nodes in a criteria query object as it is constructed and/or modified. Such variables, when used to refer to the entities and embeddable types that constitute the query domain, play a role analogous to that of the identification variables of the Jakarta Persistence query language.

These concepts are further described in the sections that follow. Sections Section 6.2 through Section 6.6 describe the construction and modification of criteria query objects. Additional requirements on the persistence provider are described in Section 6.7.

The metamodel on which criteria queries are based was already presented in Chapter 5. The static metamodel classes which are used to construct strongly-typed criteria queries are described in Section 5.1.

6.2. Criteria Query API Usage

The `jakarta.persistence.criteria` API interfaces are designed both to allow criteria queries to be constructed in a strongly-typed manner, using metamodel objects to provide type safety, and to allow for string-based use as an alternative:

Metamodel objects are used to specify navigation through joins and through path expressions[1]. Typesafe navigation is achieved by specification of the source and target types of the navigation.

Strings may be used as an alternative to metamodel objects, whereby joins and navigation are specified by use of strings that correspond to attribute names.

Using either the approach based on metamodel objects or the string-based approach, queries can be constructed both statically and dynamically. Both approaches are equivalent in terms of the range of queries that can be expressed and operational semantics.

Section 6.3 provides a description of the use of the criteria API interfaces. This section is illustrated on the basis of the construction of strongly-typed queries using static metamodel classes. Section 6.4 describes how the `jakarta.persistence.metamodel` API can be used to construct strongly-typed queries in the absence of such classes. String-based use of the criteria API is described in Section 6.5.

6.3. Constructing Criteria Queries

A criteria query is constructed through the creation and modification of an instance of the `CriteriaQuery` interface found in Section C.2.

The `CriteriaBuilder` interface found in Section C.1 is used to construct `CriteriaQuery`, `CriteriaUpdate`, and `CriteriaDelete` objects. The `CriteriaBuilder` implementation is accessed through the `getCriteriaBuilder` method of the `EntityManager` or `EntityManagerFactory` interface.

For example:

```
@PersistenceUnit
EntityManagerFactory emf;

CriteriaBuilder cb = emf.getCriteriaBuilder();
```

6.3.1. CriteriaQuery Creation

A `CriteriaQuery` object is created by means of one of the `createQuery` methods or the `createTupleQuery` method of the `CriteriaBuilder` interface. A `CriteriaQuery` object is typed according to its expected result type when the `CriteriaQuery` object is created. A `TypedQuery` instance created from the `CriteriaQuery` object by means of the `EntityManager` `createQuery` method will result in instances of this type when the resulting query is executed.

The following methods are provided for the creation of `CriteriaQuery` objects:

```
<T> CriteriaQuery<T> createQuery(Class<T> resultClass);

CriteriaQuery<Tuple> createTupleQuery();

CriteriaQuery<Object> createQuery();
```

Methods for the creation of update and delete queries are described in Section 6.3.15.

The methods `<T> CriteriaQuery<T> createQuery(Class<T> resultClass)` and `createTupleQuery` provide for typing of criteria query results and for typesafe query execution using the `TypedQuery` API.

The effect of the `createTupleQuery` method is semantically equivalent to invoking the `createQuery` method with the `Tuple.class` argument. The `Tuple` interface supports the extraction of multiple selection items in a strongly typed manner. See Section B.13 and Section B.14.

The `CriteriaQuery<Object> createQuery()` method supports both the case where the `select` or `multiselect` method specifies only a single selection item and where the `multiselect` method specifies multiple selection items. If only a single item is specified, an instance of type `Object` is returned for each result of the query execution. If multiple selection items are specified, an instance of type `Object[]` is instantiated and returned for each result of the execution.

See Section 6.3.11 for further discussion of the specification of selection items.

6.3.2. Query Roots

A `CriteriaQuery` object defines a query over one or more entity, embeddable, or basic abstract schema types. The root objects of the query are entities, from which the other types are reached by navigation. A query root plays a role analogous to that of a range variable in the Jakarta Persistence query language and forms the basis for defining the domain of the query.

A query root is created and added to the query by use of the `from` method of the `AbstractQuery` interface (from which both the `CriteriaQuery` and `Subquery` interfaces inherit). The argument to the `from` method is the entity class or `EntityType`

instance for the entity. The result of the `from` method is a `Root` object. The `Root` interface extends the `From` interface, which represents objects that may occur in the from clause of a query.

A `CriteriaQuery` object may have more than one root. The addition of a query root has the semantic effect of creating a cartesian product between the entity type referenced by the added root and those of the other roots.

The following query illustrates the definition of a query root. When executed, this query causes all instances of the `Customer` entity to be returned.

```
CriteriaBuilder cb = ...
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
q.select(customer);
```

6.3.3. Joins

The `join` methods of the `From` interface extend the query domain by creating a join with a related class that can be navigated to or that is an element of the given class of the query domain.

The target of the join is specified by the corresponding `SingularAttribute` or collection-valued attribute (`CollectionAttribute`, `SetAttribute`, `ListAttribute`, or `MapAttribute`) of the corresponding metamodel class.[2] [3]

The `join` methods may be applied to instances of the `Root` and `Join` types.

The result of a `join` method is a `Join` object (instance of the `Join`, `CollectionJoin`, `SetJoin`, `ListJoin`, or `MapJoin` types) that captures the source and target types of the join.

For example, given the `Order` entity and corresponding `Order_` metamodel class shown in Section 5.2.4, a join to the lineItems of the order would be expressed as follows:

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Order> order = q.from(Order.class);
Join<Order, Item> item = order.join(Order_.lineItems);
q.select(order);
```

The argument to the `join` method, `Order.lineItems`, is of type `jakarta.persistence.metamodel.SetAttribute<Order, Item>`.

The `join` methods have the same semantics as the corresponding Jakarta Persistence query language operations, as described in Section 4.4.7.

Example:

```
CriteriaBuilder cb = ...
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Customer> customer = q.from(Customer.class);
Join<Customer, Order> order = customer.join(Customer_.orders);
Join<Order, Item> item = order.join(Order_.lineItems);
q.select(customer.get(Customer_.name))
    .where(cb.equal(item.get(Item_.product).get(Product_.productType), "printer"));
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT c.name
FROM Customer c JOIN c.orders o JOIN o.lineItems i
WHERE i.product.productType = 'printer'
```

Joins can be chained, thus allowing this query to be written more concisely:

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Customer> customer = q.from(Customer.class);
Join<Order, Item> item = customer.join(Customer_.orders).join(Order_.lineItems);
q.select(customer.get(Customer_.name))
    .where(cb.equal(item.get(Item_.product).get(Product_.productType), "printer"));
```

By default, the `join` method defines an inner join. Outer joins are defined by explicitly specifying a `JoinType` argument.

The following query uses a left outer join:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
Join<Customer,Order> order = customer.join(Customer_.orders, JoinType.LEFT);
q.where(cb.equal(customer.get(Customer_.status), 1))
    .select(customer);
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT c FROM Customer c LEFT JOIN c.orders o WHERE c.status = 1
```

On-conditions can be specified for joins. The following query uses an on-condition with a left outer join:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Supplier> s = q.from(Supplier.class);
Join<Supplier, Product> p = s.join(Supplier_.products, JoinType.LEFT);
p.on(cb.equal(p.get(Product_.status), "inStock"));
q.groupBy(s.get(Supplier_.name));
q.multiselect(s.get(Supplier_.name), cb.count(p));
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p ON p.status = 'inStock'
GROUP BY s.name
```

6.3.4. Fetch Joins

Fetch joins are specified by means of the `fetch` method. The `fetch` method specifies that the referenced association or attribute is to be fetched as a side effect of the execution of the query. The `fetch` method can be applied to a `Root` or `Join` instance.

An association or attribute referenced by the `fetch` method must be referenced from an entity or embeddable that is returned as the result of the query. A fetch join has the same join semantics as the corresponding inner or outer join, except that the related objects are not top-level objects in the query result and cannot be referenced elsewhere by the query. See Section 4.4.5.3.

The `fetch` method must not be used in a subquery.

Multiple levels of fetch joins are not required to be supported by an implementation of this specification. Applications that use multi-level fetch joins are not portable.

Example:

```
CriteriaQuery<Department> q = cb.createQuery(Department.class);
Root<Department> d = q.from(Department.class);
d.fetch(Department.employees, JoinType.LEFT);
q.where(cb.equal(d.get(Department_.deptno), 1)).select(d);
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

6.3.5. Path Navigation

A `Path` instance can be a `Root` instance, a `Join` instance, a `Path` instance that has been derived from another `Path` instance by means of the `get` navigation method, or a `Path` instance derived from a map-valued association or element collection by use of the `key` or `value` method.

When a criteria query is executed, path navigation—like path navigation using the Jakarta Persistence query language—is obtained using "inner join" semantics. That is, if the value of a non-terminal `Path` instance is null, the path is considered to have no value, and does not participate in the determination of the query result. See Section 4.4.4.

The `get` method is used for path navigation. The argument to the `get` method is specified by the corresponding `SingularAttribute` or collection-valued attribute (`CollectionAttribute`, `SetAttribute`, `ListAttribute`, or `MapAttribute`) of the corresponding metamodel class[4].

Example 1:

In the following example, `ContactInfo` is an embeddable class consisting of an address and set of phones. `Phone` is an entity.

```
CriteriaQuery<Vendor> q = cb.createQuery(Vendor.class);
Root<Employee> emp = q.from(Employee.class);
Join<ContactInfo, Phone> phone =
    emp.join(Employee_.contactInfo).join(ContactInfo_.phones);
q.where(cb.equal(emp.get(Employee_.contactInfo)
        .get(ContactInfo_.address)
        .get(Address_.zipcode), "95054"))
    .select(phone.get(Phone_.vendor));
```

The following Jakarta Persistence query language query is equivalent:

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo.phones p
WHERE e.contactInfo.address.zipcode = '95054'
```

Example 2:

In this example, the `photos` attribute corresponds to a map from photo label to filename. The map key is a string, the value an object. The result of this query is returned as a `Tuple` object whose elements are of types `String` and `Object`. The `multiselect` method, described further in Section 6.3.11, is used to specify that the query returns multiple selection items.

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Item> item = q.from(Item.class);
MapJoin<Item, String, Object> photo = item.join(Item_.photos);
q.multiselect(item.get(Item_.name), photo)
    .where(cb.like(photo.key(), "%egret%"));
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT i.name, p
FROM Item i JOIN i.photos p
```

```
WHERE KEY(p) LIKE '%egret%'
```

6.3.6. Restricting the Query Result

The result of a query can be restricted by specifying one or more predicate conditions. Restriction predicates are applied to the `CriteriaQuery` object by means of the `where` method. Invocation of the `where` method results in the modification of the `CriteriaQuery` object with the specified restriction(s).

The argument to the `where` method can be either an *Expression<Boolean>* instance or zero or more `Predicate` instances. A predicate can be either simple or compound.

A simple predicate is created by invoking one of the conditional methods of the `CriteriaBuilder` interface, or by the `isNull`, `isNotNull`, and `in` methods of the `Expression` interface. The semantics of the conditional methods—e.g., `equal`, `notEqual`, `gt`, `ge`, `lt`, `le`, `between`, and `like` — mirror those of the corresponding Jakarta Persistence query language operators as described in Chapter 4.

Compound predicates are constructed by means of the `and`, `or`, and `not` methods of the `CriteriaBuilder` interface.

The restrictions upon the types to which conditional operations are permitted to be applied are the same as the respective operators of the Jakarta Persistence query language as described in subsections Section 4.6.3 through Section 4.7. The same null value semantics as described in Section 4.6.13 and the subsections of Section 4.6 apply. The equality and comparison semantics described in Section 4.6.14 likewise apply.

Example 1:

```
CriteriaQuery<TransactionHistory> q = cb.createQuery(TransactionHistory.class);
Root<CreditCard> cc = q.from(CreditCard.class);
ListJoin<CreditCard,TransactionHistory> t = cc.join(CreditCard_.transactionHistory);
q.select(t)
    .where(cb.equal(cc.get(CreditCard_.customer)
              .get(Customer_.accountNum), 321987),
           cb.between (t.index(), 0, 9));
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT t
FROM CreditCard c JOIN c.transactionHistory t
WHERE c.customer.accountNum = 321987 AND INDEX(t) BETWEEN 0 AND 9
```

Example 2:

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Order> order = q.from(Order.class);
q.where(cb.isEmpty(order.get(Order_.lineItems)))
    .select(order);
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

6.3.7. Downcasting

Downcasting by means of the `treat` method is supported in joins and in the construction of `where` conditions.

Example 1:

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Order> order = q.from(Order.class);
Join<Order,Book> book = cb.treat(order.join(Order_.product), Book.class);
q.select(book.get(Book_.isbn));
```

This query is equivalent to the following Jakarta Persistence query language query.

```
SELECT b.ISBN
FROM Order o JOIN TREAT(o.product AS Book) b
```

Example 2:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
Join<Customer, Order> order = customer.join(Customer_.orders);
q.where(
    cb.equal(cb.treat(order.get(Order_.product), Book.class).get(Book_.name), "Iliad"));
q.select(customer);
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE TREAT(o.product AS Book).name = 'Iliad'
```

Example 3:

```
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> e = q.from(Employee.class);
q.where(
    cb.or(cb.gt(cb.treat(e, Exempt.class).get(Exempt_.vacationDays), 10),
    cb.gt(cb.treat(e, Contractor.class).get(Contractor_.hours), 100)));
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT e
FROM Employee e
WHERE TREAT(e AS Exempt).vacationDays > 10
    OR TREAT(e AS Contractor).hours > 100
```

6.3.8. Expressions

An `Expression` or one of its subtypes can be used in the construction of the query's select list or in the construction of `where` or `having` method conditions.

Paths and boolean predicates are expressions.

Other expressions are created by means of the methods of the `CriteriaBuilder` interface. The `CriteriaBuilder` interface provides methods corresponding to the built-in arthmetic, string, datetime, and case operators and functions of the Jakarta Persistence query language.

Example 1:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> cust = q.from(Customer.class);
Join<Customer, Order> order = cust.join(Customer_.orders);
Join<Customer, Address> addr = cust.join(Customer_.address);
q.where(cb.equal(addr.get(Address_.state), "CA"),
        cb.equal(addr.get(Address_.county), "Santa Clara"));
```

```
q.multiselect(order.get(Order_.quantity),
        cb.prod(order.get(Order_.totalCost), 1.08),
        addr.get(Address_.zipcode));
```

The following Jakarta Persistence query language query is equivalent:

```
SELECT o.quantity, o.totalCost*1.08, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA' AND a.county = 'Santa Clara'
```

Example 2:

```
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> emp = q.from(Employee.class);
q.select(emp)
    .where(cb.notEqual(emp.type(), Exempt.class));
```

The `type` method can only be applied to a path expression. Its result denotes the type navigated to by the path.

The following Jakarta Persistence query language query is equivalent:

```
SELECT e
FROM Employee e
WHERE TYPE(e) <> Exempt
```

Example 3:

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Course> c = q.from(Course.class);
ListJoin<Course, Student> w = c.join(Course_.studentWaitlist);
q.where(cb.equal(c.get(Course_.name), "Calculus"),
        cb.equal(w.index(), 0))
 .select(w.get(Student_.name));
```

The `index` method can be applied to a `ListJoin` object that corresponds to a list for which an order column has been specified. Its result denotes the position of the item in the list.

The following Jakarta Persistence query language query is equivalent:

```
SELECT w.name
FROM Course c JOIN c.studentWaitlist w
WHERE c.name = 'Calculus' AND INDEX(w) = 0
```

Example 4:

```
CriteriaQuery<BigDecimal> q = cb.createQuery(BigDecimal.class);
Root<Order> order = q.from(Order.class);
Join<Order, Item> item = order.join(Order_.lineItems);
Join<Order, Customer> cust = order.join(Order_.customer);

q.where(
    cb.equal(cust.get(Customer_.lastName), "Smith"),
    cb.equal(cust.get(Customer_.firstName), "John"));
q.select(cb.sum(item.get(Item_.price)));
```

The aggregation methods `avg`, `max`, `min` , `sum`, `count` can only be used in the construction of the select list or in `having` method conditions.

The following Jakarta Persistence query language query is equivalent:

```
SELECT SUM(i.price)
FROM Order o JOIN o.lineItems i JOIN o.customer c
WHERE c.lastName = 'Smith' AND c.firstName = 'John'
```

Example 5:

```
CriteriaQuery<Integer> q = cb.createQuery(Integer.class);
Root<Department> d = q.from(Department.class);
q
    .where(cb.equal(d.get(Department_.name), "Sales"))
    .select(cb.size(d.get(Department_.employees)));
```

The `size` method can be applied to a path expression that corresponds to an association or element collection. Its result denotes the number of elements in the association or element collection.

The following Jakarta Persistence query language query is equivalent:

```
SELECT SIZE(d.employees)
FROM Department d
WHERE d.name = 'Sales'
```

Example 6:

Both simple and general case expressions are supported. The query below illustrates use of a general case expression.

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Employee> e = q.from(Employee.class);
q.where(
    cb.equal(e.get(Employee_.department).get(Department_.name), "Engineering"));
q.multiselect(
    e.get(Employee_.name),
    cb.selectCase()
        .when(
            cb.equal(e.get(Employee_.rating), 1),
            cb.prod(e.get(Employee_.salary), 1.1))
        .when(
            cb.equal(e.get(Employee_.rating), 2),
            cb.prod(e.get(Employee_.salary), 1.2))
        .otherwise(cb.prod(e.get(Employee_.salary), 1.01)));
```

The following Jakarta Persistence query language query is equivalent:

```
SELECT e.name,
    CASE
        WHEN e.rating = 1 THEN e.salary * 1.1
        WHEN e.rating = 2 THEN e.salary * 1.2
        ELSE e.salary * 1.01
    END
FROM EMPLOYEE e
WHERE e.department.name = 'Engineering'
```

6.3.8.1. Result Types of Expressions

The `getJavaType` method, as defined in the `TupleElement` interface, returns the runtime type of the object on which it is invoked.

In the case of the `In`, `Case`, `SimpleCase`, and `Coalesce` builder interfaces, the runtime results of the `getJavaType` method may differ from the `Expression` type and may vary as the expression is incrementally constructed. For non-numerical

operands, the implementation must return the most specific common superclass of the types of the operands used to form the result.

In the case of the two-argument `sum`, `prod`, `diff`, `quot`, `coalesce`, and `nullif` methods, and the `In`, `Case`, `SimpleCase`, and `Coalesce` builder methods, the runtime result types will differ from the `Expression` type when the latter is `Number`. The following rules must be observed by the implementation when materializing the results of numeric expressions involving these methods. These rules correspond to those specified for the Jakarta Persistence query language as defined in Section 4.7.13.

- If there is an operand of type Double, the result of the operation is of type Double;
- otherwise, if there is an operand of type Float, the result of the operation is of type Float;
- otherwise, if there is an operand of type BigDecimal, the result of the operation is of type BigDecimal;
- otherwise, if there is an operand of type BigInteger, the result of the operation is of type BigInteger, unless the method is `quot`, in which case the numeric result type is not further defined;
- otherwise, if there is an operand of type Long, the result of the operation is of type Long, unless the method is `quot`, in which case the numeric result type is not further defined;
- otherwise, if there is an operand of integral type, the result of the operation is of type Integer, unless the method is `quot`, in which case the numeric result type is not further defined.

> The semantics of the SQL division operation are not standard across databases. In particular, when both operands are of integral types, the result of the division operation is an integral type in some databases, and a non-integral type in others. Portable applications should not assume a particular result type.

6.3.9. Literals

An `Expression` literal instance is obtained by passing a value to the `literal` method of the `CriteriaBuilder` interface. An `Expression` instance representing a null is created by the `nullLiteral` method of the `CriteriaBuilder` interface.

Example:

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Employee> emp = q.from(Employee.class);
Join<Employee, FrequentFlierPlan> fp = emp.join(Employee_.frequentFlierPlan);

q.select(
    cb.<String>selectCase()
        .when(
            cb.gt(fp.get(FrequentFlierPlan_.annualMiles), 50000),
            cb.literal("Platinum"))
        .when(
            cb.gt(fp.get(FrequentFlierPlan_.annualMiles), 25000),
            cb.literal("Silver"))
        .otherwise(cb.nullLiteral(String.class)));
```

The following Jakarta Persistence query language query is equivalent:

```
SELECT
    CASE
        WHEN fp.annualMiles > 50000 THEN 'Platinum'
        WHEN fp.annualMiles > 25000 THEN 'Gold'
        ELSE NULL
    END
```

6.3.10. Parameter Expressions

A `ParameterExpression` instance is an expression that corresponds to a parameter whose value will be supplied before the query is executed. Parameter expressions can only be used in the construction of conditional predicates.

Example:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> c = q.from(Customer.class);
ParameterExpression<Integer> param = cb.parameter(Integer.class);

q.select(c)
    .where(cb.equal(c.get(Customer_.status), param));
```

If a name is supplied when the `ParameterExpression` instance is created, the parameter may also be treated as a named parameter when the query is executed:

```
CriteriaQuery<Customer> q =
cb.createQuery(Customer.class);

Root<Customer> c = q.from(Customer.class);
ParameterExpression<Integer> param = cb.parameter(Integer.class, "stat");
q.select(c).where(cb.equal(c.get(Customer_.status), param));
```

This is equivalent to the following query in the Jakarta Persistence query language:

```
SELECT c FROM Customer c WHERE c.status = :stat
```

6.3.11. Specifying the Select List

The select list of a query is specified by use of the `select` or `multiselect` methods of the `CriteriaQuery` interface. The arguments to the `select` and `multiselect` methods are `Selection` instances.

> Portable applications should use the `select` or `multiselect` method to specify the query's selection list. Applications that do not use one of these methods are not portable.

The `select` method takes a single `Selection` argument, which can be either an `Expression` instance or a `CompoundSelection` instance. The type of the `Selection` item must be assignable to the defined `CriteriaQuery` result type, as described in Section 6.3.1.

The `construct`, `tuple` and `array` methods of the `CriteriaBuilder` interface are used to aggregate multiple selection items into a `CompoundSelection` instance.

The `multiselect` method also supports the specification and aggregation of multiple selection items. When the `multiselect` method is used, the aggregation of the selection items is determined by the result type of the `CriteriaQuery` object as described in Section 6.3.1.

A `Selection` instance passed to the `construct`, `tuple`, `array`, or `multiselect` methods can be one of the following:

- An `Expression` instance.
- A `Selection` instance obtained as the result of the invocation of the *CriteriaBuilder construct* method.

The `distinct` method of the `CriteriaQuery` interface specifies that duplicate values must be eliminated from the query result. If the `distinct` method is not used or `distinct(false)` is invoked on the criteria query object, duplicate values are not eliminated. When `distinct(true)` is used, and the select items include embeddable objects or map entry results, the elimination of duplicates is undefined.

The semantics of the `construct` method used in the selection list is as described in Section 4.9.2. The semantics of embeddables returned by the selection list areas described in Section 4.9.4.

Example 1:

In the following example, `videoInventory` is a Map from the entity `Movie` to the number of copies in stock.

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<VideoStore> v = q.from(VideoStore.class);
MapJoin<VideoStore, Movie, Integer> inv = v.join(VideoStore_.videoInventory);

q.multiselect(
    v.get(VideoStore_.location).get(Address_.street),
    inv.key().get(Movie_.title),
    inv);
q.where(cb.equal(v.get(VideoStore_.location).get(Address_.zipcode), "94301"),
        cb.gt(inv, 0));
```

This query is equivalent to the following, in which the `tuple` method is used:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<VideoStore> v = q.from(VideoStore.class);
MapJoin<VideoStore, Movie, Integer> inv = v.join(VideoStore_.videoInventory);

q.select(cb.tuple(
        v.get(VideoStore_.location).get(Address_.street),
        inv.key().get(Movie_.title),
        inv));
q.where(cb.equal(v.get(VideoStore_.location).get(Address_.zipcode), "94301"),
        cb.gt(inv, 0));
```

Both are equivalent to the following Jakarta Persistence query language query:

```
SELECT v.location.street, KEY(i).title, VALUE(i)
FROM VideoStore v JOIN v.videoInventory i
WHERE v.location.zipcode = '94301' AND VALUE(i) > 0
```

Example 2:

The following two queries are equivalent to the Jakarta Persistence query language query above. Because the result type is not specified by the `createQuery` method, an `Object[]` is returned as a result of the query execution:

```
CriteriaQuery<Object> q = cb.createQuery();
Root<VideoStore> v = q.from(VideoStore.class);
MapJoin<VideoStore, Movie, Integer> inv = v.join(VideoStore_.videoInventory);

q.multiselect(
    v.get(VideoStore_.location).get(Address_.street),
    inv.key().get(Movie_.title),
    inv);

q.where(cb.equal(v.get(VideoStore_.location).get(Address_.zipcode), "94301"),
        cb.gt(inv, 0));
```

Equivalently:

```
CriteriaQuery<Object> q = cb.createQuery();
Root<VideoStore> v = q.from(VideoStore.class);
MapJoin<VideoStore, Movie, Integer> inv = v.join(VideoStore_.videoInventory);

q.select(cb.array(
        v.get(VideoStore_.location).get(Address_.street),
```

```
        inv.key().get(Movie_.title),
        inv));
q.where(cb.equal(v.get(VideoStore_.location).get(Address_.zipcode), "94301"),
        cb.gt(inv, 0));
```

Example 3:

The following example illustrates the specification of a constructor.

```
CriteriaQuery<CustomerDetails> q = cb.createQuery(CustomerDetails.class);
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);

q.where(cb.gt(o.get(Order_.quantity), 100));
q.select(cb.construct(
        CustomerDetails.class,
        c.get(Customer_.id),
        c.get(Customer_.status),
        o.get(Order_.quantity)));
```

The following Jakarta Persistence query language query is equivalent:

```
SELECT NEW com.acme.example.CustomerDetails(c.id, c.status, o.quantity)
FROM Customer c JOIN c.orders o
WHERE o.quantity > 100
```

6.3.11.1. Assigning Aliases to Selection Items

The `alias` method of the `Selection` interface can be used to assign an alias to a selection item. The alias may then later be used to extract the corresponding item from the query result when the query is executed. The `alias` method assigns the given alias to the `Selection` item. Once assigned, the alias cannot be changed.

Example:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);

q.where(cb.equal(c.get(Customer_.id), 97510));
q.multiselect(
        o.get(Order_.quantity).alias("quantity"),
        cb.prod(o.get(Order_.totalCost), 1.08).alias("taxedCost"),
        a.get(Address_.zipcode).alias("zipcode"));

TypedQuery<Tuple> typedQuery = em.createQuery(q);
Tuple result = typedQuery.getSingleResult();
Double cost = (Double)result.get("taxedCost");
```

6.3.12. Subqueries

Both correlated and non-correlated subqueries can be used in restriction predicates. A subquery is constructed through the creation and modification of a `Subquery` object.

A `Subquery` instance can be passed as an argument to the `all`, `any`, or `some` methods of the `CriteriaBuilder` interface for use in conditional expressions.

A `Subquery` instance can be passed to the `CriteriaBuilder` `exists` method to create a conditional predicate.

Example 1: Non-correlated subquery

The query below contains a non-correlated subquery. A non-correlated subquery does not reference objects of the query of which it is a subquery. In particular, `Root`, `Join`, and `Path` instances are not shared between the subquery and the criteria query instance of which it is a subquery.

```
// create criteria query instance, with root Customer
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> goodCustomer = q.from(Customer.class);

// create subquery instance, with root Customer
// the Subquery object is typed according to its return type
Subquery<Double> sq = q.subquery(Double.class);
Root<Customer> customer = sq.from(Customer.class);

// the result of the first query depends on the subquery
q.where(cb.lt(
    goodCustomer.get(Customer_.balanceOwed),
    sq.select(cb.avg(customer.get(Customer_.balanceOwed)))));
q.select(goodCustomer);
```

This query corresponds to the following Jakarta Persistence query language query.

```
SELECT goodCustomer
FROM Customer goodCustomer
WHERE goodCustomer.balanceOwed < (SELECT AVG(c.balanceOwed) FROM Customer c)
```

Example 2: Correlated subquery

```
// create CriteriaQuery instance, with root Employee
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> emp = q.from(Employee.class);

// create Subquery instance, with root Employee
Subquery<Employee> sq = q.subquery(Employee.class);
Root<Employee> spouseEmp = sq.from(Employee.class);

// the subquery references the root of the containing query
sq.where(cb.equal(spouseEmp, emp.get(Employee_.spouse)))
    .select(spouseEmp);

// an exists condition is applied to the subquery result:
q.where(cb.exists(sq));
q.select(emp).distinct(true);
```

The above query corresponds to the following Jakarta Persistence query language query.

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
    SELECT spouseEmp
    FROM Employee spouseEmp
    WHERE spouseEmp = emp.spouse)
```

Example 3: Subquery qualified by all()

```
// create CriteriaQuery instance, with root Employee
CriteriaQuery<Employee> q = cb.createQuery(Employee.class);
Root<Employee> emp = q.from(Employee.class);

// create Subquery instance, with root Manager
Subquery<BigDecimal> sq = q.subquery(BigDecimal.class);
Root<Manager> manager = sq.from(Manager.class);
```

```
sq.select(manager.get(Manager_.salary));
sq.where(cb.equal(
    manager.get(Manager_.department),
    emp.get(Employee_.department)));

// an all expression is applied to the subquery result
q.select(emp)
    .where(cb.gt(emp.get(Employee_.salary), cb.all(sq)));
```

This query corresponds to the following Jakarta Persistence query language query:

```
SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
    SELECT m.salary
    FROM Manager m
    WHERE m.department = emp.department)
```

Example 4: A Special case

In order to express some correlated subqueries involving unidirectional relationships, it may be useful to correlate the domain of the subquery with the domain of the containing query. This is performed by using the `correlate` method of the `Subquery` interface.

For example:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
Subquery<Long> sq = q.subquery(Long.class);
Root<Customer> customerSub = sq.correlate(customer);
Join<Customer,Order> order = customerSub.join(Customer_.orders);

q.where(cb.gt(sq.select(cb.count(order)), 10))
    .select(customer);
```

This query corresponds to the following Jakarta Persistence query language query:

```
SELECT c
FROM Customer c
WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

Note that joins involving the derived subquery root do not affect the join conditions of the containing query. The following two query definitions thus differ in semantics:

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Order> order = q.from(Order.class);
Subquery<Integer> sq = q.subquery(Integer.class);
Root<Order> orderSub = sq.correlate(order);
Join<Order,Customer> customer = orderSub.join(Order_.customer);
Join<Customer,Account> account = customer.join(Customer_.accounts);

sq.select(account.get(Account_.balance));
q.where(cb.lt(cb.literal(10000), cb.all(sq)));
```

and

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Order> order = q.from(Order.class);
Join<Order,Customer> customer = order.join(Order_.customer);
Subquery<Integer> sq = q.subquery(Integer.class);
Join<Order,Customer> customerSub = sq.correlate(customer);
```

```
Join<Customer,Account> account = customerSub.join(Customer_.accounts);

sq.select(account.get(Account_.balance));
q.where(cb.lt(cb.literal(10000), cb.all(sq)));
```

The first of these queries returns orders that are not associated with customers, whereas the second does not. The corresponding Jakarta Persistence query language queries are the following:

```
SELECT o
FROM Order o
WHERE 10000 < ALL (
    SELECT a.balance
    FROM o.customer c JOIN c.accounts a)
```

and

```
SELECT o
FROM Order o JOIN o.customer c
WHERE 10000 < ALL (
    SELECT a.balance
    FROM c.accounts a)
```

6.3.13. GroupBy and Having

The `groupBy` method of the `CriteriaQuery` interface is used to define a partitioning of the query results into groups. The `having` method of the `CriteriaQuery` interface can be used to filter over the groups.

The arguments to the `groupBy` method are `Expression` instances.

When the `groupBy` method is used, each selection item that is not the result of applying an aggregate method must correspond to a path expression that is used for defining the grouping. Requirements on the types that correspond to the elements of the grouping and having constructs and their relationship to the select items are as specified in Section 4.8.

Example:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> customer = q.from(Customer.class);

q.groupBy(customer.get(Customer_.status));
q.having(cb.in(customer.get(Customer_.status)).value(1).value(2));
q.select(cb.tuple(
        customer.get(Customer_.status),
        cb.avg(customer.get(Customer_.filledOrderCount)),
        cb.count(customer)));
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT c.status, AVG(c.filledOrderCount), COUNT(c)
FROM Customer c
GROUP BY c.status
HAVING c.status IN (1, 2)
```

6.3.14. Ordering the Query Results

The ordering of the results of a query is defined by use of the `orderBy` method of the `CriteriaQuery` instance. The arguments to the `orderBy` method are `Order` instances.

An `Order` instance is created by means of the `asc` and `desc` methods of the `CriteriaBuilder` interface. An argument to either of these methods must be one of the following:

- Any `Expression` instance that corresponds to an orderable state field of an entity or embeddable class abstract schema type that is specified as an argument to the `select` or `multiselect` method or that is an argument to a tuple or array constructor that is passed as an argument to the `select` method.
- Any `Expression` instance that corresponds to the same state field of the same entity or embeddable abstract schema type as an `Expression` instance that is specified as an argument to the `select` or `multiselect` method or that is an argument to a tuple or array constructor that is passed as an argument to the `select` method.
- An `Expression` instance that is specified as an argument to the `select` or `multiselect` method or that is an argument to a tuple or array constructor that is passed as an argument to the `select` method or that is semantically equivalent to such an `Expression` instance.

If more than one `Order` instance is specified, the order in which they appear in the argument list of the `orderBy` method determines the precedence, whereby the first item has highest precedence.

SQL rules for the ordering of null values apply, as described in Section 4.10.

Example 1:

```
CriteriaQuery<Order> q = cb.createQuery(Order.class);
Root<Customer> c = q.from(Customer.class);
Join<Customer,Order> o = c.join(Customer_.orders);
Join<Customer,Address> a = c.join(Customer_.address);

q.where(cb.equal(a.get(Address_.state), "CA"));
q.select(o);
q.orderBy(cb.desc(o.get(Order_.quantity)),
    cb.asc(o.get(Order_.totalCost)));
```

This query corresponds to the following Jakarta Persistence query language query:

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity DESC, o.totalcost
```

Example 2:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);

q.where(cb.equal(a.get(Address_.state), "CA"));
q.orderBy(cb.asc(o.get(Order_.quantity)),
    cb.asc(a.get(Address_.zipcode)));
q.multiselect(o.get(Order_.quantity),
    a.get(Address_.zipcode));
```

This query corresponds to the following Jakarta Persistence query language query:

```
SELECT o.quantity, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, a.zipcode
```

It can be equivalently expressed as follows:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);

q.where(cb.equal(a.get(Address_.state), "CA"));
q.orderBy(cb.asc(o.get(Order_.quantity)),
    cb.asc(a.get(Address_.zipcode)));
q.select(cb.tuple(o.get(Order_.quantity),
    a.get(Address_.zipcode)));
```

Example 3:

```
CriteriaQuery<Object[]> q = cb.createQuery(Object[].class);
Root<Customer> c = q.from(Customer.class);
Join<Customer, Order> o = c.join(Customer_.orders);
Join<Customer, Address> a = c.join(Customer_.address);

q.where(cb.equal(a.get(Address_.state), "CA"),
    cb.equal(a.get(Address_.county), "Santa Clara"));
q.select(cb.array(o.get(Order_.quantity),
    cb.prod(o.get(Order_.totalCost), 1.08),
    a.get(Address_.zipcode)));
q.orderBy(cb.asc(o.get(Order_.quantity)),
    cb.asc(cb.prod(o.get(Order_.totalCost), 1.08)),
    cb.asc(a.get(Address_.zipcode)));
```

This query corresponds to the following Jakarta Persistence query language query:

```
SELECT o.quantity, o.totalCost * 1.08 AS taxedCost, a.zipcode
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA' AND a.county = 'Santa Clara'
ORDER BY o.quantity, taxedCost, a.zipcode
```

6.3.15. Bulk Update and Delete Operations

A bulk update query is constructed through the creation and modification of a `jakarta.persistence.criteria.CriteriaUpdate` object.

A `CriteriaUpdate` object is created by means of one of the `createCriteriaUpdate` methods of the `CriteriaBuilder` interface. A `CriteriaUpdate` object is typed according to the entity type that is the target of the update. A `CriteriaUpdate` object has a single root, the entity that is being updated.

A bulk delete query is constructed through the creation and modification of a `jakarta.persistence.criteria.CriteriaDelete` object.

A `CriteriaDelete` object is created by means of one of the `createCriteriaDelete` methods of the `CriteriaBuilder` interface. A `CriteriaDelete` object is typed according to the entity type that is the target of the delete. A `CriteriaDelete` object has a single root, the entity that is being deleted.

Example 1:

```
CriteriaUpdate<Customer> q = cb.createCriteriaUpdate(Customer.class);
Root<Customer> c = q.from(Customer.class);

q.set(c.get(Customer_.status), "outstanding")
    .where(cb.lt(c.get(Customer_.balance), 10000));
```

The following Jakarta Persistence query language update statement is equivalent.

```
UPDATE Customer c
SET c.status = 'outstanding'
WHERE c.balance < 10000
```

Example 2:

```
CriteriaUpdate<Employee> q = cb.createCriteriaUpdate(Employee.class);
Root<Employee> e = q.from(Employee.class);

q.set(e.get(Employee_.address).get(Address_.building), 22)
    .where(
        cb.equal(e.get(Employee_.address).get(Address_.building), 14),
        cb.equal(e.get(Employee_.address).get(Address_.city), "Santa Clara"),
        cb.equal(e.get(Employee_.project).get(Project_.name), "Jakarta EE"));
```

`Address` is an embeddable class. Note that updating across implicit joins is not supported.

The following Jakarta Persistence query language update statement is equivalent.

```
UPDATE Employee e
SET e.address.building = 22
WHERE e.address.building = 14
    AND e.address.city = 'Santa Clara'
    AND e.project.name = 'Jakarta EE'
```

Example 3:

The following update query causes multiple attributes to be updated.

```
CriteriaUpdate<Employee> q = cb.createCriteriaUpdate(Employee.class);
Root<Employee> e = q.from(Employee.class);

q.set(e.get(Employee_.salary), cb.prod(e.get(Employee_.salary), 1.1f))
    .set(e.get(Employee_.commission), cb.prod(e.get(Employee_.commission), 1.1f))
    .set(e.get(Employee_.bonus), cb.sum(e.get(Employee_.bonus), 5000))
    .where(cb.equal(e.get(Employee_.dept).get(Department_.name), "Sales"));
```

The following Jakarta Persistence query language update statement is equivalent.

```
UPDATE Employee e
SET e.salary = e.salary * 1.1,
    e.commission = e.commission * 1.1,
    e.bonus = e.bonus + 5000
WHERE e.dept.name = 'Sales'
```

Example 4:

```
CriteriaDelete<Customer> q = cb.createCriteriaDelete(Customer.class);
Root<Customer> c = q.from(Customer.class);

q.where(
    cb.equal(c.get(Customer_.status), "inactive"),
    cb.isEmpty(c.get(Customer_.orders)));
```

The following Jakarta Persistence query language delete statement is equivalent.

```
DELETE
FROM Customer c
WHERE c.status = 'inactive'
    AND c.orders IS EMPTY
```

Like bulk update and delete operations made through the Jakarta Persistence query language, criteria API bulk update and delete operations map directly to database operations, bypassing any optimistic locking checks. Portable applications using bulk update operations must manually update the value of the version column, if desired, and/or manually validate the value of the version column.

The persistence context is not synchronized with the result of the bulk update or delete. See Section 4.11.

6.4. Constructing Strongly-typed Queries using the jakarta.persistence.metamodel Interfaces

Strongly-typed queries can also be constructed, either statically or dynamically, in the absence of generated metamodel classes. The `jakarta.persistence.metamodel` interfaces are used to access the metamodel objects that correspond to the managed classes.

The following examples illustrate this approach. These are equivalent to the example queries shown in Section 6.3.5.

The `Metamodel` interface is obtained from the EntityManager or EntityManagerFactory for the persistence unit, and then used to obtain the corresponding metamodel objects for the managed types referenced by the queries.

Example 1:

```
EntityManager em = ...;

Metamodel mm = em.getMetamodel();
EntityType<Employee> emp_ =mm.entity(Employee.class);
EmbeddableType<ContactInfo> cinfo_ = mm.embeddable(ContactInfo.class);
EntityType<Phone> phone_ = mm.entity(Phone.class);
EmbeddableType<Address> addr_ = mm.embeddable(Address.class);

CriteriaQuery<Vendor> q = cb.createQuery(Vendor.class);
Root<Employee> emp = q.from(Employee.class);
Join<Employee, ContactInfo> cinfo =
    emp.join(emp_.getSingularAttribute("contactInfo", ContactInfo.class));
Join<ContactInfo, Phone> p =
    cinfo.join(cinfo_.getSingularAttribute("phones", Phone.class));
q.where(
    cb.equal(emp.get(emp_.getSingularAttribute("contactInfo", ContactInfo.class))
        .get(cinfo_.getSingularAttribute("address", Address.class))
        .get(addr_.getSingularAttribute("zipcode", String.class)), "95054"))
    .select(p.get(phone_.getSingularAttribute("vendor",Vendor.class)));
```

Example 2:

```
EntityManager em = ...;
Metamodel mm = em.getMetamodel();

EntityType<Item> item_ = mm.entity(Item.class);
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Item> item = q.from(Item.class);
MapJoin<Item, String, Object> photo =
    item.join(item_.getMap("photos", String.class, Object.class));
q.multiselect(
    item.get(item_.getSingularAttribute("name", String.class)), photo)
        .where(cb.like(photo.key(), "%egret%"));
```

6.5. Use of the Criteria API with Strings to Reference Attributes

The Criteria API provides the option of specifying the attribute references used in joins and navigation by attribute names used as arguments to the various `join`, `fetch`, and `get` methods.

The resulting queries have the same semantics as described in Section 6.3, but do not provide the same level of type safety.

The examples in this section illustrate this approach. These examples are derived from among those of sections Section 6.3.3 and Section 6.3.5.

Example 1:

```
CriteriaBuilder cb = ...
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Customer> cust = q.from(Customer.class);
Join<Order, Item> item = cust.join("orders").join("lineItems");
q.select(cust.<String>get("name"))
    .where(cb.equal(item.get("product").get("productType"), "printer"));
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT c.name
FROM Customer c JOIN c.orders o JOIN o.lineItems i
WHERE i.product.productType = 'printer'
```

It is not required that type parameters be used. However, their omission may result in compiler warnings, as with the below version of the same query:

```
CriteriaBuilder cb = ...
CriteriaQuery q = cb.createQuery();
Root cust = q.from(Customer.class);
Join item = cust.join("orders").join("lineItems");
q.select(cust.get("name")).where(
    cb.equal(item.get("product").get("productType"),"printer"));
```

Example 2:

The following query uses an outer join:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> cust = q.from(Customer.class);
Join<Customer,Order> order = cust.join("orders", JoinType.LEFT);
q.where(cb.equal(cust.get("status"), 1))
    .select(cust);
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT c FROM Customer c LEFT JOIN c.orders o
WHERE c.status = 1
```

Example 3:

In the following example, `ContactInfo` is an embeddable class consisting of an address and set of phones. `Phone` is an entity.

```
CriteriaQuery<Vendor> q = cb.createQuery(Vendor.class);
Root<Employee> emp = q.from(Employee.class);
Join<ContactInfo, Phone> phone = emp.join("contactInfo").join("phones");
q.where(cb.equal(emp.get("contactInfo")
    .get("address")
    .get("zipcode"), "95054"));
q.select(phone.<Vendor>get("vendor"));
```

The following Jakarta Persistence query language query is equivalent:

```
SELECT p.vendor
FROM Employee e JOIN e.contactInfo.phones p
WHERE e.contactInfo.address.zipcode = '95054'
```

Example 4:

In this example, the `photos` attribute corresponds to a map from photo label to filename. The map key is a string, the value an object.

```
CriteriaQuery<Object> q = cb.createQuery();
Root<Item> item = q.from(Item.class);
MapJoin<Item, String, Object> photo = item.joinMap("photos");
q.multiselect(item.get("name"), photo)
    .where(cb.like(photo.key(), "%egret%"));
```

This query is equivalent to the following Jakarta Persistence query language query:

```
SELECT i.name, p
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE '%egret%'
```

6.6. Query Modification

A `CriteriaQuery`, `CriteriaUpdate`, or `CriteriaDelete` object may be modified, either before or after `Query` or `TypedQuery` objects have been created and executed from it. For example, such modification may entail replacement of the `where` predicate or the `select` list. Modifications may thus result in the same query object "base" being reused for several query instances.

For example, the user might create and execute a query from the following `CriteriaQuery` object:

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> c = q.from(Customer.class);

Predicate pred = cb.equal(c.get(Customer_.address).get(Address_.city),"Chicago");

q.select(c);
q.where(pred);
```

The `CriteriaQuery` object might then be modified to reflect a different predicate condition, for example:

```
Predicate pred2 = cb.gt(c.get(Customer_.balanceOwed), 1000);
q.where(pred2);
```

Note, however, that query elements—in this example, predicate conditions—are dependent on the `CriteriaQuery`, `CriteriaUpdate`, or `CriteriaDelete` instance, and are thus not portably reusable with different instances.

6.7. Query Execution

A criteria query is executed by passing the `CriteriaQuery`, `CriteriaUpdate`, or `CriteriaDelete` object to the `createQuery` method of the `EntityManager` interface to create an executable `TypedQuery` object (or, in the case of `CriteriaUpdate` and `CriteriaDelete`, a `Query` object), which can then be passed to one of the query execution methods of the `TypedQuery` or `Query` interface.

A `CriteriaQuery`, `CriteriaUpdate`, or `CriteriaDelete` object may be further modified after an executable query object has

been created from it. The modification of the `CriteriaQuery`, `CriteriaUpdate`, or `CriteriaDelete` object does not have any impact on the already created executable query object. If the modified `CriteriaQuery`, `CriteriaUpdate`, or `CriteriaDelete` object is passed to the `createQuery` method, the persistence provider must ensure that a new executable query object is created and returned that reflects the semantics of the changed query definition.

`CriteriaQuery`, `CriteriaUpdate`, and `CriteriaDelete` objects must be serializable. A persistence vendor is required to support the subsequent deserialization of such an object into a separate JVM instance of that vendor's runtime, where both runtime instances have access to any required vendor implementation classes. `CriteriaQuery`, `CriteriaUpdate`, and `CriteriaDelete` objects are not required to be interoperable across vendors.

[1] The attributes of these metamodel objects play a role analogous to that which would be played by member literals.

[2] Metamodel objects are used to specify typesafe nagivation through joins and through path expressions. These metamodel objects capture both the source and target types of the attribute through which navigation occurs, and are thus the mechanism by which typesafe navigation is achieved.

[3] Attribute names serve this role for string-based queries. See Section 6.5.

[4] Attribute names serve this role for string-based queries. See Section 6.5.

Chapter 7. Entity Managers and Persistence Contexts

7.1. Persistence Contexts

A persistence context is a set of managed entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed by the entity manager.

In Jakarta EE environments, a JTA (Jakarta Transactions)[8] transaction typically involves calls across multiple components. Such components may often need to access the same persistence context within a single transaction. To facilitate such use of entity managers in Jakarta EE environments, when an entity manager is injected into a component or looked up directly in the JNDI naming context, its persistence context will automatically be propagated with the current JTA transaction, and the `EntityManager` references that are mapped to the same persistence unit will provide access to this same persistence context within the JTA transaction. This propagation of persistence contexts by the Jakarta EE container avoids the need for the application to pass references to `EntityManager` instances from one component to another. An entity manager for which the container manages the persistence context in this manner is termed a *container-managed entity manager*. A container-managed entity manager's lifecycle is managed by the Jakarta EE container.

In less common use cases within Jakarta EE environments, applications may need to access a persistence context that is "stand-alone"—i.e. not propagated along with the JTA transaction across the `EntityManager` references for the given persistence unit. Instead, each instance of creating an entity manager causes a new isolated persistence context to be created that is not accessible through other `EntityManager` references within the same transaction. These use cases are supported through the `createEntityManager` methods of the `EntityManagerFactory` interface. An entity manager that is used by the application to create and destroy a persistence context in this manner is termed an *application-managed entity manager*. An application-managed entity manager's lifecycle is managed by the application.

Both container-managed entity managers and application-managed entity managers and their persistence contexts are required to be supported in Jakarta EE web containers and EJB containers. Within an EJB environment, container-managed entity managers are typically used.

In Java SE environments and in Jakarta EE application client containers, only application-managed entity managers are required to be supported[1].

7.2. Obtaining an EntityManager

The entity manager for a persistence context is obtained from an entity manager factory.

When container-managed entity managers are used (in Jakarta EE environments), the application does not interact with the entity manager factory. The entity managers are obtained directly through dependency injection or from JNDI, and the container manages interaction with the entity manager factory transparently to the application.

When application-managed entity managers are used, the application must use the entity manager factory to manage the entity manager and persistence context lifecycle.

An entity manager must not be shared among multiple concurrently executing threads, as the entity manager and persistence context are not required to be threadsafe. Entity managers must only be accessed in a single-threaded manner.

7.2.1. Obtaining an Entity Manager in the Jakarta EE Environment

A container-managed entity manager is obtained by the application through dependency injection or through direct

lookup of the entity manager in the JNDI namespace. The container manages the persistence context lifecycle and the creation and the closing of the entity manager instance transparently to the application.

The `PersistenceContext` annotation is used for entity manager injection. The `type` element specifies whether a transaction-scoped or extended persistence context is to be used, as described in Section 7.7. The `synchronization` element specifies whether the persistence context is always automatically joined to the current transaction (the default) or is not joined to the current transaction unless the `joinTransaction` method is invoked by the application. The `unitName` element may optionally be specified to designate the persistence unit whose entity manager factory is used by the container. The semantics of the persistence context synchronization type are further described in Section 7.7.1. Section Section 10.5.3 provides further information about the `unitName` element.

For example,

```
@PersistenceContext
EntityManager em;

@PersistenceContext(type=PersistenceContextType.EXTENDED)
EntityManager orderEM;
```

The JNDI lookup of an entity manager is illustrated below:

```
@Stateless
@PersistenceContext(name="OrderEM")
public class MySessionBean implements MyInterface {
    @Resource
    SessionContext ctx;

    public void doSomething() {
        EntityManager em = (EntityManager) ctx.lookup("OrderEM");

        // ...
    }
}
```

7.2.2. Obtaining an Entity Agent in the Jakarta EE Environment

A container-managed entity agent is obtained by the application through dependency injection or through direct lookup of the entity agent in the JNDI namespace. The container manages its lifecycle and the creation and the closing of the entity agent instance transparently to the application.

The `PersistenceAgent` annotation is used for entity agent injection. The `unitName` element may optionally be specified to designate the persistence unit whose entity manager factory is used by the container. Section Section 10.5.3 provides further information about the `unitName` element.

For example,

```
@PersistenceAgent
EntityAgent agent;
```

The JNDI lookup of an entity agent is illustrated below:

```
@RequestScoped
@PersistenceAgent(name="OrderEA")
public class MyBean implements MyInterface {

    public void doSomething() {
        EntityAgent em = (EntityAgent) ctx.lookup("java:comp/env/OrderEA");
```

```
        // ...
    }
}
```

7.2.3. Obtaining an Application-managed Entity Manager or Entity Agent

An application-managed entity manager or entity agent is obtained by the application from an entity manager factory.

The `EntityManagerFactory` API used to obtain an application-managed entity manager or entity agent is the same, independently of whether this API is used in Jakarta EE or Java SE environments.

7.3. Obtaining an Entity Manager Factory

The `EntityManagerFactory` interface is used by the application to create an application-managed entity manager or entity agent[2].

Every entity manager obtained from a given `EntityManagerFactory` is configured in the same manner—connecting to the same database, using the same initial settings as defined by the implementation, and so on.

More than one entity manager factory instance may be available simultaneously in the JVM.[3]

Methods of the `EntityManagerFactory` interface are threadsafe.

7.3.1. Obtaining an Entity Manager Factory in a Jakarta EE Container

Within a Jakarta EE environment, an entity manager factory can be injected using the `PersistenceUnit` annotation or obtained through JNDI lookup. The `unitName` element may optionally be specified to designate the persistence unit whose entity manager factory is used. (See Section 10.5.3).

For example,

```
@PersistenceUnit
EntityManagerFactory emf;
```

7.3.2. Obtaining an Entity Manager Factory in a Java SE Environment

Outside a Jakarta EE container environment, the `Persistence` class described in Section 9.7 is the bootstrap class providing access to an entity manager factory. The application creates an entity manager factory by calling the `createEntityManagerFactory` method of the `Persistence` class.

For example,

```
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("Order");
EntityManager em = emf.createEntityManager();
```

7.3.3. Obtaining an Entity Manager Factory for a programmatically-defined persistence unit

The class `PersistenceConfiguration` described in Section 9.8 may be used to programmatically define and configure a persistence unit (see Chapter 8), as an alternative to packaging a `persistence.xml` file, mapping files, and classes inside an archive as described in Section 8.1.

An `EntityManagerFactory` may be obtained directly from the `PersistenceConfiguration`.

For example,

```
DataSource datasource = (DataSource)
        new InitialContext()
                .lookup("java:global/jdbc/MyOrderDB");
EntityManagerFactory emf =
        new PersistenceConfiguration()
                .name("OrderManagement")
                .jtaDataSource(datasource)
                .mappingFile("ormap.xml")
                .managedClass(Order.class)
                .managedClass(Customer.class)
                .createEntityManagerFactory();
```

7.4. EntityManagerFactory Interface

The `EntityManagerFactory` interface found in Section B.5

An `EntityManagerFactory` may be used by the application to obtain an application-managed entity manager. When the application has finished using the entity manager factory, and/or at application shutdown, the application should close the entity manager factory. Once an entity manager factory has been closed, all its entity managers are considered to be in the closed state.

An `EntityManagerFactory` also provides access to information and services that are global to the persistence unit. This includes access to the second level cache that is maintained by the persistence provider and to the `PersistenceUnitUtil` interface. The `Cache` interface is described in Section 3.14.3; the `PersistenceUnitUtil` interface in Section 7.11.

Any number of vendor-specific properties may be included in the map passed to the `createEntityManager` methods. Properties that are not recognized by a vendor must be ignored.

Note that the policies of the installation environment may restrict some information from being made available through the `EntityManagerFactory` `getProperties` method (for example, JDBC user, password, URL).

Vendors should use vendor namespaces for properties (e.g., `com.acme.persistence.logging`). Entries that make use of the namespace `jakarta.persistence` and its subnamespaces must not be used for vendor-specific information. The namespace `jakarta.persistence` is reserved for use by this specification.

7.5. Controlling Transactions

Every entity manager factory is defined to be of a given transactional type—either JTA or resource-local—at the time it is configured and created. See sections Section 8.2.1.2 and Section 9.1.

Depending on the transactional type of the entity manager factory, transactions involving `EntityManager` or `EntityAgent` operations may be controlled either through JTA or through use of the resource-local `EntityTransaction` API.

- An entity manager or entity agent whose underlying transactions are controlled through JTA is termed a *JTA entity manager or agent*. A JTA entity manager participates in the current JTA transaction, which is begun and committed externally to the entity manager and propagated to the underlying resource manager.
- An entity manager or entity agent whose underlying transactions are controlled by the application through the `EntityTransaction` API is termed a *resource-local entity manager or agent*. A resource-local transaction is mapped to a transaction over the underlying resource by the persistence provider. Resource-local entity managers may use server or local resources to connect to the database and are unaware of the presence of JTA transactions that may or may not be active.

A container-managed entity manager or entity agent must be a JTA entity manager or agent. An application-managed entity manager or entity agent may be of either transactional type, depending on the transactional type of its entity manager factory.

JTA entity managers and entity agents are only specified for use in Jakarta EE containers. This specification does not address the use of JTA outside the Jakarta EE environment.

Both JTA entity managers and resource-local entity managers are required to be supported in Jakarta EE web containers and EJB containers. Within an EJB environment, a JTA entity manager is typically used. In general, in Java SE environments only resource-local entity managers are supported.

7.5.1. The EntityTransaction Interface

The `EntityTransaction` interface found in Section B.4 is used to control resource transactions on resource-local entity managers. The `getTransaction()` method of `EntityHandler` returns an instance of the `EntityTransaction` interface.

When a resource-local entity manager is used, and the persistence provider runtime throws an exception defined to cause transaction rollback, the persistence provider must mark the transaction for rollback.

If the `EntityTransaction.commit` operation fails, the persistence provider must roll back the transaction.

The following example illustrates the creation of an entity manager factory in a Java SE environment, and its use in creating and using a resource-local entity manager.

```
import jakarta.persistence.*;

public class PasswordChanger {
    public static void main (String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("Order");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        User user = em.createQuery
            ("SELECT u FROM User u WHERE u.name=:name AND u.pass=:pass", User.class)
            .setParameter("name", args[0])
            .setParameter("pass", args[1])
            .getSingleResult();

        user.setPassword(args[2]);

        em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

7.6. The runInTransaction and callInTransaction methods

The `runInTransaction` and `callInTransaction` methods of the `EntityManagerFactory` provide a shortcut for persistence context and transaction management with an application-managed `EntityManager`.

```
entityManagerFactory.runInTransaction(entityManager -> {
    User user = em.createQuery
        ("SELECT u FROM User u WHERE u.name=:name AND u.pass=:pass", User.class)
        .setParameter("name", args[0])
        .setParameter("pass", args[1])
        .getSingleResult();

    user.setPassword(args[2]);
})
```

The argument function passed to `runInTransaction` or `callInTransaction` must be called and passed a new instance of `EntityManager`. When the argument function returns or throws an exception, this `EntityManager` must be closed before `runInTransaction` or `callInTransaction` returns.

The argument function is executed in the context of a transaction associated with this new `EntityManager`.

- If the transaction type of the persistence unit is JTA, and there is a JTA transaction already associated with the caller, then the `EntityManager` is associated with this current transaction. If the argument function throws an exception, the JTA transaction must be marked for rollback, and the exception must be rethrown by `runInTransaction` or `callInTransaction`. Otherwise, `callInTransaction` must return the same value returned by the argument function.
- Otherwise, if the transaction type of the persistence unit is resource-local, or if there is no JTA transaction already associated with the caller, then the `EntityManager` is associated with a new transaction. If the argument function throws an exception, this transaction must be rolled back, and then the exception must be rethrown by `runInTransaction` or `callInTransaction`. If the argument function returns, then `runInTransaction` or `callInTransaction` must attempt to commit the transaction. If the attempt to commit the transaction fails, the exception must be rethrown. Otherwise, `callInTransaction` must return the same value returned by the argument function.

The application should not attempt to manage the lifecycle of the transaction or `EntityManager` directly. If the application calls an operation of `EntityTransaction` from within a call to `runInTransaction` or `callInTransaction`, the behavior is undefined.

7.7. Container-managed Persistence Contexts

When a container-managed entity manager is used, the lifecycle of the persistence context is always managed automatically, transparently to the application, and the persistence context is propagated with the JTA transaction.

A container-managed persistence context may be defined to have either a lifetime that is scoped to a single transaction or an extended lifetime that spans multiple transactions, depending on the `PersistenceContextType` that is specified when its entity manager is created. This specification refers to such persistence contexts as *transaction-scoped persistence contexts* and *extended persistence contexts* respectively.

The lifetime of the persistence context is declared using the `PersistenceContext` annotation or the `persistence-context-ref` deployment descriptor element. By default, a transaction-scoped persistence context is used.

Sections Section 7.7.2 and Section 7.7.3 describe transaction-scoped and extended persistence contexts in the absence of persistence context propagation. Persistence context propagation is described in Section 7.7.4.

Persistence contexts are always associated with an entity manager factory. In the following sections, “the persistence context” should be understood to mean “the persistence context associated with a particular entity manager factory”.

7.7.1. Persistence Context Synchronization Type

By default, a container-managed persistence context is of type `SynchronizationType.SYNCHRONIZED`. Such a persistence context is automatically joined to the current JTA transaction, and updates made to the persistence context are propagated to the underlying resource manager.

A container-managed persistence context may be specified to be of type `SynchronizationType.UNSYNCHRONIZED`. A persistence context of type `SynchronizationType.UNSYNCHRONIZED` is not enlisted in any JTA transaction unless explicitly joined to that transaction by the application. A persistence context of type `SynchronizationType.UNSYNCHRONIZED` is enlisted in a JTA transaction and registered for subsequent transaction notifications against that transaction by the invocation of the `EntityManager joinTransaction` method. The persistence context remains joined to the transaction until the transaction commits or rolls back. After the transaction commits or rolls back, the persistence context will not be joined to any subsequent transaction unless the `joinTransaction` method is invoked in the scope of that subsequent transaction.

A persistence context of type `SynchronizationType.UNSYNCHRONIZED` must not be flushed to the database unless it is joined to a transaction. The application's use of queries with pessimistic locks, bulk update or delete queries, etc. result in the provider throwing the `TransactionRequiredException`. After the persistence context has been joined to the JTA transaction, these operations are again allowed.

The application is permitted to invoke the persist, merge, remove, and refresh entity lifecycle operations on an entity manager of type `SynchronizationType.UNSYNCHRONIZED` independent of whether the persistence context is joined to the current transaction. After the persistence context has been joined to a transaction, changes in a persistence context can be flushed to the database either explicitly by the application or by the provider. If the `flush` method is not explicitly invoked, the persistence provider may defer flushing until commit time depending on the operations invoked and the flush mode setting in effect.

If an extended persistence context of type `SynchronizationType.UNSYNCHRONIZED` has not been joined to the current JTA transaction, rollback of the JTA transaction will have no effect upon the persistence context. In general, it is recommended that a non-JTA datasource be specified for use by the persistence provider for a persistence context of type `SynchronizationType.UNSYNCHRONIZED` that has not been joined to a JTA transaction in order to alleviate the risk of integrating uncommitted changes into the persistence context in the event that the transaction is later rolled back.

If a persistence context of type `SynchronizationType.UNSYNCHRONIZED` has been joined to the JTA transaction, transaction rollback will cause the persistence context to be cleared and all pre-existing managed and removed instances to become detached. (See Section 3.7.5.)

When a JTA transaction exists, a persistence context of type `SynchronizationType.UNSYNCHRONIZED` is propagated with that transaction according to the rules in Section 7.7.4.1 regardless of whether the persistence context has been joined to that transaction.

7.7.2. Container-managed Transaction-scoped Persistence Context

The application can obtain a container-managed entity manager with transaction-scoped persistence context by injection or direct lookup in the JNDI namespace. The persistence context type for the entity manager must be defaulted or explicitly specified as `PersistenceContextType.TRANSACTION`.

A new persistence context begins when the container-managed entity manager is invoked[4] in the scope of an active JTA transaction, and there is no current persistence context already associated with the transaction. The persistence context is created and associated with the transaction. This affinity of the persistence context with the JTA transaction is independent of the synchronization type of the persistence context and of whether the persistence context has been joined to the transaction.

The persistence context ends when the associated transaction commits or rolls back, and all entities that were managed by the `EntityManager` become detached.[5]

If the entity manager is invoked outside the scope of a transaction, all entities loaded from the database immediately become detached at the end of the method call.

7.7.3. Container-managed Extended Persistence Context

A container-managed extended persistence context can only be initiated within the scope of a stateful session bean which declares as dependency on the extended persistence context by means of the `PersistenceContext` annotation or `persistence-context-ref` deployment descriptor element.

The extended persistence context exists from the point at which the stateful session bean declaring a dependency on the persistence context is created and is said to be *bound* to the stateful session bean.

The container destroys the extended persistence context after the `@Remove` method of the stateful session bean completes

or when the stateful session bean instance is otherwise destroyed.

7.7.3.1. Inheritance of Extended Persistence Context

If a stateful session bean instantiates a stateful session bean (executing in the same EJB container instance) which also declares a dependency on an extended persistence context with the same unit and synchronization type, the extended persistence context of the first stateful session bean is inherited by the second stateful session bean.

This rule applies recursively, independently of whether transactions are active at the moment of the creation of the stateful session beans. If the stateful session beans differ in declared synchronization type, an `EJBException` is thrown by the container.

The container does not destroy an extended persistence context until every stateful session bean inheriting the persistence context has been removed or otherwise destroyed.

7.7.4. Persistence Context Propagation

As described in Section 7.1, a single persistence context may correspond to one or more JTA entity manager instances (all associated with the same entity manager factory[6]).

The persistence context is propagated across the entity manager instances as the JTA transaction is propagated. A persistence context of type `SynchronizationType.UNSYNCHRONIZED` is propagated with the JTA transaction regardless of whether it has been joined to the transaction.

Propagation of persistence contexts only applies within a local environment. Persistence contexts are not propagated to remote tiers.

7.7.4.1. Requirements for Persistence Context Propagation

Persistence contexts are propagated by the container across component invocations as follows.

If a component is called and there is no JTA transaction or the JTA transaction is not propagated, the persistence context is not propagated.

- If an entity manager is then invoked from within the component:
  - Invocation of an entity manager defined with `PersistenceContextType.TRANSACTION` will result in use of a new persistence context (as described in Section 7.7.2).
  - Invocation of an entity manager defined with `PersistenceContextType.EXTENDED` will result in the use of the existing extended persistence context bound to that component.
  - If the entity manager is invoked within a JTA transaction, the persistence context will be associated with the JTA transaction.

If a component is called and the JTA transaction is propagated into that component:

- If the component is a stateful session bean to which an extended persistence context has been bound and there is a different persistence context associated with the JTA transaction, an `EJBException` is thrown by the container.
- If there is a persistence context of type `SynchronizationType.UNSYNCHRONIZED` associated with the JTA transaction and the target component specifies a persistence context of type `SynchronizationType.SYNCHRONIZED`, the `IllegalStateException` is thrown by the container.
- Otherwise, if there is a persistence context associated with the JTA transaction, that persistence context is propagated and used.

Note that a component with a persistence context of type `SynchronizationType.UNSYNCHRONIZED` may be

> called by a component propagating either a persistence context of type `SynchronizationType.UNSYNCHRONIZED` or a persistence context of type `SynchronizationType.SYNCHRONIZED` into it.

The following example shows a container-managed, transaction-scoped persistence context:

```
@Stateless
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceContext
    EntityManager em;

    public Order getOrder(Long id) {
        Order order = em.find(Order.class, id);
        order.getLineItems();
        return order;
    }

    public Product getProduct(String name) {
        return (Product) em.createQuery("select p from Product p where p.name = : name")
                .setParameter("name", name)
                .getSingleResult();
    }

    public LineItem createLineItem(Order order, Product product, int quantity) {
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        return li;
    }
}
```

This example shows a container-managed extended persistence context:

```
/*
 * An extended transaction context is used. The entities remain
 * managed in the persistence context across multiple transactions.
 */
@Stateful
@Transaction(REQUIRES_NEW)
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceContext(type = EXTENDED)
    EntityManager em;

    private Order order;
    private Product product;

    public void initOrder(Long id) {
        order = em.find(Order.class, id);
    }

    public void initProduct(String name) {
        product = (Product) em.createQuery("select p from Product p where p.name = : name")
                  .setParameter("name", name)
                  .getSingleResult();
    }

    public LineItem createLineItem(int quantity) {
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        return li;
    }
}
```

7.8. Application-managed Persistence Contexts

When an application-managed entity manager is used, the application interacts directly with the persistence provider's entity manager factory to manage the entity manager lifecycle and to obtain and destroy persistence contexts.

All such application-managed persistence contexts are extended in scope, and can span multiple transactions.

The `EntityManagerFactory.createEntityManager` method and the `EntityManager close` and `isOpen` methods are used to manage the lifecycle of an application-managed entity manager and its associated persistence context.

The extended persistence context exists from the point at which the entity manager has been created using `EntityManagerFactory.createEntityManager` until the entity manager is closed by means of `EntityManager.close`.

An extended persistence context obtained from the application-managed entity manager is a stand-alone persistence context—it is not propagated with the transaction.

When a JTA application-managed entity manager is used, an application-managed persistence context may be specified to be of type `SynchronizationType.UNSYNCHRONIZED`. A persistence context of type `SynchronizationType.UNSYNCHRONIZED` is not enlisted in any JTA transaction unless explicitly joined to that transaction by the application. A persistence context of type `SynchronizationType.UNSYNCHRONIZED` is enlisted in a JTA transaction and registered for subsequent transaction notifications against that transaction by the invocation of the `EntityManager joinTransaction` method. The persistence context remains joined to the transaction until the transaction commits or rolls back. After the transaction commits or rolls back, the persistence context will not be joined to any subsequent transaction unless the `joinTransaction` method is invoked in the scope of that subsequent transaction.

When a JTA application-managed entity manager is used, if the entity manager is created outside the scope of the current JTA transaction, it is the responsibility of the application to join the entity manager to the transaction (if desired) by calling `EntityManager.joinTransaction`. If the entity manager is created outside the scope of a JTA transaction, it is not joined to the transaction unless `EntityManager.joinTransaction` is called.

The `EntityManager.close` method closes an entity manager to release its persistence context and other resources. After calling `close`, the application must not invoke any further methods on the `EntityManager` instance except for `getTransaction` and `isOpen`, or the `IllegalStateException` will be thrown. If the `close` method is invoked when a transaction is active, the persistence context remains managed until the transaction completes.

The `EntityManager.isOpen` method indicates whether the entity manager is open. The `isOpen` method returns true until the entity manager has been closed.

This example shows an application-managed persistence context used in a stateless session bean:

```
/*
 * Container-managed transaction demarcation is used.
 * The session bean creates and closes an entity manager
 * in each business method.
 */
@Stateless
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceUnit
    private EntityManagerFactory emf;

    public Order getOrder(Long id) {
        EntityManager em = emf.createEntityManager();
        Order order = em.find(Order.class, id);
        order.getLineItems();
        em.close();
        return order;
    }

    public Product getProduct() {
```

```
        EntityManager em = emf.createEntityManager();
        Product product = (Product)
                          em.createQuery("select p from Product p where p.name = :name")
                          .setParameter("name", name)
                          .getSingleResult();
        em.close();
        return product;
    }

    public LineItem createLineItem(Order order, Product product, int quantity) {
        EntityManager em = emf.createEntityManager();
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        em.close();
        return li; // remains managed until JTA transaction ends
    }
}
```

This examples shows an application-managed persistence context used in a stateless session bean:

```
/*
 * Container-managed transaction demarcation is used.
 * The session bean creates entity manager in PostConstruct
 * method and clears persistence context at the end of each
 * business method.
 */
@Stateless
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceUnit
    private EntityManagerFactory emf;

    private EntityManager em;

    @PostConstruct
    public void init() {
        em = emf.createEntityManager();
    }

    public Order getOrder(Long id) {
        Order order = em.find(Order.class, id);
        order.getLineItems();
        em.clear(); // entities are detached
        return order;
    }

    public Product getProduct() {
        Product product = (Product)
                          em.createQuery("select p from Product p where p.name = :name")
                          .setParameter("name", name)
                          .getSingleResult();
        em.clear();
        return product;
    }

    public LineItem createLineItem(Order order, Product product, int quantity) {
        em.joinTransaction();
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        // persistence context is flushed to database;
        // all updates will be committed to database on tx commit
        em.flush();
        // entities in persistence context are detached
        em.clear();
        return li;
```

```
    }

    @PreDestroy
    public void destroy() {
        em.close();
    }
}
```

This example shows an application-managed persistence context used in a stateful session bean:

```
/*
 * Container-managed transaction demarcation is used.
 * Entities remain managed until the entity manager is closed.
 */
@Stateful
public class ShoppingCartImpl implements ShoppingCart {
    @PersistenceUnit
    private EntityManagerFactory emf;

    private EntityManager em;

    private Order order;

    private Product product;

    @PostConstruct
    public void init() {
        em = emf.createEntityManager();
    }

    public void initOrder(Long id) {
        order = em.find(Order.class, id);
    }

    public void initProduct(String name) {
        product = (Product) em.createQuery("select p from Product p where p.name = : name")
                  .setParameter("name", name)
                  .getSingleResult();
    }

    public LineItem createLineItem(int quantity) {
        em.joinTransaction();
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        return li;
    }

    @Remove
    public void destroy() {
        em.close();
    }
}
```

Finally, this example shows an application-managed persistence context used with a resource transaction:

```
// Usage in an ordinary Java class
public class ShoppingImpl {
    private EntityManager em;
    private EntityManagerFactory emf;

    public ShoppingCart() {
        emf = Persistence.createEntityManagerFactory("orderMgt");
        em = emf.createEntityManager();
    }
```

```
    private Order order;
    private Product product;

    public void initOrder(Long id) {
        order = em.find(Order.class, id);
    }

    public void initProduct(String name) {
        product = (Product) em.createQuery("select p from Product p where p.name = : name")
                  .setParameter("name", name)
                  .getSingleResult();
    }

    public LineItem createLineItem(int quantity) {
        em.getTransaction().begin();
        LineItem li = new LineItem(order, product, quantity);
        order.getLineItems().add(li);
        em.persist(li);
        em.getTransaction().commit();
        return li;
    }

    public void destroy() {
        em.close();
        emf.close();
    }
}
```

7.9. Requirements on the Container

7.9.1. Application-managed Persistence Contexts

When application-managed persistence contexts are used, the container must instantiate the entity manager factory and expose it to the application via JNDI. The container might use internal APIs to create the entity manager factory, or it might use the `PersistenceProvider.createContainerEntityManagerFactory` method. However, the container is required to support third-party persistence providers, and in this case the container must use the `PersistenceProvider.createContainerEntityManagerFactory` method to create the entity manager factory and the `EntityManagerFactory.close` method to destroy the entity manager factory prior to shutdown (if it has not been previously closed by the application).

7.9.2. Container Managed Persistence Contexts

The container is responsible for managing the lifecycle of container-managed persistence contexts, for injecting `EntityManager` references into web components and session bean and message-driven bean components, and for making `EntityManager` references available to direct lookups in JNDI.

When operating with a third-party persistence provider, the container uses the contracts defined in Section 7.10 to create and destroy container-managed persistence contexts. It is undefined whether a new entity manager instance is created for every persistence context, or whether entity manager instances are sometimes reused. Exactly how the container maintains the association between persistence context and JTA transaction is not defined.

If a persistence context is already associated with a JTA transaction, the container uses that persistence context for subsequent invocations within the scope of that transaction, according to the semantics for persistence context propagation defined in Section 7.7.4.

7.10. Runtime Contracts between the Container and Persistence Provider

This section describes contracts between the container and the persistence provider for the pluggability of third-party persistence providers. Containers are required to support these pluggability contracts.[7]

7.10.1. Container Responsibilities

For the management of a transaction-scoped persistence context, if there is no EntityManager already associated with the JTA transaction:

- The container creates a new entity manager by calling `EntityManagerFactory.createEntityManager` when the first invocation of an entity manager with `PersistenceContextType.TRANSACTION` occurs within the scope of a business method executing in the JTA transaction.
- After the JTA transaction has completed (either by transaction commit or rollback), the container closes the entity manager by calling `EntityManager.close`. [8] Note that the JTA transaction may rollback in a background thread (e.g., as a result of transaction timeout), in which case the container should arrange for the entity manager to be closed but the `EntityManager.close` method should not be concurrently invoked while the application is in an `EntityManager` invocation.

The container must throw the `TransactionRequiredException` if a transaction-scoped persistence context is used and the `EntityManager` `persist`, `remove`, `merge`, or `refresh` method is invoked when no transaction is active.

For stateful session beans with extended persistence contexts:

- The container creates an entity manager by calling `EntityManagerFactory.createEntityManager` when a stateful session bean is created that declares a dependency on an entity manager with `PersistenceContextType.EXTENDED`. (See Section 7.7.3).
- The container closes the entity manager by calling `EntityManager.close` after the stateful session bean and all other stateful session beans that have inherited the same persistence context as the entity manager have been removed.
- When a business method of the stateful session bean is invoked, if the stateful session bean uses container managed transaction demarcation, and the entity manager is not already associated with the current JTA transaction, the container associates the entity manager with the current JTA transaction and, if the persistence context is of type `SynchronizationType.SYNCHRONIZED`, the container calls `EntityManager.joinTransaction`. If there is a different persistence context already associated with the JTA transaction, the container throws the `EJBException`.
- When a business method of the stateful session bean is invoked, if the stateful session bean uses bean managed transaction demarcation and a UserTransaction is begun within the method, the container associates the persistence context with the JTA transaction and, if the persistence context is of type `SynchronizationType.SYNCHRONIZED`, the container calls `EntityManager.joinTransaction`.

The container must throw the `IllegalStateException` if the application calls `EntityManager.close` on a container-managed entity manager.

When the container creates an entity manager, it may pass a map of properties to the persistence provider by using the `EntityManagerFactory.createEntityManager(Map map)` method. If properties have been specified in the `PersistenceContext` annotation or the `persistence-context-ref` deployment descriptor element, this method must be used and the map must include the specified properties.

If the application invokes `EntityManager.unwrap(Class<T> cls)`, and the container cannot satisfy the request, the container must delegate the `unwrap` invocation to the provider's entity manager instance.

7.10.2. Provider Responsibilities

The Provider has no knowledge of the distinction between transaction-scoped and extended persistence contexts. It provides entity managers to the container when requested and registers for transaction synchronization notifications.

- When `EntityManagerFactory.createEntityManager` is invoked, the provider must create and return a new entity manager. If a JTA transaction is active and the persistence context is of type `SynchronizationType.SYNCHRONIZED`, the provider must register for synchronization notifications against the JTA transaction.
- When `EntityManager.joinTransaction` is invoked, the provider must register for synchronization notifications against the current JTA transaction if a previous `joinTransaction` invocation for the transaction has not already been processed.
- When the JTA transaction commits, if the persistence context is of type `SynchronizationType.SYNCHRONIZED` or has otherwise been joined to the transaction, the provider must flush all modified entity state to the database.
- When the JTA transaction rolls back, the provider must detach all managed entities if the persistence context is of type `SynchronizationType.SYNCHRONIZED` or has otherwise been joined to the transaction. Note that the JTA transaction may roll back in a background thread (e.g., as a result of transaction timeout), in which case the provider should arrange for the managed entities to be detached from the persistence context but not concurrently while the application is in an `EntityManager` invocation.
- When the provider throws an exception defined to cause transaction rollback, the provider must mark the transaction for rollback if the persistence context is of type `SynchronizationType.SYNCHRONIZED` or has otherwise been joined to the transaction.
- When `EntityManager.close` is invoked, the provider should release all resources that it may have allocated after any outstanding transactions involving the entity manager have completed. If the entity manager was already in a closed state, the provider must throw the `IllegalStateException`.
- When `EntityManager.clear` is invoked, the provider must detach all managed entities.

7.11. PersistenceUnitUtil Interface

The `PersistenceUnitUtil` interface found in Section B.25 declares utility methods that can be invoked on entities associated with the persistence unit. The behavior is undefined if these methods are invoked on an entity instance that is not associated with the persistence unit from whose entity manager factory this interface has been obtained.

7.12. SchemaManager Interface

The `SchemaManager` interface may be found in Section B.21. An instance of `SchemaManager` may be obtained by calling the `getSchemaManager()` method of `EntityManagerFactory`.

The `SchemaManager` interface allows programmatic control over schema generation and cleanup at runtime. This differs from the functionality described in Section 9.4 which allows schema generation before or during the application deployment and initialization process. Similarly, the `generateSchema` method described in Section 9.2.1 is intended to be called before the `EntityManagerFactory` is available. By contrast, an instance of `SchemaManager` is only available after an `EntityManagerFactory` has already been created.

For example, `SchemaManager` is especially useful in tests.

The methods of `SchemaManager` correspond to values of the property `jakarta.persistence.schema-generation.scripts.action`. The methods `create()`, `drop()`, and `validate()` correspond to the actions `create`, `drop`, and `validate`. The method `truncate()` has no corresponding action. The method `populate()` triggers data loading as described in Section 9.4.2 and corresponds to the action `populate`.

Thus, the behavior of the `SchemaManager` may be controlled via the properties defined in Section 9.4 and Section 8.2.1.12.

[1] Note that the use of JTA is not required to be supported in application client containers.

[2] It may also be used internally by the Jakarta EE container. See Section 7.10.

[3] This is usually the case when the application uses more than one database, since it is typical that each entity manager communicates which exactly one database. On the other hand, there is usually only one entity manager factory for each persistence unit.

[4] That is, when a method of the `EntityManager` interface is invoked.

[5] This also applies to a transaction-scoped persistence context of type `SynchronizationType.UNSYNCHRONIZED` that has not been joined to the transaction.

[6] Entity manager instances obtained from different entity manager factories never share the same persistence context.

[7] It is not required that these contracts be used when a third-party persistence provider is not used: the container might use these same APIs or its might use its own internal APIs.

[8] The container may choose to pool EntityManagers: it instead of creating and closing in each case, it may acquire one from its pool and call `clear()` on it.

Chapter 8. Entity Packaging

A *persistence unit* is a logical grouping that includes:

- an entity manager factory and its entity managers, together with their configuration information,
- the set of managed classes included in the persistence unit and managed by entity managers created by the entity manager factory, and
- mapping metadata (in the form of metadata annotations and/or XML metadata) specifying the mapping of these classes to the database, and
- named queries and named statements involving these managed classes.

This chapter describes the packaging of persistence units.

8.1. Persistence Unit Packaging

Within Jakarta EE environments, any EJB-JAR, WAR, EAR, or application client JAR can define a persistence unit. Any number of persistence units may be defined within these scopes.

A persistence unit may be packaged:

- within one or more jar files contained within a WAR or EAR,
- as a set of classes within an EJB-JAR file or in the WAR `classes` directory, or
- as a combination of these, as defined below.

A persistence unit is defined by a `persistence.xml` file. The jar file or directory whose `META-INF` directory contains the `persistence.xml` file is termed the `root` of the persistence unit. In Jakarta EE environments, the root of a persistence unit must be either:

- an EJB-JAR file,
- the `WEB-INF/classes` directory of a WAR file[1],
- a jar file in the `WEB-INF/lib` directory of a WAR file,
- a jar file in the library directory or an EAR, or
- an application client JAR file.

It is not required that an EJB-JAR or WAR file containing a persistence unit be packaged in an EAR unless the persistence unit contains extra persistence classes in addition to those contained within the EJB-JAR or WAR. See Section 8.2.1.8.

> Java Persistence 1.0 supported the use of a jar file in the root of the EAR as the root of a persistence unit. This use is no longer supported. Portable applications should use the EAR library directory for this case instead. See [6].

A persistence unit must have a name. The name of the persistence unit must be unique within a given EJB-JAR file, within a given WAR file, within a given application client JAR, or within an EAR. See Section 8.3.

The `persistence.xml` file may be used to define more than one persistence unit within the same scope.

All persistence classes defined at the level of the Jakarta EE EAR must be accessible to other Jakarta EE components in the application—that is, to all components loaded by the application classloader—such that if the same entity class is referenced by two different Jakarta EE components (which may be using different persistence units), the referenced class is the same identical class.

In Java SE environments, the metadata mapping files, jar files, and classes described in the following sections can be

used. To ensure the portability of a Java SE application, it is necessary to explicitly list the managed persistence classes included in the persistence unit using the `class` element of the `persistence.xml` file. See Section 8.2.1.8.

8.2. persistence.xml file

A `persistence.xml` file defines a persistence unit. The `persistence.xml` file is located in the `META-INF` directory of the root of the persistence unit. It may be used to specify:

- managed persistence classes included in the persistence unit,
- object/relational mapping information for those classes,
- scripts for use in schema generation and bulk loading of data, and
- other configuration information for the persistence unit and for the entity managers and entity manager factory of the persistence unit.

This information may be defined by containment or by reference, as described below.

The object/relational mapping information can take the form of:

- annotations on the managed persistence classes included in the persistence unit,
- an `orm.xml` file contained in the `META-INF` directory of the root of the persistence unit,
- one or more XML files accessible on the classpath and referenced from the `persistence.xml` file, or
- any combination of the previous options.

The managed persistence classes may be:

- contained within the root of the persistence unit,
- specified by reference—that is, by naming the classes, class archives, or XML mapping files (which in turn reference classes) that are accessible on the application classpath, or
- specified by any combination of these means.

See Section 8.2.1.8.

8.2.1. Elements of persistence.xml

The root element of the `persistence.xml` file is the `persistence` element. The `persistence` element consists of one or more `persistence-unit` elements.

The `persistence-unit` element consists of:

- the `name` and `transaction-type` attributes and
- the following sub-elements: `description`, `provider`, `qualifier`, `scope` `jta-data-source`, `non-jta-data-source`, `mapping-file`, `jar-file`, `class`, `exclude-unlisted-classes`, `shared-cache-mode`, `validation-mode`, and `properties`.

The `name` attribute is required; the other attributes and elements are optional. Their semantics are described in the following subsections.

Examples:

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description>
            This unit manages orders and customers.
            It does not rely on any vendor-specific features and can
            therefore be deployed to any persistence provider.
        </description>
```

```
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <mapping-file>ormap.xml</mapping-file>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>


<persistence>
    <persistence-unit name="OrderManagement2">
        <description>
            This unit manages inventory for auto parts.
            It depends on features provided by the
            com.acme.persistence implementation.
        </description>
        <provider>com.acme.AcmePersistence</provider>
        <jta-data-source>jdbc/MyPartDB</jta-data-source>
        <mapping-file>ormap2.xml</mapping-file>
        <jar-file>MyPartsApp.jar</jar-file>
        <properties>
            <property name="com.acme.persistence.sql-logging" value="on"/>
        </properties>
    </persistence-unit>
</persistence>
```

8.2.1.1. name

The `name` attribute defines the name of the persistence unit. This name is used to identify the persistence unit referred to by a `PersistenceContext` or `PersistenceUnit` annotation and in the programmatic API for creating an entity manager factory.

8.2.1.2. transaction-type

The `transaction-type` attribute specifies whether entity managers and entity agents created by the entity manager factory for the persistence unit are JTA entity managers and entity agents or resource-local entity managers and entity agents. The value of this element must be `JTA` or `RESOURCE_LOCAL`:

- `JTA` means that a JTA data source is provided—either as specified by the `jta-data-source` element, or by the container.
- In a Jakarta EE environment, `RESOURCE_LOCAL` usually means that a non-JTA datasource is provided.

Configuration of datasources is described below in Section 8.2.1.7.

If the `transaction-type` is not explicitly specified, its value is defaulted:

- in a Jakarta EE environment, the default is `JTA`, but
- in a Java SE environment, the default is `RESOURCE_LOCAL`.

8.2.1.3. description

The `description` element provides optional descriptive information about the persistence unit.

8.2.1.4. provider

The `provider` element specifies the name of a provider-specific implementation of `jakarta.persistence.spi.PersistenceProvider`. The `provider` element is optional, but should be explicitly specified if the application depends on the use of a particular persistence provider.

8.2.1.5. qualifier

The `qualifier` element specifies the fully-qualified class name of an annotation annotated `jakarta.inject.Qualifier`. This qualifier annotation may be used to disambiguate the persistence unit for the purposes of dependency injection.

8.2.1.6. scope

The `scope` element specifies the fully-qualified class name of an annotation annotated `jakarta.inject.Scope` or `jakarta.enterprise.context.NormalScope`. This scope annotation may be used to determine the scope of a persistence context for the purposes of dependency injection.

8.2.1.7. jta-data-source, non-jta-data-source

In Jakarta EE environments:

- the `jta-data-source` element specifies the JNDI name of a JTA data source, and/or
- the `non-jta-data-source` element specifies the JNDI name of a non-JTA data source.

The specified data source is used by the persistence provider to obtain database connections. If neither element is specified, the deployer must specify a data source at deployment, or a default data source must be provided by the container.

In Java SE environments, these elements may be used, or the data source information may be specified by other means, depending upon the requirements of the provider.

8.2.1.8. mapping-file, jar-file, class, exclude-unlisted-classes

The following classes must be implicitly or explicitly denoted as managed persistence classes to be included within a persistence unit:

- entity classes;
- embeddable classes;
- mapped superclasses;
- converter classes.

Furthermore, other discoverable types, as defined below in Section 8.2.1.8.1, including types declaring named queries and statements, as defined in Section 3.15.11, must also be implicitly or explicitly denoted as belonging to a persistence unit.

> Terminology Note: In this context, the word "class" refers to compiled Java `.class` files, and implicitly includes both Java types and Java module and package descriptors.

A given type may belong to more than one persistence unit.

The set of types belonging to a persistence unit is determined by:

- classes bearing discoverable annotation types located in the root of the persistence unit (unless `<exclude-unlisted-classes/>` is specified);
- one or more object/relational mapping XML files;
- one or more JAR files to be searched for classes;
- an explicit list of classes.

The set of managed classes belonging to the persistence unit is the union of these sources, with mapping metadata

annotations (or annotation defaults) for any given class being overridden by the XML mapping file information if there are both annotations and XML mappings for that class. The minimum portable level of overriding is at the level of the persistent field or property.

The classes and jars listed as belonging to a persistence unit must be on the classpath; referencing them from the `persistence.xml` file does not cause them to be placed on the classpath.

All classes must be on the classpath to ensure that entity managers from different persistence units that map the same class will be accessing the same identical class.

Discoverable annotation types

A *discoverable annotation type* is an annotation type annotated with the `Discoverable` meta-annotation.

```
@Retention(CLASS)
@Target(ANNOTATION_TYPE)
public @interface Discoverable {
}
```

This specification includes a number of discoverable annotation types, including `@Entity`, `@Embeddable`, `@Converter`, `@NamedQuery`, and so on. A persistence provider may define its own custom discoverable annotation types.

Exceptionally, the annotation `jakarta.data.repository.Repository` must be treated as a discoverable annotation type, even though it is not explicitly annotated `@Discoverable`.

Annotated Classes in the Root of the Persistence Unit

By default, in the Java EE environment, the root of the persistence unit is searched for classes bearing discoverable annotation types. Each class annotated with a discoverable annotation type belongs to the persistence unit. Mapping metadata annotations found on these classes are processed; where mapping annotations are missing, the classes are mapped using mapping annotation defaults.

This behavior is disabled if the `exclude-unlisted-classes` of the `persistence.xml` file is specified as `true`. In this case, an annotated class located in the root of the persistence unit does not belong to the persistence unit unless it is explicitly listed in a `class` element of the `persistence.xml` file or in an XML mapping file.

In the Java SE environment, this behavior is not required. Portable Java SE applications should explicitly list each class belonging to the persistence unit in a `class` element of the `persistence.xml` file or in an XML mapping file. The `exclude-unlisted-classes` element is not intended for use in Java SE environments.

Object/relational Mapping Files

An object/relational mapping XML file contains mapping information for the classes it lists.

- An object/relational mapping XML file named `orm.xml` may be located in the `META-INF` directory in the root of the persistence unit or in the `META-INF` directory of any jar file referenced by the `persistence.xml`.
- Alternatively, or in addition, one or more mapping files may be referenced by the `mapping-file` elements of the `persistence-unit` element. These mapping files may be present anywhere on the class path.

An `orm.xml` mapping file or other mapping file is loaded as a resource by the persistence provider. If a mapping file is specified, the classes and mapping information listed in the mapping file are used as described in Chapter 12.

If multiple mapping files are specified (possibly including one or more `orm.xml` files), the resulting mappings are obtained by combining the mappings from all the files. If multiple mapping files referenced within a single persistence

unit (including any `orm.xml` file) contain overlapping mapping information for a given class, the result is undefined. That is, the object/relational mapping information contained in any given mapping file referenced within the persistence unit must be disjoint at the class level from object/relational mapping information contained in other mapping files referenced within the persistence unit.

Jar Files

One or more JAR files may be specified using `jar-file` elements instead of, or in addition to, the mapping files listed by the `mapping-file` elements. These JAR files are searched for classes bearing discoverable annotation types, and metadata annotations found on these classes are processed by the provider. Where mapping annotations are missing, the classes are mapped using the mapping annotation defaults defined by this specification.

Such JAR files are specified by paths relative to the directory or jar file that contains the root of the persistence unit.[2]

The following examples illustrate the use of the `jar-file` element to reference additional persistence classes. These examples make use of the convention that a jar file with a name terminating in “PUnit” contains the `persistence.xml` file and that a jar file with a name terminating in “Entities” contains additional persistence classes.

Example 1:

```
app.ear
    lib/earEntities.jar
    earRootPUnit.jar (with META-INF/persistence.xml)
```

`persistence.xml` contains:

```
<jar-file>lib/earEntities.jar</jar-file>
```

Example 2:

```
app.ear
    lib/earEntities.jar
    lib/earLibPUnit.jar (with META-INF/persistence.xml)
```

`persistence.xml` contains:

```
<jar-file>earEntities.jar</jar-file>
```

Example 3:

```
app.ear
    lib/earEntities.jar
    ejbjar.jar (with META-INF/persistence.xml)
```

`persistence.xml` contains:

```
<jar-file>lib/earEntities.jar</jar-file>
```

Example 4:

```
app.ear
    war1.war
        WEB-INF/lib/warEntities.jar
        WEB-INF/lib/warPUnit.jar (with META-INF/persistence.xml)
```

`persistence.xml` contains:

```
<jar-file>warEntities.jar</jar-file>
```

Example 5:

```
app.ear
    war2.war
        WEB-INF/lib/warEntities.jar
        WEB-INF/classes/META-INF/persistence.xml
```

`persistence.xml` contains:

```
<jar-file>lib/warEntities.jar</jar-file>
```

Example 6:

```
app.ear
    lib/earEntities.jar
    war2.war
        WEB-INF/classes/META-INF/persistence.xml
```

`persistence.xml` contains:

```
<jar-file>../../lib/earEntities.jar</jar-file>
```

Example 7:

```
app.ear
    lib/earEntities.jar
    war1.war
        WEB-INF/lib/warPUnit.jar (with META-INF/persistence.xml)
```

`persistence.xml` contains:

```
<jar-file>../../../lib/earEntities.jar</jar-file>
```

List of Managed Classes

A list of classes may be specified instead of, or in addition to, the listed JAR files and mapping files. Any mapping metadata annotations found on these classes are processed. Where mapping annotations are missing, the classes are mapped using the mapping annotation defaults. The `class` element is used to list a class.

In Java SE environments, an explicit list of all managed persistence class names must be specified to ensure portability. Portable Java SE applications should not rely on the other mechanisms described here to determine the managed persistence classes of a persistence unit. In Java SE environments, a persistence provider may require that the set of entity classes and other classes to be managed is fully enumerated in each `persistence.xml` file.

8.2.1.9. shared-cache-mode

The `shared-cache-mode` element determines whether second-level caching is in effect for the persistence unit. See Section 3.14.1.

8.2.1.10. validation-mode

The `validation-mode` element determines whether automatic lifecycle event time validation is in effect. See Section

3.11.1.2.

8.2.1.11. default-to-one-fetch-type

The `default-to-one-fetch-type` element determines the default fetch type for one-to-one and many-to-one associations, as specified in Section 11.1.

8.2.1.12. properties

The `properties` element lists settings for both standard and vendor-specific properties and hints that affect the configuration of the persistence unit and creation of its `EntityManagerFactory`.

The recognized standard properties are listed below in Section 8.2.2.

8.2.2. Persistence Unit Properties

The following standard hints are intended for use in both Jakarta EE and Java SE environments:

`jakarta.persistence.lock.timeout`

Hint specifying a preferred pessimistic lock timeout in milliseconds.

`jakarta.persistence.query.timeout`

Hint specifying a preferred query timeout in milliseconds.

The following properties are intended for use in Jakarta EE and Java SE environments when an implementation of Jakarta Validation is available:

`jakarta.persistence.validation.group.pre-persist`

Jakarta Validation groups that are targeted for validation upon the pre-persist event (overrides the default behavior).

`jakarta.persistence.validation.group.pre-merge`

Jakarta Validation groups that are targeted for validation upon the pre-merge event (overrides the default behavior).

`jakarta.persistence.validation.group.pre-remove`

Jakarta Validation groups that are targeted for validation upon the pre-remove event (overrides the default behavior).

`jakarta.persistence.validation.group.pre-insert`

Jakarta Validation groups that are targeted for validation upon the pre-insert event (overrides the default behavior).

`jakarta.persistence.validation.group.pre-update`

Jakarta Validation groups that are targeted for validation upon the pre-update event (overrides the default behavior).

`jakarta.persistence.validation.group.pre-upsert`

Jakarta Validation groups that are targeted for validation upon the pre-upsert event (overrides the default behavior).

`jakarta.persistence.validation.group.pre-delete`

Jakarta Validation groups that are targeted for validation upon the pre-delete event (overrides the default behavior).

The following standard properties are intended for use in Java SE environments.

`jakarta.persistence.jdbc.driver`

Fully qualified name of the JDBC driver class.

`jakarta.persistence.jdbc.url`

Driver-specific connection URL.

`jakarta.persistence.jdbc.user`

Username for database connection authentication.

`jakarta.persistence.jdbc.password`

Password for database connection authentication

The following properties are relevant in both SE and EE environments.

`jakarta.persistence.jdbc.fetchSize`

Override the default JDBC fetch size.

`jakarta.persistence.jdbc.batchSize`

Hint to enable JDBC statement batching by setting a batch size.

Scripts for use in schema generation, as defined in Section 9.4, or for bulk loading of data, defined in Section 9.4.2, may be specified using the following properties. These properties are intended for use in both Jakarta EE and Java SE environments. Each property specifies the name of a script packaged as part of the persistence application, or a file URL identifying a script.

`jakarta.persistence.schema-generation.create-script-source`

A DDL script which creates database objects.

`jakarta.persistence.schema-generation.drop-script-source`

A DDL script which drops database objects.

`jakarta.persistence.sql-load-script-source`

A DML script used for bulk data loading.

When scripts are packaged as part of the persistence application, these properties must specify locations relative to the root of the persistence unit. When scripts are provided externally, or when schema generation is configured to write script files, as described below, strings identifying file URLs must be specified. In a Jakarta EE environment, such file URLs must be absolute paths, and all source and target file locations must be accessible to the application server deploying the persistence unit.

In general, it is expected that schema generation will be initiated by means of the APIs described in Section 9.4. However, schema generation actions may also be controlled using the following properties in the `persistence.xml` file.

`jakarta.persistence.schema-generation.database.action`

This property specifies the schema management action to be performed by the persistence provider on database artifacts. The values for this property are `none`, `create`, `drop-and-create`, `drop`, `validate`, `populate`. If this property is not specified, the provider must not perform schema management actions on the database. (See Section 9.4.)

`jakarta.persistence.schema-generation.scripts.action`

This property specifies which scripts are to be generated by the persistence provider. Legal values for this property are `none`, `create`, `drop-and-create`, `drop`. If this property is not specified, no DDL scripts are generated. (See Section 9.4.)

`jakarta.persistence.schema-generation.create-source`

This property specifies whether the creation of database artifacts is based on the object/relational mapping metadata, a DDL script, or a combination. Legal values for this property are `metadata`, `script`, `metadata-then-script`, `script-then-metadata`. When this property is not specified:

- if a script is specified by the property `jakarta.persistence.schema-generation.create-script-source`, the script (only) is executed to create database artifacts;
- otherwise, if that property is not specified, the provider infers a schema from the object/relational mapping metadata (only) and creates the artifacts of the inferred schema.

The values `metadata-then-script` and `script-then-metadata` specify that a combination of metadata and script is used and the order in which this use occurs. If either of these values is specified and the resulting database actions are not disjoint, the results are undefined and the creation of database artefacts might fail.

`jakarta.persistence.schema-generation.drop-source`

This property specifies whether the dropping of database artifacts is based on the object/relational mapping metadata, a DDL script, or a combination. Legal values for this property are `metadata`, `script`, `metadata-then-script`, `script-then-metadata`. When this property is not specified:

- if a script is specified by the property `jakarta.persistence.schema-generation.drop-script-source`, the script (only) is executed to drop database artifacts;
- otherwise, if that property is not specified, the provider infers a schema from the object/relational mapping metadata (only) and drops the artifacts of the inferred schema.

The values `metadata-then-script` and `script-then-metadata` specify that a combination of metadata and script is to be used and the order in which this use occurs. If either of these values is specified and the resulting database actions are not disjoint, the results are undefined and the dropping of database artifacts may fail.

`jakarta.persistence.schema-generation.create-database-schemas`

Specifies whether the persistence provider creates database schemas in addition to creating database objects such as tables, sequences, constraints, and so on. The value `true` specifies that the persistence provider creates schemas in the database or generates DDL containing `CREATE SCHEMA` commands. If this property is not supplied, the provider does not create database schemas.

`jakarta.persistence.schema-generation.scripts.create-target`,

`jakarta.persistence.schema-generation.scripts.drop-target`

If scripts are generated, these properties must specify the target locations where the scripts are written. Each property specifies a string designating the file URL of the DDL script. The properties should only be specified if scripts are generated.

Ordinarily, the persistence provider automatically determines the database in use by querying the JDBC database metadata or other vendor-specific heuristics. In some environments, JDBC metadata is unavailable, and information about the database must be explicitly specified.

jakarta.persistence.database-product-name,

jakarta.persistence.database-major-version,

jakarta.persistence.database-minor-version

In situations where JDBC metadata is not available, the property `jakarta.persistence.database-product-name` must be specified. The value of this property should be the value returned for the target database by the method `getDatabaseProductName` of the JDBC `DatabaseMetaData` object. If sufficient database version information is not included in the result of this method, the `jakarta.persistence.database-major-version` and `jakarta.persistence.database-minor-version` properties should be specified as needed. These should specify the values returned by the `DatabaseMetaData` methods `getDatabaseMajorVersion` and `getDatabaseMinorVersion` respectively.

If a persistence provider does not recognize a property (other than a property defined by this specification), the provider must ignore it.

Vendors should define properties in vendor-specific namespaces, (e.g `com.acme.persistence.logging`). The namespace `jakarta.persistence` is reserved for use by this specification, and must not be used to define vendor-specific properties.

8.2.3. Example Persistence Unit Configurations

The following are sample contents of a `persistence.xml` file.

Example 1:

```
<persistence-unit name="OrderManagement"/>
```

A persistence unit named `OrderManagement` is created.

Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. If a `META-INF/orm.xml` file exists, any classes referenced by it and mapping information contained in it are used as specified above. Because no provider is specified, the persistence unit is assumed to be portable across providers. Because the transaction type is not specified, `JTA` is assumed for Jakarta EE environments. The container must provide the data source (it may be specified at application deployment, for example). In Java SE environments, the data source may be specified by other means and a transaction type of `RESOURCE_LOCAL` is assumed.

Example 2:

```
<persistence-unit name="OrderManagement2">
    <mapping-file>mappings.xml</mapping-file>
</persistence-unit>
```

A persistence unit named `OrderManagement2` is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. The `mappings.xml` resource exists on the classpath and any classes and mapping information contained in it are used as specified above. If a `META-INF/orm.xml` file exists, any classes and mapping information contained in it are used as well. The transaction type, data source, and provider are as described above.

Example 3:

```
<persistence-unit name="OrderManagement3">
    <jar-file>order.jar</jar-file>
    <jar-file>order-supplemental.jar</jar-file>
</persistence-unit>
```

A persistence unit named `OrderManagement3` is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed persistence classes. If a `META-INF/orm.xml` file exists, any classes and

mapping information contained in it are used as specified above. The `order.jar` and `order-supplemental.jar` files are searched for managed persistence classes and any annotated managed persistence classes found in them and/or any classes specified in the `orm.xml` files of these jar files are added. The transaction-type, data source and provider are as described above.

Example 4:

```
<persistence-unit name="OrderManagement4" transaction-type="RESOURCE_LOCAL">
    <non-jta-data-source>java:app/jdbc/MyDB</non-jta-data-source>
    <mapping-file>order-mappings.xml</mapping-file>
    <class>com.acme.Order</class>
    <class>com.acme.Customer</class>
    <class>com.acme.Item</class>
    <exclude-unlisted-classes/>
</persistence-unit>
```

A persistence unit named `OrderManagement4` is created. The file `order-mappings.xml` is read as a resource and any classes referenced by it and mapping information contained in it are used[3]. The annotated `Order`, `Customer` and `Item` classes are loaded and are added. No (other) classes contained in the root of the persistence unit are added to the list of managed persistence classes. The persistence unit assumed to be portable across providers. An entity manager factory supplying resource-local entity managers will be created. The data source `java:app/jdbc/MyDB` must be used.

Example 5:

```
<persistence-unit name="OrderManagement5">
    <provider>com.acme.AcmePersistence</provider>
    <mapping-file>order1.xml</mapping-file>
    <mapping-file>order2.xml</mapping-file>
    <jar-file>order.jar</jar-file>
    <jar-file>order-supplemental.jar</jar-file>
</persistence-unit>
```

A persistence unit named `OrderManagement5` is created. Any annotated managed persistence classes found in the root of the persistence unit are added to the list of managed classes. The `order1.xml` and `order2.xml` files are read as resources and any classes referenced by them and mapping information contained in them are also used as specified above. The `order.jar` is a jar file on the classpath containing another persistence unit, while `order-supplemental.jar` is just a library of classes. Both of these jar files are searched for annotated managed persistence classes and any annotated managed persistence classes found in them and any classes specified in the `orm.xml` files (if any) of these jar files are added. The provider `com.acme.AcmePersistence` must be used.

> Note that the `persistence.xml` file contained in `order.jar` is not used to augment the persistence unit `OrderManagement5` with the classes of the persistence unit whose root is `order.jar`.

8.3. Persistence Unit Scope

An EJB-JAR, WAR, application client JAR, or EAR can define a persistence unit. When referencing a persistence unit using the `unitName` annotation element or `persistence-unit-name` deployment descriptor element, the visibility scope of the persistence unit is determined by its point of definition:

- A persistence unit defined at the level of an EJB-JAR, WAR, or application client JAR is scoped to that EJB-JAR, WAR, or application JAR, respectively, and is visible to the components defined in that jar or WAR.
- A persistence unit defined at the level of an EAR is generally visible to all components in the application.
- It is an error for a persistence unit defined at the level of an EAR to have the same name as a persistence unit defined by an EJB-JAR, WAR, or application JAR file contained within the EAR.[4]

The # syntax may be used with both the `unitName` annotation element or `persistence-unit-name` deployment descriptor element to reference a persistence unit defined at EAR level.

When the # syntax is used, the path name is interpreted relative to the referencing application component jar file. For example, the syntax `../lib/persistenceUnitRoot.jar#myPersistenceUnit` refers to a persistence unit with:

- name `myPersistenceUnit`, as specified in the `name` element of the `persistence.xml` file, and
- root given by the relative path name `../lib/persistenceUnitRoot.jar`.

8.4. persistence.xml Schema

This section provides the XML schema for the `persistence.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- persistence.xml schema -->
<xsd:schema targetNamespace="https://jakarta.ee/xml/ns/persistence"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:persistence="https://jakarta.ee/xml/ns/persistence"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="4.0">

   <xsd:annotation>
     <xsd:documentation><![CDATA[

     This is the XML Schema for the persistence configuration file.
     The file must be named "META-INF/persistence.xml" in the
     persistence archive.

     Persistence configuration files must indicate
     the persistence schema by using the persistence namespace:

     https://jakarta.ee/xml/ns/persistence

     and indicate the version of the schema by
     using the version element as shown below:

      <persistence xmlns="https://jakarta.ee/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
          https://jakarta.ee/xml/ns/persistence/persistence_4_0.xsd"
        version="4.0">
          ...
      </persistence>

    ]]></xsd:documentation>
  </xsd:annotation>

  <xsd:simpleType name="versionType">
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-9]+(\.[0-9]+)*"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- ************************************************** -->

  <xsd:element name="persistence">
    <xsd:complexType>
      <xsd:sequence>

        <!-- ************************************************** -->

        <xsd:element name="persistence-unit"
```

```
          minOccurs="1" maxOccurs="unbounded">
<xsd:complexType>
  <xsd:annotation>
    <xsd:documentation>

      Configuration of a persistence unit.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>

  <!-- ************************************************** -->

    <xsd:element name="description" type="xsd:string"
                 minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>

          Description of this persistence unit.

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>

    <!-- ************************************************** -->

    <xsd:element name="provider" type="xsd:string"
                 minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>

          Provider class that supplies EntityManagers for this
          persistence unit.

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>

    <!-- ************************************************** -->

    <xsd:element name="qualifier" type="xsd:string"
                 minOccurs="0" maxOccurs="unbounded">
      <xsd:annotation>
        <xsd:documentation>

          Qualifier annotation class used for dependency injection.

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>

    <!-- ************************************************** -->

    <xsd:element name="scope" type="xsd:string"
                 minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>

          Scope annotation class used for dependency injection.

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>

    <!-- ************************************************** -->

    <xsd:element name="jta-data-source" type="xsd:string"
```

```
            minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      The container-specific name of the JTA datasource to use.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- ******************************************************* -->

<xsd:element name="non-jta-data-source" type="xsd:string"
            minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      The container-specific name of a non-JTA datasource to use.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- ******************************************************* -->

<xsd:element name="mapping-file" type="xsd:string"
            minOccurs="0" maxOccurs="unbounded">
  <xsd:annotation>
    <xsd:documentation>

      File containing mapping information. Loaded as a resource
      by the persistence provider.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- ******************************************************* -->

<xsd:element name="jar-file" type="xsd:string"
            minOccurs="0" maxOccurs="unbounded">
  <xsd:annotation>
    <xsd:documentation>

      Jar file that is to be scanned for managed classes.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- ******************************************************* -->

<xsd:element name="class" type="xsd:string"
            minOccurs="0" maxOccurs="unbounded">
  <xsd:annotation>
    <xsd:documentation>

      Managed class to be included in the persistence unit and
      to scan for annotations.  It should be annotated
      with either @Entity, @Embeddable or @MappedSuperclass.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- ******************************************************* -->
```

```
<xsd:element name="exclude-unlisted-classes" type="xsd:boolean"
             default="true" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      When set to true then only listed classes and jars will
      be scanned for persistent classes, otherwise the
      enclosing jar or directory will also be scanned.
      Not applicable to Java SE persistence units.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>


<!-- ***************************************************** -->

<xsd:element name="default-to-one-fetch-type"
             type="persistence:persistence-unit-default-to-one-fetch-type"
             minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      The default fetch type for one-to-one and many-to-one associations.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>


<!-- ***************************************************** -->

<xsd:element name="shared-cache-mode"
             type="persistence:persistence-unit-caching-type"
             minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      Defines whether caching is enabled for the
      persistence unit if caching is supported by the
      persistence provider. When set to ALL, all entities
      will be cached. When set to NONE, no entities will
      be cached. When set to ENABLE_SELECTIVE, only entities
      specified as cacheable will be cached. When set to
      DISABLE_SELECTIVE, entities specified as not cacheable
      will not be cached. When not specified or when set to
      UNSPECIFIED, provider defaults may apply.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- ***************************************************** -->

<xsd:element name="validation-mode"
             type="persistence:persistence-unit-validation-mode-type"
             minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      The validation mode to be used for the persistence unit.

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
```

```
  <!-- ************************************************** -->

  <xsd:element name="properties" minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>

        A list of standard and vendor-specific properties
        and hints.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="property"
                     minOccurs="0" maxOccurs="unbounded">
          <xsd:annotation>
            <xsd:documentation>
              A name-value pair.
            </xsd:documentation>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:attribute name="name" type="xsd:string"
                           use="required"/>
            <xsd:attribute name="value" type="xsd:string"
                           use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:any namespace="##other" processContents="lax"
           minOccurs="0" maxOccurs="unbounded">
    <xsd:annotation>
      <xsd:documentation>
        An extension point for integration related configuration, e.g. cdi:
        <!--
        <persistence-unit name="my-unit" xmlns:cdi="https://jakarta.ee/xml/ns/persistence-cdi">
          ...
          <cdi:scope>com.example.jpa.ACustomScope</cdi:scope>
          <cdi:qualifier>com.example.jpa.CustomQualifier</cdi:qualifier>
        </persistence-unit>
        -->
      </xsd:documentation>
    </xsd:annotation>
  </xsd:any>
</xsd:sequence>

<!-- ************************************************** -->

<xsd:attribute name="name" type="xsd:string" use="required">
  <xsd:annotation>
    <xsd:documentation>

      Name used in code to reference this persistence unit.

    </xsd:documentation>
  </xsd:annotation>
</xsd:attribute>

<!-- ************************************************** -->

<xsd:attribute name="transaction-type"
               type="persistence:persistence-unit-transaction-type">
  <xsd:annotation>
    <xsd:documentation>
```

```
                Type of transactions used by EntityManagers from this
                persistence unit.

              </xsd:documentation>
            </xsd:annotation>
          </xsd:attribute>

        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="version" type="persistence:versionType"
                   fixed="4.0" use="required"/>
  </xsd:complexType>
</xsd:element>

<!-- ***************************************************** -->

<xsd:simpleType name="persistence-unit-transaction-type">
  <xsd:annotation>
    <xsd:documentation>

      public enum PersistenceUnitTransactionType {JTA, RESOURCE_LOCAL};

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="JTA"/>
    <xsd:enumeration value="RESOURCE_LOCAL"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- ***************************************************** -->

<xsd:simpleType name="persistence-unit-default-to-one-fetch-type">
  <xsd:annotation>
    <xsd:documentation>

      public enum FetchType { LAZY, EAGER, DEFAULT };

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="LAZY"/>
    <xsd:enumeration value="EAGER"/>
    <!-- no DEFAULT -->
  </xsd:restriction>
</xsd:simpleType>

<!-- ***************************************************** -->

<xsd:simpleType name="persistence-unit-caching-type">
  <xsd:annotation>
    <xsd:documentation>

      public enum SharedCacheMode { ALL, NONE, ENABLE_SELECTIVE, DISABLE_SELECTIVE, UNSPECIFIED};

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="ALL"/>
    <xsd:enumeration value="NONE"/>
    <xsd:enumeration value="ENABLE_SELECTIVE"/>
    <xsd:enumeration value="DISABLE_SELECTIVE"/>
    <xsd:enumeration value="UNSPECIFIED"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<!-- ************************************************** -->

  <xsd:simpleType name="persistence-unit-validation-mode-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum ValidationMode { AUTO, CALLBACK, NONE };

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="AUTO"/>
      <xsd:enumeration value="CALLBACK"/>
      <xsd:enumeration value="NONE"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

[1] The root of the persistence unit is the `WEB-INF/classes` directory; the `persistence.xml` file is therefore contained in the `WEB-INF/classes/META-INF` directory.

[2] Persistence providers are encouraged to support this syntax in Java SE environments.

[3] Note that in this example a META-INF/orm.xml file is assumed not to exist.

[4] This was permitted prior to 4.0.

Chapter 9. Container and Provider Contracts for Deployment and Bootstrapping

This chapter defines requirements on the Jakarta EE container and on the persistence provider for deployment and bootstrapping.

9.1. Jakarta EE Deployment

Each persistence unit deployed into a Jakarta EE container consists of a single `persistence.xml` file, any number of mapping files, and any number of class files.

At deployment time the container is responsible for scanning the locations specified in Section 8.1 and discovering the `persistence.xml` files and processing them.

When the container finds a `persistence.xml` file, it must process the persistence unit definitions that it contains. The container must validate the `persistence.xml` file against the `persistence_4_0.xsd`, `persistence_3_2.xsd`, `persistence_3_0.xsd` or `persistence_2_2.xsd` schema in accordance with the version specified by the `persistence.xml` file and report any validation errors. Provider or data source information not specified in the `persistence.xml` file must be provided at deployment time or defaulted by the container. The container may optionally add any container-specific properties to be passed to the provider when creating the entity manager factory for the persistence unit.

Once the container has read the persistence metadata, it determines the `jakarta.persistence.spi.PersistenceProvider` implementation class for each deployed named persistence unit. The container then creates an instance of the `PersistenceProvider` implementation class for each deployed named persistence unit and invokes the `createContainerEntityManagerFactory` method on that instance.

- The container must implement the `PersistenceUnitInfo` interface described in Section 9.6 and pass the metadata—in the form of a `PersistenceUnitInfo` instance—to the persistence provider as part of this call.
- If a Bean Validation provider exists in the container environment and the `validation-mode NONE` is not specified, a `ValidatorFactory` instance must be made available by the container. The container is responsible for passing this `ValidatorFactory` instance via the map that is passed as an argument to the `createContainerEntityManagerFactory` call. The map key used must be the standard property name `jakarta.persistence.validation.factory`.
- If CDI is enabled, a `BeanManager` instance must be made available by the container. The container is responsible for passing this `BeanManager` instance via the map that is passed as an argument to the `createContainerEntityManagerFactory` call. The map key used must be the standard property name `jakarta.persistence.bean.manager`.

The `EntityManagerFactory` instance obtained as a result is used by the container to create container-managed entity managers. Only one `EntityManagerFactory` is permitted to be created for each deployed persistence unit configuration. Any number of `EntityManager` instances may be created from a given factory.

In a Jakarta EE environment, the classes of the persistence unit should not be loaded by the application class loader or any of its parent class loaders until after the entity manager factory for the persistence unit has been created.

When a persistence unit is redeployed, the container should call the `close` method on the previous `EntityManagerFactory` instance and call the `createContainerEntityManagerFactory` method again, with the required `PersistenceUnitInfo` metadata, to achieve the redeployment.

9.1.1. Integration with JNDI

The container must make the `EntityManager`, `EntityAgent`, and `EntityManagerFactory` available for lookup in JNDI and injection via the `@PersistenceContext` and `@PersistenceUnit` annotations. The `name` of the persistence unit, as specified in Section 8.2.1.1, determines the JNDI addresses of these objects. For a persistence unit with name `unitName` located in a WAR, EJB-JAR, or application client JAR:

- the `EntityManager` is bound to the address `java:module/persistence/unitName/EntityManager`,
- the `EntityAgent` is bound to the address `java:module/persistence/unitName/EntityAgent`, and
- the `EntityManagerFactory` is bound to the address `java:module/persistence/unitName/EntityManagerFactory`.

For a persistence unit with name `unitName` located directly in an EAR:

- the `EntityManager` is bound to the address `java:app/persistence/unitName/EntityManager`,
- the `EntityAgent` is bound to the address `java:app/persistence/unitName/EntityAgent`, and
- the `EntityManagerFactory` is bound to the address `java:app/persistence/unitName/EntityManagerFactory`.

The `name` member of the `@PersistenceContext` or `@PersistenceUnit` annotation or the corresponding element of the XML deployment descriptors may be used to assign a name to in the `java:comp/env` JNDI namespace.

The application may obtain a reference to these objects via injection or JNDI lookup, as specified in Section 7.2.1 and Section 7.3.1.

9.2. Bootstrapping in Java SE Environments

In Java SE environments, the `Persistence.createEntityManagerFactory` method is used by the application to create an entity manager factory[1].

A persistence provider implementation running in a Java SE environment should also act as a service provider by supplying a service provider configuration file as defined by the Java SE platform.

The provider configuration file serves to export the provider implementation class to the `Persistence` bootstrap class, positioning the provider as a candidate for backing named persistence units. The provider supplies the provider configuration file by creating a text file named `jakarta.persistence.spi.PersistenceProvider` and placing it in the `META-INF/services` directory of one of its JAR files. The contents of the file should be the name of the provider implementation class of the `jakarta.persistence.spi.PersistenceProvider` interface.

Example:

A persistence vendor called ACME persistence products ships a JAR called `acme.jar` that contains its persistence provider implementation. The JAR includes the provider configuration file.

```
acme.jar
    META-INF/services/jakarta.persistence.spi.PersistenceProvider
    com.acme.PersistenceProvider
    ...
```

The contents of the `META-INF/services/jakarta.persistence.spi.PersistenceProvider` file is nothing more than the name of the implementation class: `com.acme.PersistenceProvider`.

Persistence provider jars may be installed or made available in the same ways as other service providers, e.g. as extensions or added to the application classpath.

The `Persistence` bootstrap class must locate all of the persistence providers using the `PersistenceProviderResolver` mechanism described in Section 9.3 and call `createEntityManagerFactory` on them in turn until an appropriate backing provider returns an `EntityManagerFactory` instance. A provider may deem itself as appropriate for the persistence unit if any of the following are true:

- Its implementation class has been specified in the `provider` element for that persistence unit in the `persistence.xml` file and has not been overridden by a different `jakarta.persistence.provider` property value included in the Map passed to the `createEntityManagerFactory` method.
- The `jakarta.persistence.provider` property was included in the Map passed to `createEntityManagerFactory` and the

value of the property is the provider's implementation class.
- No provider was specified for the persistence unit in either the `persistence.xml` or the property map.

If a provider does not qualify as the provider for the named persistence unit, it must return `null` when `createEntityManagerFactory` is invoked on it.

9.2.1. Schema Generation

In Java SE environments, the `Persistence.generateSchema` method may be used by the application to cause schema generation to occur as a separate phase from entity manager factory creation.

In this case, the `Persistence` bootstrap class must locate all of the persistence providers using the `PersistenceProviderResolver` mechanism described in Section 9.3 and call `generateSchema` on them in turn until an appropriate backing provider returns `true`. A provider may deem itself as appropriate for the persistence unit if any of the following are true:

- Its implementation class has been specified in the `provider` element for that persistence unit in the `persistence.xml` file and has not been overridden by a different `jakarta.persistence.provider` property value included in the Map passed to the `generateSchema` method.
- The `jakarta.persistence.provider` property was included in the Map passed to `generateSchema` and the value of the property is the provider's implementation class.
- No provider was specified for the persistence unit in either the `persistence.xml` or the property map.

If a provider does not qualify as the provider for the named persistence unit, it must return `false` when `generateSchema` is invoked on it.

9.3. Determining the Available Persistence Providers

The `PersistenceProviderResolver` and `PersistenceProviderResolverHolder` mechanism supports the dynamic discovery of persistence providers.[2]

- The `PersistenceProviderResolver` instance is responsible for returning the list of providers available in the environment.
- The `PersistenceProviderResolverHolder` class holds the `PersistenceProviderResolver` instance that is in use.

These interfaces may be found in Appendix E.

The implementation of `PersistenceProviderResolverHolder` must be threadsafe, but no guarantee is made against multiple threads setting the resolver.

The container is allowed to implement and set a specific `PersistenceProviderResolver` provided that it respects the `PersistenceProviderResolver` contract. The `PersistenceProviderResolver` instance to be used is set by the container using the `PersistenceProviderResolverHolder.setPersistenceProviderResolver` method.[3]

If no `PersistenceProviderResolver` is set, the `PersistenceProviderResolverHolder` must return a `PersistenceProviderResolver` that returns the providers whose persistence provider jars have been installed or made available as service providers or extensions. This default `PersistenceProviderResolver` instance does not guarantee the order in which persistence providers are returned.

A `PersistenceProviderResolver` must be threadsafe.

The `PersistenceProviderResolver.getPersistenceProviders()` method must be used to determine the list of available persistence providers.

The results of calling the `PersistenceProviderResolverHolder.getPersistenceProviderResolver` and the `PersistenceProviderResolver.getPersistenceProviders` methods must not be cached. In particular, the following methods must use the `PersistenceProviderResolver` instance returned by the `PersistenceProviderResolverHolder.getPersistenceProviderResolver` method to determine the list of available providers:

- `Persistence.createEntityManagerFactory(String)`
- `Persistence.createEntityManagerFactory(String, Map)`
- `PersistenceUtil.isLoaded(Object)`
- `PersistenceUtil.isLoaded(Object, String)`

These methods must not cache the list of providers and must not cache the `PersistenceProviderResolver` instance.

> Note that the `PersistenceProviderResolver.getPersistenceProviders()` method can potentially be called many times. It is therefore recommended that the implementation of this method make use of caching.

Note that only a single `PersistenceProviderResolver` instance can be defined in a given classloader hierarchy at a given time.

9.4. Schema Generation

In cases where a preconfigured database (or “legacy” database) is not in use or is not available, the Jakarta Persistence schema generation facility may be used to generate the tables and other database artifacts required by the persistence application. Whether schema generation entails the creation of schemas proper in the database is determined by the environment and by the configuration of the schema generation process, as described below.

Schema generation may happen either prior to application deployment or when the `EntityManagerFactory` is created as part of the application deployment and initialization process.

- In Jakarta EE environments, the container may call the static method `PersistenceProvider.generateSchema()` separately from and/or prior to the creation of the entity manager factory for a persistence unit, or the container may pass additional information to the `createContainerEntityManagerFactory()` call that causes schema generation to happen as part of the entity manager factory creation and application initialization process.
- In Java SE environments, the application may call the static method `Persistence.generateSchema()` method separately from and/or prior to the creation of the entity manager factory or may pass information to the `createEntityManagerFactory()` method to cause schema generation to occur as part of the entity manager factory creation.

The information passed to these methods in the form of property settings controls whether:

- generation occurs directly in the target database,
- DDL scripts for schema generation are created as files,
- or both.

9.4.1. Properties for Schema Generation

The application may provide DDL scripts to be used for schema generation, as described in Section 8.2.1.12. The application developer may package these scripts as part of the persistence unit or may specify file URLs identifying the location of such scripts as strings.

- In Jakarta EE environments, such scripts may be executed by the container, or the container may direct the persistence provider to execute the scripts.
- In Java SE environments, execution of the scripts is the responsibility of the persistence provider.

In the absence of the specification of scripts, schema generation, if requested, is determined by the object/relational metadata of the persistence unit.

The following standard properties are defined for configuring the schema generation process.

- In Jakarta EE, these properties are passed by the container in the `Map` argument of either the `generateSchema()` method or the `createContainerEntityManagerFactory()` method of `PersistenceProvider`.
- In Java SE, they are passed in the `Map` argument of either the `generateSchema()` method or the `createEntityManagerFactory` method of `Persistence`.

In Jakarta EE environments:

- Every string representing the file URL of a script source or target must specify an absolute path. If a file URL is specified as a relative path, the behavior is undefined.
- Every source file and target file location must be accessible to the application server deploying the persistence unit.

> This specification does not define the behavior if a target location for a script is already occupied by an existing file. An implementation might overwrite the file, append to the file, or fail fast. Portable applications must delete or truncate any file existing at a script target location.

`jakarta.persistence.schema-generation.database.action`

This property specifies the schema management action to be performed by the persistence provider on database artifacts. The values for this property are `none`, `create`, `drop-and-create`, `drop`, `validate`, `populate`. If this property is not specified, the provider must not perform schema management actions on the database.

`jakarta.persistence.schema-generation.scripts.action`

This property specifies which scripts are to be generated by the persistence provider. Legal values for this property are `none`, `create`, `drop-and-create`, `drop`. If this property is not specified, no DDL scripts are generated.

`jakarta.persistence.schema-generation.create-source`

This property specifies whether the creation of database artifacts is based on the object/relational mapping metadata, a DDL script, or a combination. Legal values for this property are `metadata`, `script`, `metadata-then-script`, `script-then-metadata`. When this property is not specified:

- if a script is specified by the property `jakarta.persistence.schema-generation.create-script-source`, the script (only) is executed to create database artifacts;
- otherwise, if that property is not specified, the provider infers a schema from the object/relational mapping metadata (only) and creates the artifacts of the inferred schema.

The values `metadata-then-script` and `script-then-metadata` specify that a combination of metadata and script is used and the order in which this use occurs. If either of these values is specified and the resulting database actions are not disjoint, the results are undefined and the creation of database artefacts might fail.

`jakarta.persistence.schema-generation.drop-source`

This property specifies whether the dropping of database artifacts is based on the object/relational mapping metadata, a DDL script, or a combination. Legal values for this property are `metadata`, `script`, `metadata-then-script`, `script-then-metadata`. When this property is not specified:

- if a script is specified by the property `jakarta.persistence.schema-generation.drop-script-source`, the script (only) is executed to drop database artifacts;
- otherwise, if that property is not specified, the provider infers a schema from the object/relational mapping metadata (only) and drops the artifacts of the inferred schema.

The values `metadata-then-script` and `script-then-metadata` specify that a combination of metadata and script is to

be used and the order in which this use occurs. If either of these values is specified and the resulting database actions are not disjoint, the results are undefined and the dropping of database artifacts may fail.

`jakarta.persistence.schema-generation.create-database-schemas`

This property specifies whether the persistence provider creates database schemas in addition to creating database objects such as tables, sequences, constraints, and so on. The value `true` specifies that the persistence provider creates schemas in the database or generates DDL containing `CREATE SCHEMA` commands. If this property is not supplied, the provider must not create database schemas.[4]

`jakarta.persistence.schema-generation.scripts.create-target`,

`jakarta.persistence.schema-generation.scripts.drop-target`

If scripts are generated, these properties must specify the target locations where the scripts are written. Each property specifies either a `java.io.Writer` configured for output of the DDL script or a string designating the file URL of the DDL script. The properties should only be specified if scripts are generated.

`jakarta.persistence.database-product-name`,

`jakarta.persistence.database-major-version`,

`jakarta.persistence.database-minor-version`

If scripts are generated by the persistence provider, and a connection to the target database is not supplied, the `jakarta.persistence.database-product-name` property must be specified. The value of this property should be the value returned for the target database by the method `getDatabaseProductName` of the JDBC `DatabaseMetaData` object. If sufficient database version information is not included in the result of this method, the `jakarta.persistence.database-major-version` and `jakarta.persistence.database-minor-version` properties should be specified as needed. These should specify the values returned by the `DatabaseMetaData` methods `getDatabaseMajorVersion` and `getDatabaseMinorVersion` respectively.

`jakarta.persistence.schema-generation.create-script-source`,

`jakarta.persistence.schema-generation.drop-script-source`

These properties specify executable DDL scripts.[5] Each property specifies either a `java.io.Reader` configured for reading the DDL script or a string designating a file URL of the DDL script.

`jakarta.persistence.schema-generation.connection`

This property supplies a JDBC connection used for executing schema management actions on the database.[6] If this property is not specified, the persistence provider uses the `DataSource` configured for use by the persistence unit to obtain a JDBC connection.

9.4.2. Data Loading

Data loading via execution of a SQL DML script may occur as part of the schema generation process, after the creation of the database artifacts, or independently of schema creation. The specification of the property `jakarta.persistence.sql-load-script-source` supplies the location of the DML script which loads data, and helps control whether data loading occurs.

`jakarta.persistence.sql-load-script-source`

This property specifies either a `java.io.Reader` configured for reading a SQL DML script for data loading or a string designating a file URL where the script is located.

When this property is set, the behavior depends on the schema generation action specified by the property `jakarta.persistence.schema-generation.database.action`:

- if the action is `create` or `drop-and-create`, the persistence provider must execute the DML script after creating the database artifacts,
- if the action is `populate`, the persistence provider must execute the DML script,
- if the action is `drop`, `validate`, or `none`, the persistence provider must not execute the DML script.

If `jakarta.persistence.sql-load-script-source` is set, but the property `jakarta.persistence.schema-generation.database.action` is not set, the behavior is undefined and is not portable between persistence providers.

9.5. Responsibilities of the Persistence Provider

The persistence provider must implement the `PersistenceProvider` SPI.

In Jakarta EE environments, the persistence provider must process the metadata that is passed to it at the time `createContainerEntityManagerFactory` method is called and create an instance of `EntityManagerFactory` using the `PersistenceUnitInfo` metadata for the factory. The factory is then returned to the container.

In Java SE environments, the persistence provider must validate the `persistence.xml` file against the `persistence` schema that corresponds to the version specified by the `persistence.xml` file and report any validation errors.

The persistence provider processes the metadata annotations on the managed classes of the persistence unit.

When the entity manager factory for a persistence unit is created, it is the responsibility of the persistence provider to initialize the state of the metamodel classes of the persistence unit.

When the persistence provider obtains an object/relational mapping file, it processes the definitions that it contains. The persistence provider must validate any object/relational mapping files against the object/relational mapping schema version specified by the object/relational mapping file and report any validation errors. The object relational mapping file must specify the object/relational mapping schema that it is written against by indicating the `version` element.

In Java SE environments, the application can pass the `ValidatorFactory` instance via the map that is passed as an argument to the `Persistence.createEntityManagerFactory` call. The map key used must be the standard property name `jakarta.persistence.validation.factory`. If no `ValidatorFactory` instance is provided by the application, and if a Bean Validation provider is present in the classpath, the persistence provider must instantiate the `ValidatorFactory` using the default bootstrapping approach as defined by the Bean Validation specification [5], namely `Validation.buildDefaultValidatorFactory()`.

9.5.1. jakarta.persistence.spi.PersistenceProvider

The `PersistenceProvider` interface found in Section E.3 must be implemented by the persistence provider.

The `PersistenceProvider` implementation class must have a public constructor with no parameters.

An instance of `PersistenceProvider` is responsible for creating provider-specific implementations of `EntityManagerFactory`. It is invoked by the container in Jakarta EE environments and by the `jakarta.persistence.Persistence` class in Java SE environments. The `jakarta.persistence.spi.PersistenceProvider` implementation is not intended to be used by the application.

The properties passed to the `createEntityManagerFactory()` method in Java SE environments are described further in Section 9.7 below.

9.5.2. jakarta.persistence.spi.ProviderUtil

The `ProviderUtil` interface found in Section E.7 is called by the `PersistenceUtil` implementation to determine the load status of an entity or entity attribute. It is not intended to be invoked by the application.

9.6. jakarta.persistence.spi.PersistenceUnitInfo Interface

The `PersistenceUnitInfo` interface may be found in Section E.6.

The enum `PersistenceUnitTransactionType` lists options which determine whether entity managers created by the factory are JTA or resource-local entity managers.

```
 * Enumerates the possible approaches to transaction management in Jakarta
 * Persistence. The approach to transaction management is determined by the
 * {@linkplain PersistenceConfiguration#transactionType() configuration of
 * a persistence unit}, and is common across all entity managers and entity
 * agents for the given persistence unit. An {@link EntityHandler} may be
 * either:
 * <ul>
 * <li>a {@linkplain #JTA} entity manager or entity agent, where transaction
 *     management is done via JTA (Jakarta Transactions), or
 * <li>a {@linkplain #RESOURCE_LOCAL resource-local} entity manager or entity
 *     agent, where transaction management is a responsibility of the
 *     persistence provider and the transaction lifecycle is controlled via
 *     the {@link EntityTransaction} interface.
 * </ul>
 * <p>
 * A {@code PersistenceUnitTransactionType} may be specified by:
 * <ul>
 * <li>the {@code transaction-type} attribute of {@code <persistence-unit>}
 *     in {@code persistence.xml},
 * <li>the property
 *     {@value Persistence.UnitProperties#PERSISTENCE_UNIT_TRANSACTION_TYPE},
 *     or
 * <li>{@link PersistenceConfiguration#transactionType(PersistenceUnitTransactionType)}.
 * </ul>
 *
 * @see jakarta.persistence.spi.PersistenceUnitInfo#getTransactionType()
 * @see PersistenceConfiguration#transactionType()
 * @see Persistence.UnitProperties#PERSISTENCE_UNIT_TRANSACTION_TYPE
 *
 * @since 3.2
 */
public enum PersistenceUnitTransactionType {
    /**
     * Transaction management via JTA (Jakarta Transactions).
     */
    JTA,
    /**
     * Resource-local transaction management.
     */
    RESOURCE_LOCAL
}
```

The enum `SharedCacheMode` lists policies which control how the provider determines if an entity type is eligible for storage in the second-level cache, as defined in Section 3.14.1. The `shared-cache-mode` element in `persistence.xml` has no default value. The `getSharedCacheMode` method must return `UNSPECIFIED` when the `shared-cache-mode` element is left unspecified.

```
import jakarta.persistence.spi.PersistenceUnitInfo;

/**
```

```
 * Enumerates policies which control how the provider determines if
 * an entity type is eligible for storage in the second-level cache.
 * The shared cache mode is determined by the
 * {@linkplain PersistenceConfiguration#sharedCacheMode() configuration
 * of the persistence unit}, and is common across all entity managers
 * and entity agents for the given persistence unit.
 * <p>
 * A {@code SharedCacheMode} may be specified by:
 * <ul>
 * <li>the {@code <shared-cache-mode>} element of {@code persistence.xml},
 * <li>the property {@value Persistence.CacheProperties#CACHE_MODE}, or
 * <li>{@link PersistenceConfiguration#sharedCacheMode(SharedCacheMode)}.
 * </ul>
 *
 * @see PersistenceUnitInfo#getSharedCacheMode()
 * @see PersistenceConfiguration#sharedCacheMode()
 * @see Persistence.CacheProperties#CACHE_MODE
 *
 * @since 2.0
 */
public enum SharedCacheMode {

    /**
     * Every entity type associated with the persistence unit and all
     * entity-related state and data is eligible for storage in the
     * second-level cache.
     */
    ALL,

    /**
     * The second-level cache is disabled for the whole persistence unit.
     * No entity type is eligible for storage in the second-level cache.
     */
    NONE,

    /**
     * An entity type is eligible for storage in the second-level cache
     * if and only if it is annotated {@link Cacheable Cacheable} or
     * {@link Cacheable Cacheable(true)}.
     */
    ENABLE_SELECTIVE,

    /**
     * An entity type is eligible for storage in the second-level cache
     * if and only if it is not annotated {@link Cacheable Cacheable(false)}.
     */
    DISABLE_SELECTIVE,

    /**
     * The persistence provider determines if an entity type is eligible
     * for storage in the second-level cache, according to vendor-specific
     * semantics.
     */
    UNSPECIFIED
}
```

The enum `ValidationMode` lists the validation modes defined in Section 3.11.1.2.

```
/**
 * The validation mode to be used by the provider for the persistence
 * unit.
 * <p>
 * A {@code ValidationMode} may be specified by:
 * <ul>
 * <li>the {@code <validation-mode>} element of {@code persistence.xml},
 * <li>the property
```

```
 *     {@value Persistence.ValidationProperties#VALIDATION_MODE}, or
 * <li>{@link PersistenceConfiguration#validationMode(ValidationMode)}.
 * </ul>
 *
 * @see jakarta.persistence.spi.PersistenceUnitInfo#getValidationMode()
 * @see PersistenceConfiguration#validationMode()
 * @see Persistence.ValidationProperties#VALIDATION_MODE
 *
 * @since 2.0
 */
public enum ValidationMode {

    /**
     * If a Jakarta Validation provider is present in the environment,
     * the persistence provider must perform the automatic validation
     * of entities. If no Jakarta Validation provider is present in the
     * environment, no lifecycle event validation takes place.
     * This is the default behavior.
     * <p>
     * To ensure deterministic behavior and prevent silent validation omissions,
     * applications should explicitly set the {@code validation-mode}
     * to {@link ValidationMode#CALLBACK} rather than relying on the default.
     */
    AUTO,

    /**
     * The persistence provider must perform the lifecycle event
     * validation. It is an error if there is no Jakarta Validation
     * provider present in the environment.
     */
    CALLBACK,

    /**
     * The persistence provider must not perform lifecycle event
     * validation.
     */
    NONE
}
```

9.6.1. jakarta.persistence.spi.ClassTransformer Interface

The `ClassTransformer` interface found in Section E.1 may be implemented by a persistence provider to transform entities and managed classes at class load time or at class redefinition time. A persistence provider is not required to implement this interface.

9.7. jakarta.persistence.Persistence Class

The `Persistence` class may be found in Section B.22.

The `Persistence` class is used to obtain an `EntityManagerFactory` instance in Java SE environments. It may also be used for schema generation, that is, to create database schemas and/or DDL scripts, as specified above in Section 9.4.

The `Persistence` class is also available in a Jakarta EE container environment; however, support for the Java SE bootstrapping APIs is not required in container environments.

The `Persistence` class is used to obtain a `PersistenceUtil` instance in both Jakarta EE and Java SE environments.

9.7.1. Properties for Java SE Bootstrap

In a Java SE environment, any standard or vendor-specific property or hint used during the creation of the

`EntityManagerFactory` may be specified:

- by including it as an entry in a `Map` passed as the second argument to the `createEntityManagerFactory()` method of `Persistence`, or
- by calling the `property()` method of `PersistenceConfiguration`.

The following properties correspond to the elements and attributes in the `persistence.xml` file. When one of these properties is specified via the `Map` parameter of the `createEntityManagerFactory()` method, its value overrides the value of any corresponding element or attribute specified in the `persistence.xml` file for the named persistence unit, along with any default value assigned by the persistence provider.

Property	Type	Corresponding element in `persistence.xml`	Notes
`jakarta.persistence.provider`	`String`	`provider`	See Section 8.2.1.4.
`jakarta.persistence.qualifiers`	`String[]`	`qualifier`	See Section 8.2.1.5.
`jakarta.persistence.scope`	`String`	`scope`	See Section 8.2.1.5.
`jakarta.persistence.transactionType`	`String`	`transaction-type`	See Section 8.2.1.2.
`jakarta.persistence.jtaDataSource`	`String`	`jta-data-source`	See Section 8.2.1.7.
`jakarta.persistence.nonJtaDataSource`	`String`	`non-jta-data-source`	See Section 8.2.1.7.
`jakarta.persistence.sharedCache.mode`	`String`	`shared-cache-mode`	See Section 8.2.1.9.
`jakarta.persistence.validation.mode`	`String`	`validation-mode`	Legal values are `auto`, `callback`, and `none`. See Section 8.2.1.10 and Section 3.11.1.2.
`jakarta.persistence.defaultToOneFetchType`	`String`	`default-to-one-fetch-type`	Legal values are `EAGER` and `LAZY`. See Section 11.1

The following properties correspond to properties in the `persistence.xml` file. When one of these properties is specified via the `Map` parameter of the `createEntityManagerFactory` method, its value overrides the value of any corresponding `<property>` in the `persistence.xml` file for the named persistence unit, along with any default value assigned by the persistence provider.

Property	Type	Corresponding property in `persistence.xml`	Notes
`jakarta.persistence.lock.timeout`	`Integer` or `String`	`jakarta.persistence.lock.timeout`	Hint only. Value in milliseconds for pessimistic lock timeout. See Section 3.9.4.3.

Property	Type	Corresponding property in `persistence.xml`	Notes
`jakarta.persistence.query.timeout`	`Integer` or `String`	`jakarta.persistence.query.timeout`	Hint only. Value in milliseconds for query timeout. See Section 3.15.5.
`jakarta.persistence.validation.group.pre-persist`	`String`	`jakarta.persistence.validation.group.pre-persist`	See Section 8.2.1.12 and Section 3.11.2.1.
`jakarta.persistence.validation.group.pre-update`	`String`	`jakarta.persistence.validation.group.pre-update`	See Section 8.2.1.12 and Section 3.11.2.1.
`jakarta.persistence.validation.group.pre-remove`	`String`	`jakarta.persistence.validation.group.pre-remove`	See Section 8.2.1.12 and Section 3.11.2.1.
`jakarta.persistence.schema-generation.create-script-source`	`String`	`jakarta.persistence.schema-generation.create-script-source`	See Section 8.2.1.12.
`jakarta.persistence.schema-generation.drop-script-source`	`String`	`jakarta.persistence.schema-generation.drop-script-source`	See Section 8.2.1.12.
`jakarta.persistence.sql-load-script-source`	`String`	`jakarta.persistence.sql-load-script-source`	See Section 8.2.1.12.
`jakarta.persistence.schema-generation.database.action`	`String`	`jakarta.persistence.schema-generation.database.action`	See Section 8.2.1.12.
`jakarta.persistence.schema-generation.scripts.action`	`String`	`jakarta.persistence.schema-generation.scripts.action`	See Section 8.2.1.12.
`jakarta.persistence.schema-generation.create-source`	`String`	`jakarta.persistence.schema-generation.create-source`	See Section 8.2.1.12.
`jakarta.persistence.schema-generation.drop-source`	`String`	`jakarta.persistence.schema-generation.drop-source`	See Section 8.2.1.12.
`jakarta.persistence.schema-generation.scripts.create-target`	`String`	`jakarta.persistence.schema-generation.scripts.create-target`	See Section 8.2.1.12.
`jakarta.persistence.schema-generation.scripts.drop-target`	`String`	`jakarta.persistence.schema-generation.scripts.drop-target`	See Section 8.2.1.12.
`jakarta.persistence.schema-generation.create-database-schemas`	`String`	`jakarta.persistence.schema-generation.create-database-schemas`	See Section 8.2.1.12.
`jakarta.persistence.database-product-name`	`String`	`jakarta.persistence.database-product-name`	See Section 8.2.1.12.
`jakarta.persistence.database-major-version`	`String`	`jakarta.persistence.database-major-version`	See Section 8.2.1.12.
`jakarta.persistence.database-minor-version`	`String`	`jakarta.persistence.database-minor-version`	See Section 8.2.1.12.

The following additional standard properties are defined by this specification for the configuration of the entity manager factory:

Property	Value
`jakarta.persistence.jdbc.driver`	Fully qualified name of the driver class.
`jakarta.persistence.jdbc.url`	Driver-specific JDBC URL as a string.
`jakarta.persistence.jdbc.user`	Username for database connection.
`jakarta.persistence.jdbc.password`	Password for database connection authentication.
`jakarta.persistence.dataSource`	Instance of `javax.sql.DataSource` to be used for the specified persistence unit.
`jakarta.persistence.jdbc.fetchSize`	Override the default JDBC fetch size.
`jakarta.persistence.jdbc.batchSize`	Hint to enable JDBC statement batching by setting a batch size.
`jakarta.persistence.validation.factory`	Instance of `jakarta.validation.ValidatorFactory`.
`jakarta.persistence.bean.manager`	Instance of `jakarta.enterprise.inject.spi.BeanManager`.

Any number of vendor-specific properties may also be included in the map. If a persistence provider does not recognize a property (other than a property defined by this specification), the provider must ignore it.

Vendors should use vendor namespaces for properties (e.g., `com.acme.persistence.logging`). Entries that make use of the namespace `jakarta.persistence` and its subnamespaces must not be used for vendor-specific information. The namespace `jakarta.persistence` is reserved for use by this specification.

9.8. jakarta.persistence.PersistenceConfiguration Class

The `PersistenceConfiguration` class found in Section B.23 is used to programmatically define and configure a persistence unit and create an `EntityManagerFactory` instance directly. Thus, `PersistenceConfiguration` is an alternative to XML-based configuration using `persistence.xml`, and so the configuration options available via this API reflect the similarly-named elements of `persistence.xml`. See Section 8.2.

A programmatically-configured persistence unit is considered a Java SE persistence unit, even when this API is used within the Jakarta EE environment.[7]

A persistence provider may define a subclass of `PersistenceConfiguration` with vendor-specific configuration options. A provider must support configuration via any instance of `PersistenceConfiguration` or of any subclass of `PersistenceConfiguration`. If a subclass defines configuration options the provider does not recognize, it should ignore those options.

9.9. PersistenceUtil Interface

The `PersistenceUtil` interface found in Section B.24 is used to determine the load state of entity instances. The semantics of the methods of this interface are defined in Section 9.9.1 below.

9.9.1. Contracts for Determining the Load State of an Entity or Entity Attribute

The implementation of the `PersistenceUtil.isLoaded(Object)` method must determine the list of persistence providers available in the runtime environment[8] and call the `ProviderUtil.isLoaded(Object)` method on each of them until either:

- one provider returns `LoadState.LOADED`. In this case `PersistenceUtil.isLoaded` returns `true`.
- one provider returns `LoadState.NOT_LOADED`. In this case `PersistenceUtil.isLoaded` returns `false`.
- all providers return `LoadState.UNKNOWN`. In this case `PersistenceUtil.isLoaded` returns `true`.

If the `PersistenceUtil` implementation determines that only a single provider is available in the environment, it is permitted to use provider-specific methods to determine the result of `isLoaded(Object)` as long as the semantics defined in Section 3.6.12 are observed.

The implementation of the *PersistenceUtil.isLoaded(Object,String)* method must determine the list of persistence providers available in the environment and call the `ProviderUtil.isLoadedWithoutReference` method on each of them until either:

- one provider returns `LoadState.LOADED`. In this case `PersistenceUtil.isLoaded` returns `true`.
- one provider returns `LoadState.NOT_LOADED`. In this case `PersistenceUtil.isLoaded` returns `false`.
- all providers return `LoadState.UNKNOWN`. In this case, the `PersistenceUtil.isLoaded` method then calls `ProviderUtil.isLoadedWithReference` on each of the providers until:
  - one provider returns `LoadState.LOADED`. In this case `PersistenceUtil.isLoaded` return `true`.
  - one provider returns `LoadState.NOT_LOADED`. In this case, `PersistenceUtil.isLoaded` returns `false`.
  - all providers return `LoadState.UNKNOWN`. In this case, `PersistenceUtil.isLoaded` returns `true`.

If the `PersistenceUtil` implementation determines that only a single provider is available in the environment, it is permitted to use provider specific methods to determine the result of *isLoaded(Object, String)* as long as the semantics defined in Section 3.6.12 are observed.

> The rationale for splitting the determination of load state between the methods isLoadedWithoutReference and isLoadedWithReference is the following.

- *It is assumed that the provider that loaded the entity is present in the environment.*
- *Providers that use bytecode enhancement don't need to access an attribute reference to determine its load state, and can determine if the entity has been provided by them.*
- *By first querying all providers using bytecode enhancement, it is ensured that no attribute will be loaded by side effect.*
- *Proxy-based providers do need to access an attribute reference to determine load state, but will not trigger attribute loading as a side effect.*
- *If no provider recognizes an entity as provided by it, it is assumed to be an object that is not instrumented and is considered loaded.*

9.10. Integration with CDI

Entity listeners, defined in Section 3.10 and attribute converters, defined in Section 3.13, may be defined to make use of dependency injection through the Contexts and Dependency Injection (CDI) API [7] when CDI is available[9]. In this section, we refer to such objects as *CDI-aware*.

A CDI-aware class that makes use of dependency injection may also define lifecycle callback methods annotated with the `@PostConstruct` and `@PreDestroy` annotations. These methods are invoked after injection has taken place and before the attribute converter instance is destroyed, respectively.

An instance of a CDI-aware class is a noncontextual object and does not have a well-defined scope. It is not a CDI bean.

In a Jakarta EE environment, a persistence provider must support dependency injection for CDI-aware classes. A persistence provider is encouraged to also support dependency injection in other environments where a CDI `BeanManager` is available, but that is not a requirement of this specification. If CDI is not available, the persistence

provider must not instantiate a CDI-aware class.

Section 9.1 specifies how a persistence provider may obtain the `BeanManager` in a Jakarta EE environment.

Where CDI-aware classes are supported, the persistence provider is responsible for using the standard CDI SPI defined in [7] to:

- create an instance of the CDI-aware class, perform injection upon the instance, invoke its `@PostConstruct` and `@PreDestroy` methods, if any, and, later
- dispose of the instance when it is no longer of use.

When instantiating and injecting, the persistence provider must behave as if it carries out the following steps:

1. Obtain a `BeanManager`.
2. Create an `AnnotatedType` instance for the CDI-aware class.
3. Create an `InjectionTarget` instance for the annotated type.
4. Create a `CreationalContext`.
5. Instantiate the CDI-aware class by calling the `produce()` method of `InjectionTarget`.
6. Inject the CDI-aware instance by calling the `inject()` method of `InjectionTarget` on the instance.
7. Invoke the `@PostConstruct` callback, if any, by calling the `postConstruct()` method of `InjectionTarget` on the instance.

When disposing of an instance, the persistence provider must behave as if it carries out the following steps.

- Call the `preDestroy()` method of `InjectionTarget` on the instance.
- Call the `dispose()` method of `InjectionTarget` on the instance.
- Call the `release()` method of `CreationalContext`.

Persistence providers may optimize the steps above, for example, by replacing calls to the standard CDI SPI with equivalent interactions with vendor-specific interfaces, as long as the outcome is identical.

[1] Use of these Java SE bootstrapping APIs may be supported in Jakarta EE containers; however, support for such use is not required.

[2] In dynamic environments (e.g., OSGi-based environments, containers based on dynamic kernels, etc.), the list of persistence providers may change.

[3] If a custom PersistenceProviderResolver is needed in a JavaSE environment, it must be set before Persistence.createEntityManagerFactory is called. Note, however, that the setPersistenceProviderResolver method is not intended for general use, but rather is aimed at containers maintaining a dynamic environment.

[4] In Jakarta EE environments, it is anticipated that the Jakarta EE platform provider may wish to control the creation of database schemas rather than delegate this task to the persistence provider. This property may also be specified in Java SE environments.

[5] In Jakarta EE container environments, it is generally expected that the container is responsible for executing DDL scripts, although the container is permitted to delegate this task to the persistence provider. If DDL scripts are executed in a Java SE environment, or if the Jakarta EE container delegates the execution of scripts to the persistence provider, these properties must be specified.

[6] This is intended for use in Jakarta EE environments, where the platform provider may want to control the database privileges that are available to the persistence provider. This connection is provided by the container and should be closed by the container when the schema generation request or entity manager factory creation completes. The connection provided must have credentials sufficient for the persistence provider to carry out the requested actions.

[7] Persistence units defined programmatically using the `PersistenceConfiguration` class do not support JNDI lookup or injection via the `PersistenceContext` or `PersistenceUnit` annotations.

[8] The determining of the persistence providers that are available is discussed in Section 9.3.

[9] CDI is available by default in Jakarta EE environments, as required by the Jakarta EE specification [6].

Chapter 10. Metadata Annotations

This chapter and chapter Chapter 11 define the metadata annotations introduced by this specification.

The XML schema defined in chapter Chapter 12 provides an alternative to the use of metadata annotations.

These annotations and types are in the package `jakarta.persistence`.

10.1. Entity

The `Entity` annotation specifies that the annotated class is an entity.

The `name` annotation element specifies the entity name. If the `name` element is not specified, the entity name defaults to the unqualified name of the entity class. This name is used to refer to the entity in queries. (See Section 4.3.1.) The name must be a legal Java identifier and must not be a reserved identifier in the Jakarta Persistence query language.

```
@Documented
@Target(TYPE)
@Retention(RUNTIME)
public @interface Entity {
    String name() default "";
}
```

10.2. Entity Listeners and Callback Annotations

The following annotations are used to declare entity listeners and callback methods, as specified in Section 3.10.

10.2.1. `EntityListener` and `EntityListeners`

The `EntityListener` annotation declares an entity listener class and determines the entity classes to which it applies. The entity listener receives entity lifecycle callbacks for every entity class belonging to the persistence unit and assignable to one of its declared entity lifecycle callback methods. It is not necessary to separately declare the entity classes to which the listener applies.

```
@Target(TYPE)
@Retention(RUNTIME)
@Discoverable
public @interface EntityListener {
}
```

Alternatively, the `EntityListeners` annotation specifies the entity listener classes associated with the annotated class, package, or module. This annotation may be applied:

- directly to an entity class or mapped superclass, to specify its listener classes,
- to a package descriptor to specify listener classes applying to every entity contained in the package, or
- to a module descriptor to specify listener classes applying to every entity contained in any package belonging to the module.

Entity listeners specified using this annotation do not need to be annotated `@EntityListener`.

```
@Target({TYPE, PACKAGE, MODULE})
@Retention(RUNTIME)
@Discoverable
public @interface EntityListeners {
    Class<?>[] value();
}
```

The `ExcludeSuperclassListeners` annotation specifies that the invocation of superclass listeners is to be excluded for the annotated entity class or mapped superclass and for its subclasses.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface ExcludeSuperclassListeners {
}
```

The `ExcludeDefaultListeners` annotation specifies that the invocation of default listeners is to be excluded for the annotated entity class or mapped superclass and for its subclasses.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface ExcludeDefaultListeners {
}
```

10.2.2. Callback methods

The following annotations are used to declare callback methods for the corresponding lifecycle events, as specified in Section 3.10.1. These annotations may be applied to methods of an entity class, of a mapped superclass, or of an entity listener class.

```
@Target({METHOD})
@Retention(RUNTIME)
public @interface PrePersist {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PostPersist {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PreRemove {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PostRemove {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PreMerge {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PreInsert {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PostInsert {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PreUpdate {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PostUpdate {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PreUpsert {}
```

```
@Target({METHOD})
@Retention(RUNTIME)
public @interface PostUpsert {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PreDelete {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PostDelete {}

@Target({METHOD})
@Retention(RUNTIME)
public @interface PostLoad {}
```

10.3. EntityGraph Annotations

The following annotations are used to declare named entity graphs and attribute-level fetch options. Entity graphs are specified in Section 3.12.

10.3.1. NamedEntityGraph and NamedEntityGraphs Annotations

The `NamedEntityGraph` annotation defines a named entity graph, or a subgraph of a named entity graph. The annotation must be applied to the root entity of the graph or subgraph. The nodes and subgraphs of the entity graph specify the limits of the graph of associated attributes and entities fetched when an operation which retrieves an instance or instances of the root entity is executed.

A well-defined entity graph is a tree. The application is responsible for ensuring that every named entity graph is acyclic. If a named entity graph contains a cycle, the behavior is undefined.

The `name` element assigns a name to the entity graph, and is used to identify the entity graph in calls to `EntityHandler.getEntityGraph()`. If no name is explicitly specified, the name defaults to the entity name of the annotated root entity. Entity graph names must be unique within the persistence unit, unless two `NamedEntityGraph` annotations declare the same name and one is a subgraph of the other, as specified by `@Fetch` annotations belonging to the named graph.

The `attributeNodes` element lists attributes of the annotated entity class that are to be included in the entity graph.

The `includeAllAttributes` element specifies that all attributes of the annotated entity class are to be included in the entity graph. An `attributeNode` element may still be used in conjunction with this element to specify a subgraph for the attribute.

The `subgraphs` element specifies a list of subgraphs, further specifying attributes that are managed types. These subgraphs are referenced by name from `NamedAttributeNode` definitions.

The `subclassSubgraphs` element specifies a list of subgraphs that add additional attributes for subclasses of the root entity to which the annotation is applied.

The nodes and subgraphs of a named entity graph defined using annotations may be specified by nesting `NamedAttributeNode` and `NamedSubgraph` annotations within the `NamedEntityGraph` annotation, by annotating persistent attributes with `Fetch`, or by mixing these two approaches.

The `NamedEntityGraphs` annotation can be used to specify multiple named entity graphs for the entity to which it is applied.

As an alternative to nesting `NamedAttributeNode` and `NamedSubgraph` annotations within `NamedEntityGraph`, the `Fetch`

annotation may be applied to a persistent attribute of an entity class. When its `graph` member names a named entity graph, the annotated attribute is an attribute node of that graph. Its `subgraph` member names one or more entity graphs to be attached as subgraphs to the attribute node.

```
@Target({TYPE})
@Retention(RUNTIME)
@Repeatable(NamedEntityGraphs.class)
public @interface NamedEntityGraph {
    String name() default "";
    NamedAttributeNode[] attributeNodes() default {};
    boolean includeAllAttributes() default false;
    NamedSubgraph[] subgraphs() default {};
    NamedSubgraph[] subclassSubgraphs() default {};
}

@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedEntityGraphs {
    NamedEntityGraph[] value();
}
```

10.3.2. NamedAttributeNode Annotation

The `NamedAttributeNode` annotation is used to specify an attribute node of within an entity graph or subgraph.

The `value` element specifies the name of the corresponding attribute.

The `subgraph` element is used to refer to a `NamedSubgraph` specification that further characterizes an attribute node corresponding to a managed type (entity or embeddable). The value of the `subgraph` element must correspond to the `name` used for the subgraph in the `NamedSubgraph` element. If the referenced attribute is an entity which has entity subclasses, there may be more than one `NamedSubgraph` element with this name, and the `subgraph` element is considered to refer to all of these.

The `keySubgraph` element is used to refer to a `NamedSubgraph` specification that further characterizes an attribute node corresponding to the key of a Map-valued attribute. The value of the `keySubgraph` element must correspond to the `name` used for the subgraph in the `NamedSubgraph` element. If the referenced attribute is an entity which has entity subclasses, there may be more than one `NamedSubgraph` element with this name, and the `keySubgraph` element is considered to refer to all of these.

```
@Target({})
@Retention(RUNTIME)
public @interface NamedAttributeNode {
    String value();
    String subgraph() default "";
    String keySubgraph() default "";
}
```

10.3.3. NamedSubgraph Annotation

The `NamedSubgraph` annotation is used to further define an attribute node. It is referenced by its name from the `subgraph` or `keySubgraph` element of a `NamedAttributeNode` element.

The `name` element is the name used to reference the subgraph from a `NamedAttributeNode` definition. In the case of entity inheritance, multiple subgraph elements have the same name.

The `type` element must be specified when the subgraph corresponds to a subclass of the entity type corresponding to the referencing attribute node.

The `attributeNodes` element lists attributes of the class that must be included. If the subgraph corresponds to a subclass of the class referenced by the corresponding attribute node, only subclass-specific attributes are listed.

```
@Target({})
@Retention(RUNTIME)
public @interface NamedSubgraph {
    String name();
    Class<?> type() default void.class;
    NamedAttributeNode[] attributeNodes();
}
```

10.3.4. Fetch Annotation

The `Fetch` annotation specifies options which influence how a persistent attribute is fetched.

When the `graph` member specifies the name of a named entity graph, the annotated attribute is an attribute node of the named graph, and the fetching options specified by the annotation apply when the attribute is fetched using that entity graph.

When the `graph` member is not specified, the annotation does not declare an attribute node of an entity graph. The fetching options specified by the annotation apply when the attribute is fetched without the use of an entity graph.

Multiple `Fetch` annotations may be specified for a persistent attribute.

```
@Target({FIELD, METHOD})
@Retention(RUNTIME)
@Repeatable(Fetch.Fetches.class)
public @interface Fetch {
    String graph() default "";
    String[] subgraph() default {};
    FetchType type() default FetchType.EAGER;
    int batchSize() default -1;
    CacheStoreMode cacheStoreMode() default CacheStoreMode.USE;
    CacheRetrieveMode cacheRetrieveMode() default CacheRetrieveMode.USE;
    QueryHint[] hints() default {};
}
```

The `graph` member specifies the name of an entity graph. When a graph name is specified, the annotated attribute is an attribute node of that graph. Otherwise, the annotation specifies fetching options which apply when the attribute is fetched without the use of an entity graph.

The `subgraph` member specifies the names of one or more named entity graphs to be attached as subgraphs of the entity graph named by the `graph` member, to the attribute node representing the annotated association. The annotated attribute must be an association.

The `type` member specifies the fetching policy for the annotated attribute. It overrides the value specified by the `fetch` member of the `Basic`, `OneToOne`, `ManyToOne`, `OneToMany`, `ManyToMany`, or `ElementCollection` annotation.

The `batchSize` member is a hint specifying how many entities should be fetched in each request to the database. The persistence provider may ignore this hint.

The `cacheStoreMode` member specifies whether the persistence provider should cache the associated entity after fetching it from the database.

The `cacheRetrieveMode` member specifies whether the persistence provider is permitted to read the associated entity data from the cache.

The `hints` member specifies vendor-specific fetching hints. Any hint not recognized by a provider is ignored.

10.4. Annotations for Queries

The following annotations are used to declare named queries.

10.4.1. NamedQuery Annotation

The `NamedQuery` annotation declares a named query written in the Jakarta Persistence query language.

The `name` element assigns a name to the query, which is used to identify the query in calls to `EntityHandler.createNamedQuery()`.

The `query` element must specify the query string itself, written in the Jakarta Persistence query language. It should be a legal JPQL `select` query. For the backward compatibility, it is permitted to be an `update` or `delete` statement.

The `resultClass` element specifies the Java class of each query result. When the result class is explicitly specified, either:

1. the select list of the query contains only a single item, which must be assignable to the result class,
2. the result class is `Object[].class`, or
3. the result class is a non-abstract class or record type with a constructor with the same number of parameters as the query has items in its select list, and the constructor parameter types must exactly match the types of the corresponding items in the select list.

In the first case, each query result is returned directly to the caller. In the second case, each query result is packaged in an array, with the array elements corresponding by position with the items of the query select list. In the third case, each query result is automatically packaged in a new instance of the result class by calling the matching constructor.

If the result class of a named query is not specified, it defaults to `Object` if the query has a single item in its select list or to `Object[]` if the query has multiple items in its select list. When generating the canonical static metamodel for a named query, the generator is permitted to infer a more precise type from the items in the select list.

The result class may be overridden by explicitly passing a class object to `EntityManager.createNamedQuery(String, Class)`.

The `lockMode` element specifies a lock mode for the entity instances in results returned by the query. If a lock mode other than `NONE` is specified, the query may only be executed within a persistence context with an associated active transaction. The `lockScope` element specifies the corresponding scope for a pessimistic lock.

The `hints` element may be used to specify query properties and hints. Properties defined by this specification must be observed by the provider; hints defined by this specification should be observed by the provider when possible. Vendor-specific hints that are not recognized by a provider must be ignored.

The `entityGraph` element may specify the name of a named entity graph interpreted as a load graph and applied to the entity returned by the query. The named `EntityGraph` may be overridden by calling `StatementOrTypedQuery.withEntityGraph()`.

The `NamedQuery` and `NamedQueries` annotations can be applied to any class belonging to the persistence unit, as specified in Section 8.2.1.8.

```
/**
 * Declares a named query written in the Jakarta Persistence Query
 * Language. Query names are scoped to the persistence unit. The
 * query should be a {@code SELECT} query.
 *
 * <p> This annotation may be applied to any class or interface
 * belonging to the persistence unit.
 * {@snippet :
 * @NamedQuery(name = "findAllCustomersWithName",
 *            query = "SELECT c FROM Customer c WHERE c.name LIKE :custName")
```

```
 * @Entity
 * class Customer { ... }
 * }
 *
 * <p> A named query may be executed by calling
 * {@link EntityHandler#createNamedQuery(String, Class)}.
 * {@snippet :
 * List<Customer> customers =
 *         em.createNamedQuery("findAllCustomersWithName", Customer.class)
 *             .setParameter("custName", "Smith")
 *             .getResultList();
 * }
 *
 * <p> Alternatively, a reference to a named query may be obtained
 * via the static metamodel.
 * {@snippet :
 * List<Customer> customers =
 *         em.createQuery(Customer_._findAllCustomersWithName_)
 *             .setParameter("custName", "Smith")
 *             .getResultList();
 * }
 *
 * @apiNote A named {@code UPDATE} or {@code DELETE} statement is
 *          usually declared using {@link NamedStatement}. For
 *          backward compatibility, this annotation may be used
 *          instead, but this usage is no longer encouraged.
 *
 * @see EntityHandler#createNamedQuery(String)
 * @see EntityHandler#createNamedQuery(String,Class)
 * @see NamedStatement
 *
 * @since 1.0
 */
@Repeatable(NamedQueries.class)
@Target({TYPE, PACKAGE, MODULE})
@Retention(RUNTIME)
@Discoverable
public @interface NamedQuery {

    /**
     * (Required) The name used to identify the query in calls to
     * {@link EntityHandler#createNamedQuery}.
     */
    String name();

    /**
     * (Required) The query string in the Jakarta Persistence
     * query language.
     */
    String query();

    /**
     * (Optional) The class of each query result.
     *
     * <p>When the result class is explicitly specified, either:
     * <ol>
     * <li>the select list of the query contains only a single
     *     item, which must be assignable to the result class,
     * <li>the result class is {@code Object[].class}, or
     * <li>the result class is a non-abstract class or record
     *     type with a constructor with the same number of
     *     parameters as the query has items in its select list,
     *     and the constructor parameter types must exactly
     *     match the types of the corresponding items in the
     *     select list.
     * </ol>
     * <p>In the first case, each query result is returned
```

```
     * directly to the caller. In the second case, each query
     * result is packaged in an array with the array elements
     * corresponding by position with the items of the query
     * select list. In the third case, each query result is
     * automatically packaged in a new instance of the result
     * class by calling the matching constructor.
     *
     * <p>If the result class of a named query is not specified:
     * it defaults to {@code Object} if the query has a single
     * item in its select list or to {@code Object[]} if the
     * query has multiple items in its select list. When
     * generating the canonical static metamodel for a named
     * query, the generator is permitted to infer a more precise
     * type from the items in the select list.
     *
     * <p>The result class may be overridden by explicitly
     * passing a class object to
     * {@link EntityHandler#createNamedQuery(String, Class)}.
     *
     * @since 3.2
     */
    Class<?> resultClass() default void.class;

    /**
     * (Optional) The lock mode type to use in query execution.
     * If a {@code lockMode} other than {@link LockModeType#NONE}
     * is specified, the query must be executed in a transaction
     * and the persistence context joined to the transaction.
     * @since 2.0
     * @see TypedQuery#setLockMode
     */
    LockModeType lockMode() default LockModeType.NONE;

    /**
     * (Optional) The pessimistic lock scope to use in query execution
     * if a pessimistic lock mode is specified via {@link #lockMode}.
     * @since 4.0
     */
    PessimisticLockScope lockScope() default PessimisticLockScope.NORMAL;

    /**
     * (Optional) Query properties and hints. May include
     * vendor-specific query hints.
     * @see Query#setHint
     */
    QueryHint[] hints() default {};

    /**
     * (Optional) The name of a {@linkplain NamedEntityGraph named
     * entity graph} interpreted as a load graph and applied to the
     * entity returned by the query. The named {@link EntityGraph}
     * may be overridden by calling
     * {@link StatementOrTypedQuery#withEntityGraph(EntityGraph)}.
     * @since 4.0
     */
    String entityGraph() default "";

    /**
     * (Optional) The {@linkplain QueryFlushMode query flush mode}
     * to use when executing the query.
     * @see Query#setQueryFlushMode(QueryFlushMode)
     * @since 4.0
     */
    QueryFlushMode flush() default QueryFlushMode.DEFAULT;
}
```

```
@Target({})
@Retention(RUNTIME)
public @interface QueryHint {
    String name();
    String value();
}
```

10.4.2. NamedStatement Annotation

The `NamedStatement` annotation declares a named statement written in the Jakarta Persistence query language.

The `name` element assigns a name to the statement, which is used to identify the statement in calls to `EntityHandler.createNamedStatement()`.

The `statement` element must specify the statement string itself, written in the Jakarta Persistence query language. It must be a legal JPQL `update` or `delete` statement.

The `hints` element may be used to specify query properties and hints. Properties defined by this specification must be observed by the provider; hints defined by this specification should be observed by the provider when possible. Vendor-specific hints that are not recognized by a provider must be ignored.

The `NamedStatement` and `NamedStatements` annotations can be applied to any class belonging to the persistence unit, as specified in Section 8.2.1.8.

10.4.3. NamedNativeQuery Annotation

The `NamedNativeQuery` annotation defines a named native SQL query.

The `name` element assigns a name to the query, which is used to identify the query in calls to `EntityHandler.createNamedQuery()`.

The `query` element must specify the query string itself, written in the native SQL dialect of the database. The query should return a result set. For backward compatibility it is permitted to be a statement which returns a row count.

The `resultClass` element specifies the class of each query result. When the result class is explicitly specified and the result set mapping is not, either:

- the result class is an entity class and is interpreted as a managed entity result with implicit field mappings determined by the names of the columns in the result set and the object/relational mapping of the entity,
- the result class is the class of a basic type and the result set must have a single column which is interpreted as a scalar result,
- the result class is a non-abstract class or record type with a constructor with the same number of parameters as the result set has columns, and is interpreted as a constructor result including all the columns of the result set, or
- the result class is `Object[].class` and each query result is packaged in an array of type `Object[]`, with the array elements corresponding by position with the columns of the select list and column values obtained according to the default type mappings defined by the JDBC specification.

Otherwise, if a result set mapping is specified, an explicitly specified result class must agree with the type inferred from the result set mapping.

If the result class is not explicitly specified, then:

- it is inferred from the result set mapping, if any, or
- it defaults to `Object` if the result set has a single column or `Object[]` if the result set has multiple columns.

When neither result class nor result set mapping is specified, column values are obtained according to the default type mappings defined by the JDBC specification.

The result class may be overridden by explicitly passing a class object to `EntityHandler.createNamedQuery(String, Class)`.

The `resultSetMapping` element specifies the name of a `SqlResultSetMapping` specification defined elsewhere in metadata. The named `SqlResultSetMapping` is used to interpret the result set of the native SQL query. Alternatively, the elements `entities`, `classes`, and `columns` may be used to specify a result set mapping. These elements may not be used in conjunction with `resultSetMapping`.

The `hints` element may be used to specify query properties and hints. Hints defined by this specification should be observed by the provider when possible. Vendor-specific hints which are not recognized by the provider must be ignored.

The `NamedNativeQuery` and `NamedNativeQueries` annotations can be applied to any class belonging to the persistence unit, as specified in Section 8.2.1.8.

```
/**
 * Declares a named native SQL query and, optionally, the mapping
 * of the result of the native SQL query. Query names are scoped
 * to the persistence unit. A named query may be executed by
 * calling {@link EntityHandler#createNamedQuery(String, Class)}.
 * The SQL query should return a result set.
 *
 * <p> In simple cases, a {@link #resultClass} specifies how the
 * native SQL query result set should be interpreted, for example:
 * {@snippet :
 * @NamedNativeQuery(
 *         name = "findWidgets",
 *         query = "SELECT o.id, o.quantity, o.item " +
 *                 "FROM Order o, Item i " +
 *                 "WHERE (o.item = i.id) AND (i.name = 'widget')",
 *         resultClass = com.acme.Order.class
 * )
 * }
 *
 * <p>
 * In more complicated cases, a {@linkplain SqlResultSetMapping
 * result set mapping} is needed, which may be specified using
 * either a separate annotation:
 * {@snippet :
 * @NamedNativeQuery(
 *         name = "OrderItems",
 *         query = "SELECT o.id, o.quantity, o.item, i.id, i.name, i.description " +
 *                 "FROM Order o, Item i " +
 *                 "WHERE (o.quantity > 25) AND (o.item = i.id)",
 *         resultSetMapping = "OrderItemResults"
 * )
 * @SqlResultSetMapping(
 *         name = "OrderItemResults",
 *         entities = {
 *                 @EntityResult(entityClass=com.acme.Order.class),
 *                 @EntityResult(entityClass=com.acme.Item.class)
 *         }
 * )
 * }
 * or using the elements of this annotation:
 * {@snippet :
 * @NamedNativeQuery(
 *         name = "OrderItems",
 *         query = "SELECT o.id, o.quantity, o.item, i.id, i.name, i.description " +
 *                 "FROM Order o, Item i " +
 *                 "WHERE (o.quantity > 25) AND (o.item = i.id)",
```

```
 *         resultSetMapping = "OrderItemResults",
 *         entities = {
 *                 @EntityResult(entityClass=com.acme.Order.class),
 *                 @EntityResult(entityClass=com.acme.Item.class)
 *         }
 * )
 * }
 *
 * <p> This annotation may be applied to any class or interface
 * belonging to the persistence unit.
 *
 * @apiNote A named native SQL statement which returns a row count
 *          is usually declared using {@link NamedNativeStatement}.
 *          For backward compatibility, this annotation may be used
 *          instead, but this usage is no longer encouraged.
 *
 * @see SqlResultSetMapping
 * @see EntityHandler#createNamedQuery(String)
 * @see EntityHandler#createNamedQuery(String,Class)
 * @see NamedNativeStatement
 *
 * @since 1.0
 */
@Repeatable(NamedNativeQueries.class)
@Target({TYPE, PACKAGE, MODULE})
@Retention(RUNTIME)
@Discoverable
public @interface NamedNativeQuery {

    /**
     * The name used to identify the query in calls to
     * {@link EntityHandler#createNamedQuery}.
     */
    String name();

    /**
     * The native SQL query string.
     */
    String query();

    /**
     * The class of each query result.
     *
     * <p>When the result class is explicitly specified and the
     * {@linkplain #resultSetMapping result set mapping} is not, either:
     * <ul>
     * <li>the result class is an entity class and is interpreted as a
     *     managed {@linkplain EntityResult entity result} with implicit
     *     field mappings determined by the names of the columns in the
     *     result set and the object/relational mapping of the entity,
     * <li>the result class is the class of a {@linkplain Basic basic}
     *     type and the result set must have a single column which is
     *     interpreted as a {@linkplain ColumnResult scalar result},
     * <li>the result class is a non-abstract class or record type
     *     with a constructor with the same number of parameters as the
     *     result set has columns, and is interpreted as a
     *     {@linkplain ConstructorResult constructor result} including
     *     all the columns of the result set, or
     * <li>the result class is {@code Object[].class} and each query
     *     result is packaged in an array of type {@code Object[]},
     *     with the array elements corresponding by position with the
     *     columns of the select list and column values obtained
     *     according to the default type mappings defined by the JDBC
     *     specification.
     * </ul>
     *
     * <p>Otherwise, if a {@linkplain #resultSetMapping result set mapping}
```

```
 * is specified, an explicitly specified result class must agree with
 * the type inferred from the result set mapping.
 *
 * <p>If the result class is not explicitly specified, then:
 * <ul>
 * <li>it is inferred from the {@linkplain #resultSetMapping result
 *     set mapping}, if any, or
 * <li>it defaults to {@code Object} if the result set has a single
 *     column or {@code Object[]} if the result set has multiple
 *     columns.
 * </ul>
 * <p>When neither result class nor result set mapping is specified,
 * column values are obtained according to the default type mappings
 * defined by the JDBC specification.
 *
 * <p>The result class may be overridden by explicitly passing a class
 * object to {@link EntityHandler#createNamedQuery(String, Class)}.
 */
Class<?> resultClass() default void.class;

/**
 * The {@linkplain SqlResultSetMapping#name name} of a
 * {@linkplain SqlResultSetMapping result set mapping} declared in
 * metadata. The named result set mapping is used to interpret the
 * result set of the native SQL query.
 *
 * <p>Alternatively, the elements {@link #entities}, {@link #classes},
 * and {@link #columns} may be used to specify a result set mapping.
 * These elements may not be used in conjunction with
 * {@code resultSetMapping}.
 *
 * @see SqlResultSetMapping
 * @see ResultSetMapping
 */
String resultSetMapping() default "";

/**
 * Specifies the result set mapping to entities.
 * May not be used in combination with {@link #resultSetMapping}.
 * @since 3.2
 */
EntityResult[] entities() default {};

/**
 * Specifies the result set mapping to constructors.
 * May not be used in combination with {@link #resultSetMapping}.
 * @since 3.2
 */
ConstructorResult[] classes() default {};

/**
 * Specifies the result set mapping to scalar values.
 * May not be used in combination with {@link #resultSetMapping}.
 * @since 3.2
 */
ColumnResult[] columns() default {};

/**
 * Query properties and hints.
 * (May include vendor-specific query hints.)
 * @see Query#setHint
 */
QueryHint[] hints() default {};

/**
 * (Optional) The {@linkplain QueryFlushMode query flush mode}
 * to use when executing the query.
```

```
     * @see Query#setQueryFlushMode(QueryFlushMode)
     * @since 4.0
     */
    QueryFlushMode flush() default QueryFlushMode.DEFAULT;
}
```

10.4.4. NamedNativeStatement Annotation

The `NamedNativeStatement` annotation defines a named native SQL statement that returns a row count.

The `name` element assigns a name to the statement, which is used to identify the statement in calls to `EntityHandler.createNamedStatement()`.

The `statement` element must specify the statement string itself, written in the native SQL dialect of the database.

The `hints` element may be used to specify query properties and hints. Hints defined by this specification should be observed by the provider when possible. Vendor-specific hints which are not recognized by the provider must be ignored.

The `NamedNativeStatement` and `NamedNativeStatements` annotations can be applied to any class belonging to the persistence unit, as specified in Section 8.2.1.8.

10.4.5. NamedStoredProcedureQuery Annotation

The `NamedStoredProcedureQuery` annotation is used to specify a stored procedure, its parameters, and its result type.

The `name` element is the name that is passed as an argument to the `createNamedStoredProcedureQuery` method to create an executable `StoredProcedureQuery` object.

The `procedureName` element is the name of the stored procedure in the database.

The parameters of the stored procedure are specified by the `parameters` element. All parameters must be specified in the order in which they occur in the parameter list of the stored procedure.

The `resultClasses` element refers to the class (or classes) that are used to map the results. The `resultSetMappings` element names one or more result set mappings, as defined by the `SqlResultSetMapping` annotation.

If there are multiple result sets, it is assumed that they will be mapped using the same mechanism—e.g., either all via a set of result class mappings or all via a set of result set mappings. The order of the specification of these mappings must be the same as the order in which the result sets will be returned by the stored procedure invocation. If the stored procedure returns one or more result sets and no `resultClasses` or `resultSetMappings` element is specified, any result set will be returned as a list of type `Object[]` . The combining of different strategies for the mapping of stored procedure result sets is undefined.

The `hints` element may be used to specify query properties and hints. Properties defined by this specification must be observed by the provider. Vendor-specific hints that are not recognized by a provider must be ignored.

The `NamedStoredProcedureQuery` and `NamedStoredProcedureQueries` annotations can be applied to an entity or mapped superclass.

```
/**
 * Declares and names a stored procedure, its parameters, and its result type.
 *
 * <p>The {@code NamedStoredProcedureQuery} annotation can be applied to an
 * entity or mapped superclass.
 *
 * <p>The {@link #name} element is the name that is passed as an argument
 * to the {@link EntityHandler#createNamedStoredProcedureQuery} method to
```

```
 * create an executable {@link StoredProcedureQuery} object. Names are
 * scoped to the persistence unit.
 *
 * <p>The {@link #procedureName} element is the name of the stored procedure
 * in the database.
 *
 * <p>The parameters of the stored procedure are specified by the
 * {@link #parameters} element. Parameters must be specified in the order
 * in which they occur in the parameter list of the stored procedure.
 *
 * <p>The {@link #resultClasses} element refers to the class (or classes)
 * that are used to map the results. The {@link #resultSetMappings} element
 * names one or more result set mappings, as defined by the
 * {@link SqlResultSetMapping} annotation.
 *
 * <p>If there are multiple result sets, it is assumed that they are
 * mapped using the same mechanism &#8212; e.g., either all via a set of
 * result class mappings or all via a set of result set mappings. The
 * order of the specification of these mappings must be the same as the
 * order in which the result sets are returned by the stored procedure
 * invocation. If the stored procedure returns one or more result sets
 * and no {@link #resultClasses} or {@link #resultSetMappings} are
 * specified, any result set is returned as a list of type
 * {@code Object[]}. The combining of different strategies for the mapping
 * of stored procedure result sets is undefined.
 *
 * <p>The {@link #hints} element may be used to specify query properties
 * and hints. Properties defined by this specification must be observed
 * by the provider. Vendor-specific hints that are not recognized by a
 * provider must be ignored.
 *
 * <p>All parameters of a named stored procedure query must be specified
 * using the {@code StoredProcedureParameter} annotation.
 *
 * @see StoredProcedureQuery
 * @see StoredProcedureParameter
 * @see EntityHandler#createNamedStoredProcedureQuery(String)
 *
 * @since 2.1
 */
@Repeatable(NamedStoredProcedureQueries.class)
@Target({TYPE, PACKAGE, MODULE})
@Retention(RUNTIME)
@Discoverable
public @interface NamedStoredProcedureQuery {

    /**
     * The name used to refer to the query with the {@link EntityManager}
     * methods that create stored procedure query objects.
     */
    String name();

    /**
     * The name of the stored procedure in the database.
     */
    String procedureName();

    /**
     * Information about all parameters of the stored procedure.
     * Parameters must be specified in the order in which they
     * occur in the parameter list of the stored procedure.
     */
    StoredProcedureParameter[] parameters() default {};

    /**
     * The result classes.
     *
```

```
     * <p>When the result classes are explicitly specified and the
     * {@linkplain #resultSetMappings result set mappings} are not, either:
     * <ul>
     * <li>the result class is an entity class and is interpreted as a
     *     managed {@linkplain EntityResult entity result} with implicit
     *     field mappings determined by the names of the columns in the
     *     result set and the object/relational mapping of the entity,
     * <li>the result class is the class of a {@linkplain Basic basic}
     *     type and the result set must have a single column which is
     *     interpreted as a {@linkplain ColumnResult scalar result},
     * <li>the result class is a non-abstract class or record type
     *     with a constructor with the same number of parameters as the
     *     result set has columns, and is interpreted as a
     *     {@linkplain ConstructorResult constructor result} including
     *     all the columns of the result set, or
     * <li>the result class is {@code Object[].class} and each query
     *     result is packaged in an array of type {@code Object[]},
     *     with the array elements corresponding by position with the
     *     columns of the select list and column values obtained
     *     according to the default type mappings defined by the JDBC
     *     specification.
     * </ul>
     *
     * <p>Otherwise, if {@linkplain #resultSetMappings result set mappings}
     * are specified, each explicitly specified result class must agree with
     * the type inferred from the corresponding result set mapping.
     *
     * <p>If the result classes are not explicitly specified, then they
     * are inferred from the result set mapping, if any, or default to
     * {@code Object} or {@code Object[]}. Any result class may be
     * overridden by explicitly passing a class object to
     * {@link StoredProcedureQuery#getResultList(Class)},
     * {@link StoredProcedureQuery#getSingleResult(Class)}, or
     * {@link StoredProcedureQuery#getSingleResultOrNull(Class)}.
     */
    Class<?>[] resultClasses() default {};

    /**
     * The {@linkplain SqlResultSetMapping#name names} of one or more
     * {@linkplain SqlResultSetMapping result set mappings} declared in
     * metadata. The named result set mappings are used to interpret the
     * result sets of the stored procedure.
     *
     * <p>Any result set mapping may be overridden by explicitly passing
     * a {@link ResultSetMapping} to
     * {@link StoredProcedureQuery#getResultList(ResultSetMapping)},
     * {@link StoredProcedureQuery#getSingleResult(ResultSetMapping)}, or
     * {@link StoredProcedureQuery#getSingleResultOrNull(ResultSetMapping)}.
     *
     * @see SqlResultSetMapping
     * @see ResultSetMapping
     */
    String[] resultSetMappings() default {};

    /**
     * Query properties and hints.
     * (May include vendor-specific query hints.)
     */
    QueryHint[] hints() default {};

}
```

Every parameter of a named stored procedure query must be declared using the `StoredProcedureParameter` annotation. The `name` element refers to the name of the parameter as defined by the stored procedure in the database. If a parameter name is not specified, it is assumed that the stored procedure uses positional parameters. The `mode` element

specifies whether the parameter is an IN, INOUT, OUT, or REF_CURSOR parameter. REF_CURSOR parameters are used by some databases to return result sets from stored procedures. The `type` element refers to the JDBC type for the parameter.

```
@Target({})
@Retention(RUNTIME)
public @interface StoredProcedureParameter {
    String name() default "";
    ParameterMode mode() default ParameterMode.IN;
    Class<?> type();
}


public enum ParameterMode {
    IN,
    INOUT,
    OUT,
    REF_CURSOR
}
```

10.4.6. Annotations for SQL Result Set Mappings

The `SqlResultSetMapping` annotation is used to specify the mapping of the result set of a native SQL query or stored procedure.

```
/**
 * Specifies an explicit mapping of the columns of a result set of a native
 * SQL query or stored procedure to {@linkplain EntityResult entity classes},
 * {@linkplain ColumnResult scalar values}, and {@linkplain ConstructorResult
 * Java class constructors}. Every SQL result set mapping has a {@link #name}.
 * SQL result set mapping names must be unique within a persistence unit.
 *
 * <p>When a SQL result set mapping specifies more than one result mapping,
 * each row of the mapped query result set is represented as an instance of
 * {@code Object[]} whose elements are, in order:
 * <ol>
 * <li>entity results, in the order in which they are declared by the
 *     {@link #entities} element;
 * <li>constructor results, in the order declared by the {@link #classes}
 *     element; and then
 * <li>scalar results, in the order declared by the {@link #columns}
 *     element.
 * </ol>
 *
 * <p>In this example, a named mapping is declared by annotating an entity
 * class:
 * {@snippet :
 * @SqlResultSetMapping(
 *         name = "orderResults",
 *         entities = @EntityResult(
 *             entityClass = Order.class,
 *             fields = {
 *                 @FieldResult(name = Order_.ID, column = "order_id"),
 *                 @FieldResult(name = Order_.TOTAL, column = "order_total"),
 *                 @FieldResult(name = Order_.ITEM, column = "order_item")
 *             }
 *         ),
 *         columns = @ColumnResult(name = "item_name")
 * )
 * @Entity
 * class Order { ... }
 * }
 * <p>A reference to the mapping is obtained from the
 * {@linkplain jakarta.persistence.metamodel.StaticMetamodel static metamodel}
```

```
 * class of the entity:
 * {@snippet :
 * List<Order> orders =
 *         entityManager.createNativeQuery(
 *                 """
 *                   SELECT o.id AS order_id,
 *                          o.total AS order_total,
 *                          o.item_id AS order_item,
 *                          i.desc_name AS item_name
 *                   FROM orders o
 *                   JOIN order_items i
 *                     ON o.id = i.order_id
 *                   WHERE o.total > 25
 *                 """,
 *                 Order_._orderResults
 *         ).getResultList();
 * }
 *
 * <p>A {@code SqlResultSetMapping} may be reified at runtime as
 * an instance of {@link jakarta.persistence.sql.ResultSetMapping}.
 * A reified representation of a {@code SqlResultSetMapping} known
 * to the persistence unit may be obtained by calling
 * {@link EntityManagerFactory#getResultSetMappings(Class)} or
 * from the static metamodel class of the annotated class.
 * {@snippet :
 * ResultSetMapping<Order> mapping =
 *         entityManager.getResultSetMappings(Order.class)
 *              .get(Order_.MAPPING_ORDER_RESULTS);
 * }
 *
 * @see NamedNativeQuery#resultSetMapping
 * @see NamedStoredProcedureQuery#resultSetMappings
 * @see EntityHandler#createNativeQuery(String, String)
 *
 * @see jakarta.persistence.sql.ResultSetMapping
 * @see jakarta.persistence.sql.CompoundMapping
 * @see EntityManagerFactory#getResultSetMappings(Class)
 *
 * @since 1.0
 */
@Repeatable(SqlResultSetMappings.class)
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Discoverable
public @interface SqlResultSetMapping {

    /**
     * The name given to the result set mapping. This name
     * identifies the mapping in:
     * <ul>
     * <li>methods of {@link EntityHandler} and
     *     {@link EntityManagerFactory}, and
     * <li>in {@link NamedNativeQuery#resultSetMapping} and
     *     {@link NamedStoredProcedureQuery#resultSetMappings}.
     * </ul>
     */
    String name();

    /**
     * Specifies the result set mapping to entities.
     */
    EntityResult[] entities() default {};

    /**
     * Specifies the result set mapping to constructors.
     * @since 2.1
     */
```

```
    ConstructorResult[] classes() default {};

    /**
     * Specifies the result set mapping to scalar values.
     */
    ColumnResult[] columns() default {};
}
```

The `name` element is the name given to the result set mapping, and is used to identify it when calling methods of the `EntityManager` which create instances of `Query` and `StoredProcedureQuery`. The `entities`, `classes`, and `columns` elements are used to specify the mapping of result set columns to entities, to constructors, and to scalar values, respectively.

```
@Target(METHOD)
@Retention(RUNTIME)
@Repeatable(EntityResult.EntityResults.class)
public @interface EntityResult {
    Class<?> entityClass();
    LockModeType lockMode() default LockModeType.NONE;
    FieldResult[] fields() default {};
    String discriminatorColumn() default "";
    ...
}
```

The `entityClass` element specifies the class of the result.

The `lockMode` element specifies the `LockModeType` obtained when the native SQL query is executed.

The `fields` element is used to map the columns specified in the SELECT list of the query to the properties or fields of the entity class.

The `discriminatorColumn` element specifies the column name or alias of the column in the SELECT list that is used to determine the type of the entity instance.

```
@Target({})
@Retention(RUNTIME)
public @interface FieldResult {
    String name();
    String column();
}
```

The `name` element is the name of the persistent field or property of the class.

The `column` element specifies the name of the corresponding column in the SELECT list—i.e., column alias, if applicable.

```
@Target(METHOD)
@Retention(RUNTIME)
@Repeatable(ConstructorResult.ConstructorResults.class)
public @interface ConstructorResult {
    Class<?> targetClass();
    ColumnResult[] columns() default {};
    EntityResult[] entities() default {};
    ...
}
```

The `targetClass` element specifies the class whose constructor is to be invoked.

The `columns` element specifies the mapping of columns in the SELECT list to the arguments of the intended constructor.

```
@Target(METHOD)
@Retention(RUNTIME)
@Repeatable(ColumnResult.ColumnResults.class)
```

```
public @interface ColumnResult {
    String name();
    Class<?> type() default void.class;
    ...
}
```

The `name` element specifies the name of the column in the SELECT list.

The `type` element specifies the Java type to which the column type is mapped. The specified type must be one of the basic types listed in Section 2.6. If the `type` element is not specified, the default JDBC type mapping for the column type is used.

10.5. References to EntityManager, EntityAgent, and EntityManagerFactory

These annotations are used to express dependencies on entity managers, entity agents, and entity manager factories.

10.5.1. PersistenceContext Annotation

The `PersistenceContext` annotation expresses a dependency on a container-managed entity manager and its associated persistence context. When this annotation occurs on a method or field of a managed bean, it declares an injection point of type `EntityManager`. When the annotation occurs on a managed bean class, it assigns a name to the entity manager in the environment referencing context `java:comp/env` of the containing module.

- The `name` element specifies the name at which the `EntityManager` is accessed in the environment referencing context. If the specified name does not begin with `java`, then the prefix `java:comp/env` is assumed. This member is not usually specified when `@PersistenceUnit` annotates an injection point.
- The optional `unitName` element references the name of the persistence unit as defined in the `persistence.xml` file. This member is optional if there is only one persistence unit defined by the containing module.
- The `type` element specifies whether a transaction-scoped or extended persistence context is required, as defined in Section 3.7.1. If the `type` element is not specified, a transaction-scoped persistence context is assumed.
- The `synchronizationType` element specifies whether the persistence context is always automatically synchronized with the current transaction or whether the persistence context must be explicitly joined to the current transaction by means of the `joinTransaction()` method of `EntityManager`, as defined in Section 3.7.3.
- The optional `properties` element may be used to specify properties for the container or persistence provider. Properties defined by this specification must be observed by the provider. Vendor-specific properties may be included in the set of properties and are passed to the persistence provider by the container when the entity manager is created. Properties that are not recognized by a vendor must be ignored.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(PersistenceContexts.class)
public @interface PersistenceContext {
    String name() default "";
    String unitName() default "";
    PersistenceContextType type() default TRANSACTION;
    SynchronizationType synchronization() default SYNCHRONIZED;
    PersistenceProperty[] properties() default {};
}


public enum PersistenceContextType {
    TRANSACTION,
    EXTENDED
}
```

```
public enum SynchronizationType {
    SYNCHRONIZED,
    UNSYNCHRONIZED
}


@Target({})
@Retention(RUNTIME)
public @interface PersistenceProperty {
    String name();
    String value();
}
```

The `PersistenceContexts` annotation declares one or more `PersistenceContext` annotations. It is used to express a dependency on multiple persistence contexts[1].

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface PersistenceContexts {
    PersistenceContext[] value();
}
```

10.5.2. PersistenceAgent Annotation

The `PersistenceAgent` annotation is used to express a dependency on an entity agent.

The `name` element refers to the name by which the entity agent is to be accessed in the environment referencing context and is not needed when dependency injection is used.

The optional `unitName` element refers to the name of the persistence unit as defined in the `persistence.xml` file. If the `unitName` element is specified, the persistence unit for the entity manager factory that is accessible in JNDI must have the same name.

The optional `properties` element may be used to specify properties for the container or persistence provider. Properties defined by this specification must be observed by the provider. Vendor-specific properties may be included in the set of properties and are passed to the persistence provider by the container when the entity agent is created. Properties that are not recognized by a vendor must be ignored.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(PersistenceAgents.class)
public @interface PersistenceAgent {
    String name() default "";
    String unitName() default "";
    PersistenceProperty[] properties() default {};
}
```

The `PersistenceAgents` annotation declares one or more `PersistenceAgent` annotations. It is used to express a dependency on multiple entity agents.

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface PersistenceAgents {
    PersistenceAgent[] value();
}
```

10.5.3. PersistenceUnit Annotation

The `PersistenceUnit` annotation expresses a dependency on an entity manager factory and its associated persistence unit. When this annotation occurs on a method or field of a managed bean, it declares an injection point of type `EntityManagerFactory`. When the annotation occurs on a managed bean class, it assigns a name to the entity manager factory in the environment referencing context `java:comp/env` of the containing module.

The `name` element specifies the name at which the `EntityManagerFactory` is accessed in the environment referencing context. If the specified name does not begin with `java:`, then the prefix `java:comp/env` is assumed. This member is not usually specified when `@PersistenceUnit` annotates an injection point.

The optional `unitName` element references the name of the persistence unit as defined in the `persistence.xml` file. This member is optional if there is only one persistence unit defined by the containing module.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(PersistenceUnits.class)
public @interface PersistenceUnit {
    String name() default "";
    String unitName() default "";
}
```

The `PersistenceUnits` annotation declares one or more `PersistenceUnit` annotations. It is used to express a dependency on multiple persistence units[2].

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface PersistenceUnits {
    PersistenceUnit[] value();
}
```

10.6. Annotations for Attribute Converter Classes

The `Converter` annotation declares that the annotated class is a converter and specifies whether the converter is applied automatically. Every converter class must implement `AttributeConverter` and must be annotated with the `Converter` annotation or declared as a converter in the XML descriptor. The target type for a converter is determined by the actual type argument of the first type parameter of `AttributeConverter`.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Converter {
    boolean autoApply() default false;
}
```

If the `autoApply` element is specified as `true`, the persistence provider must automatically apply the converter to every mapped attribute of the specified target type belonging to any entity in the persistence unit, except for attributes for which conversion is overridden by means of the `Convert` annotation (or XML equivalent). The `Convert` annotation is described in Section 11.2.11. The `Convert` annotation may be used to override or disable auto-apply conversion on a per-attribute basis.

In determining whether a converter applies to an attribute, the provider must treat primitive types and wrapper types as equivalent.

A converter never applies to id attributes, version attributes, relationship attributes, or to attributes explicitly annotated as `Enumerated` or `Temporal` (or designated as such via XML).

A converter never applies to any attribute annotated `@Convert(disableConversion=true)` or to an attribute for which the `Convert` annotation explicitly specifies a different converter.

If `autoApply` is `false`, the converter applies only to attributes of the target type for which conversion is explicitly enabled via the `Convert` annotation (or corresponding XML element).

If there is more than one converter defined for the same target type, the `Convert` annotation must be used to explicitly specify which converter applies.

10.7. Annotations for Static Queries

These annotations are used to declare queries associated with methods. In the context of Jakarta Data[9], they are used to declare repository query methods.

The persistence provider must treat each such static query as a named query, where the query name agrees with the convention for Javadoc links, that is: the concatenation of the fully qualified name of the type declaring the annotated method, with the string `#`, the name of the annotated method, and the raw types of the method parameters, enclosed in parentheses and separated by commas, for example, `"org.example.Library#findBooksByTitle(java.lang.String)"`.

10.7.1. JakartaQuery Annotation

Specifies a query expressed in the Jakarta Persistence Query Language. This annotation may be applied to:

- any method of any class or interface belonging to the persistence unit, or
- a query method of a Jakarta Data repository with an implementation backed by Jakarta Persistence.

The `value` member specifies the query.

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface JakartaQuery {
    String value();
}
```

The semantics of this annotation are defined by its Javadoc, and in Section 5.2.

The persistence provider must treat the annotation `jakarta.data.repository.Query` as a synonym for `JakartaQuery`.

10.7.2. NativeQuery Annotation

Specifies a query expressed in the native SQL dialect of the database. This annotation may be applied to:

- any method of any class or interface belonging to the persistence unit, or
- a query method of a Jakarta Data repository with an implementation backed by Jakarta Persistence.

The `value` member specifies the query.

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface NativeQuery {
    String value();
}
```

The semantics of this annotation are defined by its Javadoc, and in Section 5.2.

10.7.3. QueryOptions Annotation

Configures options that affect the execution of a query. This annotation may be applied to:

- a method with a `JakartaQuery` or Jakarta Data `jakarta.data.repository.Query` annotation,
- a method with a `NativeQuery` annotation, or
- a Jakarta Data repository method annotated `jakarta.data.repository.Find`.

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface QueryOptions {
    CacheStoreMode cacheStoreMode() default CacheStoreMode.USE;
    CacheRetrieveMode cacheRetrieveMode() default CacheRetrieveMode.USE;
    int timeout() default -1;
    QueryHint[] hints() default {};
    LockModeType lockMode() default LockModeType.NONE;
    PessimisticLockScope lockScope() default PessimisticLockScope.NORMAL;
    String entityGraph() default "";
    QueryFlushMode flush() default QueryFlushMode.DEFAULT;
}
```

[1] A dependency on ultiple persistence contexts may be needed, for example, when multiple persistence units are used.

[2] Multiple persistence units may be needed, for example, when mapping to multiple databases.

Chapter 11. Metadata for Object/Relational Mapping

The object/relational mapping metadata is part of the application domain model contract. It expresses requirements and expectations on the part of the application as to the mapping of the entities and relationships of the application domain to a database. Queries (and, in particular, SQL queries) written against the database schema that corresponds to the application domain model are dependent upon the mappings expressed by means of the object/relational mapping metadata. The implementation of this specification must assume this application dependency upon the object/relational mapping metadata and ensure that the semantics and requirements expressed by that mapping are observed.

The use of object/relational mapping metadata to control schema generation is specified in Section 11.3.

11.1. Fetching Policies

The `FetchType` enum defines policies for fetching associated data from the database:

```
public enum FetchType { LAZY, EAGER, DEFAULT }
```

- The `EAGER` policy is a requirement on the persistence provider runtime that data must be eagerly fetched.
- The `LAZY` policy is a *hint* to the persistence provider runtime that data should be fetched lazily when it is first accessed, in the case of a managed entity associated with a persistence context, or on a call to the `fetch()` method of `EntityAgent`, in the case of a detached entity. The implementation is permitted to eagerly fetch data for which the `LAZY` policy hint has been specified.[1]
- A `DEFAULT` value specifies that the fetching policy depends on the *default fetch type of the persistence unit for one-to-one and many-to-one associations*. This default fetch type is controlled via the `default-to-one-fetch-type` element in the `persistence.xml` file or by setting the `defaultToOneFetchType` of the `PersistenceConfiguration` object used to bootstrap the persistence unit, and defaults to `EAGER` for backward compatibility.

> It is very strongly recommended that new applications set the default fetch type for one-to-one and many-to-one associations to `LAZY` to avoid accidental fetching of unnecessary data.

11.2. Annotations for Object/Relational Mapping

These annotations and types are in the package `jakarta.persistence`.

XML metadata may be used as an alternative to these annotations, or to override or augment annotations, as described in Chapter 12.

11.2.1. Access Annotation

The `Access` annotation specifies an access type to be applied to an entity class, mapped superclass, or embeddable class, or to a specific attribute of such a class.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Access {
    AccessType value();
}
```

Table 4 lists the annotation elements that may be specified for the `Access` annotation.

Table 4. Access Annotation Elements

Type	Name	Description	Default
AccessType	value	(Required) The access type to be applied to the class or attribute.	

11.2.2. AssociationOverride Annotation

The `AssociationOverride` annotation is used to override a mapping for an entity relationship.

The `AssociationOverride` annotation may be applied to an entity that extends a mapped superclass to override a relationship mapping defined by the mapped superclass. If the `AssociationOverride` annotation is not specified, the association is mapped the same as in the original mapping. When used to override a mapping defined by a mapped superclass, the `AssociationOverride` annotation is applied to the entity class.

The `AssociationOverride` annotation may be used to override a relationship mapping from an embeddable within an entity to another entity when the embeddable is on the owning side of the relationship. When used to override a relationship mapping defined by an embeddable class (including an embeddable class embedded within another embeddable class), the `AssociationOverride` annotation is applied to the field or property containing the embeddable.

When the `AssociationOverride` annotation is used to override a relationship mapping from an embeddable class, the `name` element specifies the referencing relationship field or property within the embeddable class. To override mappings at multiple levels of embedding, a dot (".") notation syntax must be used in the `name` element to indicate an attribute within an embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property. When the `AssociationOverride` annotation is applied to override the mappings of an embeddable class used as a map value, " `value.` " must be used to prefix the name of the attribute within the embeddable class that is being overridden in order to specify it as part of the map value.[2]

If the relationship mapping is a foreign key mapping, the `joinColumns` element of the `AssociationOverride` annotation is used. If the relationship mapping uses a join table, the `joinTable` element of the `AssociationOverride` element must be specified to override the mapping of the join table and/or its join columns.[3]

The `joinColumns` element refers to the table for the class that contains the annotation.

The `foreignKey` element specifies or controls the generation of a foreign key constraint for the columns corresponding to the `joinColumns` element when table generation is in effect. If both this element and the `foreignKey` element of any of the `joinColumns` elements are specified, the behavior is undefined.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(AssociationOverrides.class)
public @interface AssociationOverride {
    String name();

    JoinColumn[] joinColumns() default {};

    ForeignKey foreignKey() default

    @ForeignKey(PROVIDER_DEFAULT);
    JoinTable joinTable() default @JoinTable;
}
```

Table 5 lists the annotation elements that may be specified for the `AssociationOverride` annotation.

Table 5. AssociationOverride Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the relationship property whose mapping is being overridden if property-based access is being used, or the name of the relationship field if field-based access is used.	
JoinColumn[]	joinColumns	The join column(s) being mapped to the persistent attribute(s). The joinColumns element must be specified if a foreign key mapping is used in the overriding of the mapping of the relationship. The joinColumns element must not be specified if a join table is used in the overriding of the mapping of the relationship	
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default
JoinTable	joinTable	The join table that maps the relationship. The joinTable element must be specified if a join table is used in the overriding of the mapping of the relationship. The joinTable element must not be specified if a foreign key mapping is used in the overriding of the mapping of the relationship.	.

Example 1:

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer id;

    @Version
    protected Integer version;

    @ManyToOne
    protected Address address;
```

```
    public Integer getId() { ... }

    public void setId(Integer id) { ... }

    public Address getAddress() { ... }

    public void setAddress(Address address) { ... }
}

@Entity
@AssociationOverride(name="address", joinColumns=@JoinColumn(name="ADDR_ID"))
public class PartTimeEmployee extends Employee {
    // address field mapping overridden to ADDR_ID foreign key
    @Column(name="WAGE")
    protected Float hourlyWage;

    public Float getHourlyWage() { ... }

    public void setHourlyWage(Float wage) { ... }
}
```

Example 2: Overriding of the mapping for the `phoneNumbers` relationship defined in the `ContactInfo` embeddable class.

```
@Entity
public class Employee {
    @Id
    int id;

    @AssociationOverride(
        name="phoneNumbers",
        joinTable=@JoinTable(
            name="EMPPHONES",
            joinColumns=@JoinColumn(name="EMP"),
            inverseJoinColumns=@JoinColumn(name="PHONE")
        )
    )
    @Embedded
    ContactInfo contactInfo;

    // ...
}

@Embeddable
public class ContactInfo {
    @ManyToOne Address address; // Unidirectional
    @ManyToMany(targetEntity=PhoneNumber.class)
    List phoneNumbers;
}

@Entity
public class PhoneNumber {
    @Id
    int number;

    @ManyToMany(mappedBy="contactInfo.phoneNumbers")
    Collection<Employee> employees;
}
```

11.2.3. AssociationOverrides Annotation

The mappings of multiple relationship properties or fields may be overridden. The `AssociationOverrides` annotation can be used for this purpose.

```
@Target({TYPE, METHOD, FIELD})
```

```
@Retention(RUNTIME)
public @interface AssociationOverrides {
    AssociationOverride[] value();
}
```

Table 6 lists the annotation elements that may be specified for the `AssociationOverrides` annotation.

Table 6. AssociationOverrides Annotation Elements

Type	Name	Description	Default
AssociationOverride[]	value	(Required) The association override mappings that are to be applied to the relationship field or property.	

Example:

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer id;

    @Version
    protected Integer version;

    @ManyToOne
    protected Address address;

    @OneToOne
    protected Locker locker;

    public Integer getId() { ... }

    public void setId(Integer id) { ... }

    public Address getAddress() { ... }

    public void setAddress(Address address) { ... }

    public Locker getLocker() { ... }

    public void setLocker(Locker locker) { ... }
}

@Entity
@AssociationOverrides({
    @AssociationOverride(name="address", joinColumns=@JoinColumn("ADDR_ID")),
    @AssociationOverride(name="locker", joinColumns=@JoinColumn("LCKR_ID"))})
public PartTimeEmployee { ... }
```

Alternatively:

```
@Entity
@AssociationOverride(name="address", joinColumns=@JoinColumn("ADDR_ID"))
@AssociationOverride(name="locker", joinColumns=@JoinColumn("LCKR_ID"))
public PartTimeEmployee { ... }
```

11.2.4. AttributeOverride Annotation

The `AttributeOverride` annotation is used to override the mapping of a `Basic` (whether explicit or default) property or field or `Id` property or field.

The `AttributeOverride` annotation may be applied to an entity that extends a mapped superclass or to an embedded field or property to override a `Basic` mapping or `Id` mapping defined by the mapped superclass or embeddable class (or embeddable class of one of its attributes).

The `AttributeOverride` annotation may be applied to an element collection containing instances of an embeddable class or to a map collection whose key and/or value is an embeddable class. When the `AttributeOverride` annotation is applied to a map, " `key.` " or " `value.` " must be used to prefix the name of the attribute that is being overridden in order to specify it as part of the map key or map value.

To override mappings at multiple levels of embedding, a dot (".") notation form must be used in the `name` element to indicate an attribute within an embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

If the `AttributeOverride` annotation is not specified, the column is mapped the same as in the original mapping.

Table 7 lists the annotation elements that may be specified for the `AttributeOverride` annotation.

The `column` element refers to the table for the class that contains the annotation.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(AttributeOverrides.class)
public @interface AttributeOverride {
    String name();
    Column column();
}
```

Table 7. AttributeOverride Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the property whose mapping is being overridden if property-based access is being used, or the name of the field if field-based access is used.	
Column	column	(Required) The column that is being mapped to the persistent attribute. The mapping type will remain the same as is defined in the embeddable class or mapped superclass.	

Example 1:

```
@MappedSuperclass
public class Employee {
```

```
    @Id
    protected Integer id;

    @Version
    protected Integer version;

    protected String address;

    public Integer getId() { ... }

    public void setId(Integer id) { ... }

    public String getAddress() { ... }

    public void setAddress(String address) { ... }
}

@Entity
@AttributeOverride(name="address", column=@Column(name="ADDR"))
public class PartTimeEmployee extends Employee {
    // address field mapping overridden to ADDR
    protected Float wage();

    public Float getHourlyWage() { ... }

    public void setHourlyWage(Float wage) { ... }
}
```

Example 2:

```
@Embeddable public class Address {
    protected String street;

    protected String city;

    protected String state;

    @Embedded
    protected Zipcode zipcode;
}

@Embeddable
public class Zipcode {
    protected String zip;
    protected String plusFour;
}

@Entity
public class Customer {
    @Id
    protected Integer id;

    protected String name;

    @AttributeOverride(name="state", column=@Column(name="ADDR_STATE"))
    @AttributeOverride(name="zipcode.zip", column= @Column(name="ADDR_ZIP"))
    @Embedded
    protected Address address;

    // ...
}
```

Example 3:

```
@Entity
```

```
public class PropertyRecord {
    @EmbeddedId
    PropertyOwner owner;

    @AttributeOverrides(name="key.street", column=@Column(name="STREET_NAME"))
    @AttributeOverride(name="value.size", column=@Column(name="SQUARE_FEET"))
    @AttributeOverride(name="value.tax", column=@Column(name="ASSESSMENT"))
    @ElementCollection
    Map<Address, PropertyInfo> parcels;
}

@Embeddable
public class PropertyInfo {
    Integer parcelNumber;
    Integer size;
    BigDecimal tax;
}
```

11.2.5. AttributeOverrides Annotation

The mappings of multiple properties or fields may be overridden. The `AttributeOverrides` annotation can be used for this purpose.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface AttributeOverrides {
    AttributeOverride[] value();
}
```

Table 8 lists the annotation elements that may be specified for the `AttributeOverrides` annotation.

Table 8. AttributeOverrides Annotation Elements

Type	Name	Description	Default
AttributeOverride[]	value	(Required) The AttributeOverride mappings that are to be applied to the field or property.	

Example:

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="startDate", column=@Column(name="EMP_START")),
    @AttributeOverride(name="endDate", column=@Column(name="EMP_END"))
})
public EmploymentPeriod getEmploymentPeriod() { ... }
```

11.2.6. Basic Annotation

The `Basic` annotation is the simplest type of mapping to a database column. The `Basic` annotation may be applied to any persistent property or instance variable whose type is one of the basic types listed in Section 2.6.

For the types listed above, the persistence provider must support mappings to the column types listed in tables B-2 and B-4 of Appendix B of the JDBC 4.3 specification. See [3]. In addition, the provider must support mapping:

- `java.time.Instant` to the JDBC `TIMESTAMP` or `TIMESTAMP_WITH_TIMEZONE` type,

- `java.time.Year` to the JDBC `INTEGER` and `SMALLINT` types,
- `java.math.BigInteger` and `java.math.BigDecimal` to the JDBC `NUMERIC` and `DECIMAL` types,
- `java.util.UUID` to the JDBC `CHAR` and `VARCHAR` types, and
- `char[]` to the JDBC `CHAR`, `NCHAR`, `VARCHAR`, `NVARCHAR`, `LONGVARCHAR`, and `LONGNVARCHAR` types.

As described in Section 2.10, the use of the `Basic` annotation is optional for persistent fields and properties of the types listed above. If the `Basic` annotation is not specified for such a field or property, the default values of the `Basic` annotation will apply.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

Table 9 lists the annotation elements that may be specified for the `Basic` annotation and their default values.

If the annotated field or property is of primitive type, or if it is also annotated with `jakarta.annotation.Nonnull`, then `optional=false` is implied, and the value of the `optional` annotation member is ignored.

Table 9. Basic Annotation Elements

Type	Name	Description	Default
FetchType	fetch	(Optional) Whether the value of the field or property should be lazily loaded or must be eagerly fetched. The `EAGER` policy is a requirement on the persistence provider runtime that the value must be eagerly fetched. The `LAZY` policy is a hint to the persistence provider runtime.	EAGER
boolean	optional	(Optional) Whether the value of the field or property may be null. May be used in schema generation. Ignored if the field or property is of primitive type or is annotated `Nonnull`.	true

Example 1:

```
@Basic
protected String name;
```

Example 2:

```
@Basic(fetch=LAZY)
protected String getName() { return name; }
```

If the persistence provider stores a value of type `java.util.UUID` in a column of type `VARCHAR` or equivalent, the value must be stored in its canonical representation, unless the application explicitly indicates that some other representation is preferred.

11.2.7. Cacheable Annotation

The `Cacheable` annotation specifies whether an entity should be cached in the second-level cache, as defined in Section 3.14, when:

- the persistence provider offers a second-level cache,
- caching is enabled, and
- the `SharedCacheMode` specified by the `shared-cache-mode` element in `persistence.xml`, by the `sharedCacheMode()` of the `PersistenceConfiguration` object, or by the configuration property `jakarta.persistence.sharedCache.mode`, is set to `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE`, as described in Section 3.14.1.

The value of the `Cacheable` annotation is inherited by subclasses; it can be overridden by explicitly specifying `Cacheable` on a subclass.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Cacheable {
    boolean value() default true;
}
```

A persistence provider is never *required* to store an entity in the second-level cache.

- `Cachable(true)` (or simply `Cacheable`) means that the entity and its state may be cached in the second-level cache, at the discretion of the provider, even if `ENABLE_SELECTIVE` is specified or if the superclass of the entity is not cacheable.
- `Cacheable(false)` means that the entity and its state must not be cached in the second-level cache, even if `DISABLE_SELECTIVE` is specified, or if the superclass of the entity is cacheable.

When the shared cache mode is `ALL` or `NONE`, the `Cacheable` annotation must be ignored by the provider.

The semantics of the `Cacheable` annotation are undefined when:

- the `shared-cache-mode` element is not specified in the `persistence.xml` file or the `sharedCacheMode()` of the `PersistenceConfiguration` is `UNSPECIFIED`, and
- the property `jakarta.persistence.sharedCache.mode` is not specified when the entity manager factory for the persistence unit is created.

In this case, the behavior is specific to the persistence provider.

Table 10. Cacheable Annotation Elements

Type	Name	Description	Default
boolean	value	(Optional) Whether or not the entity should be cached.	true

11.2.8. CheckConstraint Annotation

The `CheckConstraint` annotation is used to specify a SQL check constraint on a column or table when schema generation is in effect.

```
@Target({})
```

```
@Retention(RUNTIME)
public @interface CheckConstraint {
    String name() default "";
    String constraint();
    String options() default "";
}
```

The `name` element specifies a name for the check constraint.

The `constraint` member specifies a SQL expression written in the native SQL dialect of the database.

Table 11. CheckConstraint Annotation Elements

Type	Name	Description	Default
String	name	(Optional) Constraint name.	A provider-chosen name.
String	constraint	(Required) The native SQL expression to be checked.	
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended.

The semantics of the second-level cache are extremely vendor-specific. Portable applications should not rely on any specific behavior connected with the second-level cache. Most implementations of this specification require additional configuration beyond the use of the `Cacheable` annotation to make use of the second-level cache. This specification does not even require that a persistence provider provide a second-level cache.

11.2.9. CollectionTable Annotation

The `CollectionTable` annotation is used in the mapping of collections of basic or embeddable types. The `CollectionTable` annotation specifies the table that is used for the mapping of the collection and is specified on the collection-valued field or property.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface CollectionTable {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    JoinColumn[] joinColumns() default {};
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
    CheckConstraint[] check() default {};
    String comment() default "";
    String type() default "";
    String options() default "";
}
```

By default, the columns of the collection table that correspond to the embeddable class or basic type are derived from the attributes of the embeddable class or from the basic type according to the default values of the `Column` annotation, as described in Section 11.2.10. In the case of a basic type, the column name is derived from the name of the collection-valued field or property. In the case of an embeddable class, the column names are derived from the field or property

names of the embeddable class.

To override the default properties of the column used for a basic type, the `Column` annotation is used on the collection-valued attribute in addition to the `ElementCollection` annotation. The value of the `table` element of the `Column` annotation defaults to the name of the collection table.

To override these defaults for an embeddable class, the `AttributeOverride` and/or `AttributeOverrides` annotations must be used in addition to the `ElementCollection` annotation. The value of the `table` element of the `Column` annotation used in the `AttributeOverride` annotation defaults to the name of the collection table. If the embeddable class contains references to other entities, the default values for the columns corresponding to those references may be overridden by means of the `AssociationOverride` and/or `AssociationOverrides` annotations.

The `foreignKey` element specifies or controls the generation of a foreign key constraint for the columns corresponding to the `joinColumns` element when table generation is in effect. If both this element and the `foreignKey` element of any of the `joinColumns` elements are specified, the behavior is undefined. If no `foreignKey` annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

If the `CollectionTable` annotation is missing, the default values of the `CollectionTable` annotation elements apply.

This annotation may not be applied to a persistent field or property not annotated `@ElementCollection`.

Table 12 lists the annotation elements that may be specified for the `CollectionTable` annotation and their default values.

Table 12. CollectionTable Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the collection table.	The concatenation of the name of the containing entity and the name of the collection attribute, separated by an underscore.
String	catalog	(Optional) The catalog of the table.	Default catalog.
String	schema	(Optional) The schema of the table.	Default schema for user.

Type	Name	Description	Default
JoinColumn[]	joinColumns	(Optional) The foreign key columns of the collection table which reference the primary table of the entity.	(Default only applies if a single join column is used.) The same defaults as for JoinColumn (i.e., the concatenation of the following: the name of the entity; "_"; the name of the referenced primary key column.) However, if there is more than one join column, a JoinColumn annotation must be specified for each join column using the JoinColumns annotation. Both the name and the referencedColumnName elements must be specified in each such JoinColumn annotation.
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes
CheckConstraint[]	check	(Optional) Check constraints for the table. These are only used if table generation is in effect.	No check constraint
String	comment	(Optional) Comment for the table. This is only used if table generation is in effect.	No comment
String	type	(Optional) A SQL fragment indicating the type of table.	Nothing inserted
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended.

Example:

```
@Embeddable
public class Address {
    protected String street;
    protected String city;
    protected String state;

    // ...
}

@Entity public class Person {
    @Id
    protected String ssn;

    protected String name;

    protected Address home;

    // ...

    @ElementCollection // use default table (PERSON_NICKNAMES)
    @Column(name="name", length=50)
    protected Set<String> nickNames = new HashSet();

    // ...
}

@Entity
public class WealthyPerson extends Person {
    @ElementCollection
    @CollectionTable(name="HOMES") // use default join column name
    @AttributeOverrides({
        @AttributeOverride(name="street", column=@Column(name="HOME_STREET")),
        @AttributeOverride(name="city", column=@Column(name="HOME_CITY")),
        @AttributeOverride(name="state", column=@Column(name="HOME_STATE"))
    })
    protected Set<Address> vacationHomes = new HashSet();

    // ...
}
```

11.2.10. Column Annotation

The `Column` annotation specifies a mapped column for a persistent property or field.

Table 13 lists the annotation elements that may be specified for the `Column` annotation and their default values.

If no `Column` annotation is specified, the default values in Table 13 apply.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {
    String name() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String options() default "";
    String table() default "";
    int length() default 255;
    int precision() default 0; // decimal precision
    int scale() default 0; // decimal scale
```

```
    int secondPrecision() default -1; //fractional second precision
    CheckConstraint[] check() default {};
    String comment() default "";
}
```

Table 13. Column Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the column.	The property or field name.
boolean	unique	(Optional) Whether the column is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint corresponds to only a single column. This constraint applies in addition to any constraint entailed by primary key mapping and to constraints specified at the table level.	false
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column. The specified DDL must be written in the native SQL dialect of the target database, and is not portable across databases.	Generated SQL to create a column of the inferred type.

Type	Name	Description	Default
String	options	(Optional) A SQL fragment appended to the generated DDL. The specified DDL must be written in the native SQL dialect of the target database, and is not portable across databases.	Nothing appended.
String	table	(Optional) The name of the table that contains the column. If absent the column is assumed to be in the primary table for the mapped object.	Column is in primary table.
int	length	(Optional) The column length Applies only to columns whose type is parameterized by length, for example, varchar or varbinary types.	255
int	precision	(Optional) The precision of a column of SQL type `decimal` or `numeric`, or of similar database-native type. Applies only to columns of exact numeric type. The default value `0` indicates that a provider-determined precision should be inferred.	0
int	scale	(Optional) The scale of a column of SQL type `decimal` or `numeric`, or of similar database-native type. Applies only to columns of exact numeric type. The default value `0` indicates that a provider-determined scale should be inferred.	0

Type	Name	Description	Default
int	secondPrecision	(Optional) The number of decimal digits to use for storing fractional seconds in a SQL time or timestamp column. Applies only to columns of time or timestamp type. The default value `-1` indicates that fractional seconds should not be stored in a time column, or that the maximum number of digits supported by the database and JDBC driver should be stored in a timestamp column.	-1
CheckConstraint[]	check	(Optional) Check constraints for the column. These are only used if table generation is in effect.	No check constraint
String	comment	(Optional) Comment for the column. This is only used if table generation is in effect.	No comment

Portable applications which make use of schema generation should explicitly specify the `precision` and `scale` of columns of type `numeric` or `decimal`.

Example 1:

```
@Column(name="DESC", nullable=false, length=512)
public String getDescription() {
    return description;
}
```

Example 2:

```
@Column(name="DESC", columnDefinition="CLOB NOT NULL", table="EMP_DETAIL")
@Lob
public String getDescription() {
    return description;
}
```

Example 3:

```
@Column(name="ORDER_COST", updatable=false, precision=12, scale=2)
public BigDecimal getCost() {
    return cost;
}
```

11.2.11. Convert Annotation

The `Convert` annotation specifies how the values of a field or property are converted to a basic type:

- enabling a converter which was defined `autoApply=false`,
- overriding the use of a converter which was defined `autoApply=true` (see Section 10.6), or
- overriding the use of a converter specified by a field or property of an embedded type or inherited mapped superclass.

When property access is used, the `Convert` must be applied to the getter method.

It is not necessary to use the `Basic` annotation or corresponding XML element to specify the converted basic type. Nor is it usually necessary to explicitly specify the converter class, except to disambiguate cases where multiple converters would otherwise apply.

The `Convert` annotation may be applied to an entity that extends a mapped superclass to specify or override the conversion mapping for an inherited basic attribute.

```
@Target({METHOD, FIELD, TYPE})
@Retention(RUNTIME)
@Repeatable(Converts.class)
public @interface Convert {
    Class<? extends AttributeConverter> converter() default AttributeConverter.class;
    String attributeName() default "";
    boolean disableConversion() default false;
}
```

Table 14 lists the annotation elements that may be specified for the `Convert` annotation.

Table 14. Convert Annotation Elements

Type	Name	Description	Default
Class	converter	(Optional) The converter to be applied.	No converter
String	attributeName	(Optional) The name of the attribute to convert. Must be specified unless the `Convert` annotation is applied to an attribute of basic type or to an element collection of basic type. Must not be specified otherwise.	The basic attribute or basic element collection attribute to which the annotation is applied
boolean	disableConversion	(Optional) Whether conversion of the attribute is to be disabled.	false

The `converter` element specifies the converter that is applied. Even if an automatically-applied converter would otherwise be applicable to the annotated field or property, the converter specified by the `converter` element must be applied instead.

The `disableConversion` element specifies that any automatically-applied converter that would otherwise be applicable to the given field or property must not be applied.

If neither the `converter` element nor the `disableConversion` element is specified, and there is exactly one converter for the type of the annotated field or property, that converter is applied, even if it is not an automatically-applied converter.

If multiple converters are applicable to the annotated field or property, and the `converter` element is not specified, the behavior is undefined.

The `Convert` annotation should not be used to specify conversion of id attributes, (including the attributes of embedded ids and derived identities), of version attributes, of relationship attributes, or of attributes explicitly annotated (or designated via XML) as `Enumerated` or `Temporal`. Applications that depend on such conversions are not portable.

The `Convert` annotation may be applied to:

- a basic attribute, or
- a collection attribute (that is, an `ElementCollection`) of any type other than `Map`, in which case the converter is applied to the elements of the collection.

In these cases, the `attributeName` element must not be specified.

Alternatively, the `Convert` annotation may be applied to:

- an embedded attribute,
- a collection attribute (that is, an `ElementCollection`) whose element type is an embeddable type, in which case the converter is applied to the specified attribute of the embeddable instances contained in the collection,
- a map collection attribute (that is, an `ElementCollection` of type `Map`), in which case the converter is applied to the keys or values of the map, or to the specified attribute of the embeddable instances contained in the map, or
- an entity class which extends a mapped superclass, to enable or override conversion of an inherited basic or embedded attribute.

In these cases, the `attributeName` element must be specified.

To override conversion mappings at multiple levels of embedding, a dot (“.”) notation form must be used in the `attributeName` element to indicate an attribute within an embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

The dot notation may also be used with map entries:

- When the `Convert` annotation is applied to a map to specify conversion of a map key or value of basic type, `"key"` or `"value"`, respectively, must be used as the value of the `attributeName` element to specify that it is the map key or value that is converted.
- When the `Convert` annotation is applied to a map whose key or value type is an embeddable type, the `attributeName` element must be specified, and `"key."` or `"value."` (respectively) must be used to prefix the name of the attribute of the key or value type that is converted.

Example 1: Convert a basic attribute

```
@Converter
public class BooleanToIntegerConverter implements AttributeConverter<Boolean, Integer> { ... }

@Entity
public class Employee {
    @Id
    long id;

    @Convert(converter=BooleanToIntegerConverter.class)
    boolean fullTime;

    // ...
```

```
}
```

Example 2: Auto-apply conversion of a basic attribute

```
@Converter(autoApply=true)
public class EmployeeDateConverter implements
    AttributeConverter<com.acme.EmployeeDate, java.sql.Date> { ... }

@Entity
public class Employee {
    @Id
    long id;

    // ...

    // EmployeeDateConverter is applied automatically
    EmployeeDate startDate;
}
```

Example 3: Disable conversion in the presence of an auto-apply converter

```
@Convert(disableConversion=true)
EmployeeDate lastReview;
```

Example 4: Apply a converter to an element collection of basic type

```
@ElementCollection
// applies to each element in the collection
@Convert(converter=NameConverter.class)
List<String> names;
```

Example 5: Apply a converter to an element collection that is a map of basic values. The converter is applied to the map `value`.

```
@ElementCollection
@Convert(converter=EmployeeNameConverter.class)
Map<String, String> responsibilities;
```

Example 6: Apply a converter to a map key of basic type

```
@OneToMany
@Convert(converter=ResponsibilityCodeConverter.class, attributeName="key")
Map<String, Employee> responsibilities;
```

Example 7: Apply a converter to an embeddable attribute

```
@Embedded
@Convert(converter=CountryConverter.class, attributeName="country")
Address address;
```

Example 8: Apply a converter to a nested embeddable attribute

```
@Embedded
@Convert(converter=CityConverter.class, attributeName="region.city")
Address address;
```

Example 9: Apply a converter to a nested attribute of an embeddable that is a map key of an element collection

```
@Entity
```

```
public class PropertyRecord {
    // ...

    @Convert(converter=CityConverter.class, attributeName="key.region.city")
    @ElementCollection
    Map<Address, PropertyInfo> parcels;
}
```

Example 10: Apply a converter to an embeddable that is a map key for a relationship

```
@OneToMany
@Convert(converter=ResponsibilityCodeConverter.class, attributeName="key.jobType")
Map<Responsibility, Employee> responsibilities;
```

Example 11: Override conversion mappings for attributes inherited from a mapped superclass

```
@Entity
@Convert(converter=DateConverter.class, attributeName="startDate")
@Convert(converter=DateConverter.class, attributeName="endDate")
public class FullTimeEmployee extends GenericEmployee { ... }
```

11.2.12. Converts Annotation

The `Converts` annotation can be used to group `Convert` annotations. Multiple converters must not be applied to the same basic attribute.

```
@Target({METHOD, FIELD, TYPE})
@Retention(RUNTIME)
public @interface Converts {
    Convert[] value();
}
```

Table 15 lists the annotation elements that may be specified for the `Converts` annotation.

Table 15. Converts Annotation Elements

Type	Name	Description	Default
Convert[]	value	(Required) The Convert mappings that are to be applied to the entity or the field or property.	

Example: Multiple converters applied to an embedded attribute

```
@Embedded
@Converts({
    @Convert(converter=CountryConverter.class, attributeName="country"),
    @Convert(converter=CityConverter.class, attributeName="region.city")
})
Address address;
```

11.2.13. DiscriminatorColumn Annotation

Depending on the inheritance mapping strategy, as defined in Section 2.14, for an entity class hierarchy, a discriminator column might be used to determine the entity subclass which represents a given row in the database.

- For the `SINGLE_TABLE` mapping strategy, a discriminator column is always necessary.
- For the `JOINED` strategy, some persistence providers require the use of a discriminator column.
- For the `TABLE_PER_CLASS` strategy, a discriminator column is never necessary.

The `DiscriminatorColumn` annotation declares a discriminator column.

The inheritance mapping strategy, and discriminator column, if any, are determined by the mapping metadata for the root entity of an entity class hierarchy[4] Therefore, the `DiscriminatorColumn` annotation must be applied to a root entity class in an entity class hierarchy, even if the root class is an `abstract` class.

Every discriminator column has a well-defined type. The discriminator types allowed by this specification are enumerated by `DiscriminatorType`:

```
public enum DiscriminatorType { STRING, CHAR, INTEGER }
```

If the type of a discriminator column is not explicitly specified by the `DiscriminatorColumn` annotation or equivalent XML element, the type defaults to `STRING`.

If the `DiscriminatorColumn` annotation is missing, and a discriminator column is required by the mapping strategy and persistence provider:

- the name of the discriminator column defaults to `DTYPE` and
- the discriminator type defaults to `STRING`.

Table 16 lists the annotation elements that may be specified for the `DiscriminatorColumn` annotation, along with their default values.

The type of the discriminator column, if specified in the optional `columnDefinition` element, must be consistent with the discriminator type.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface DiscriminatorColumn {
    String name() default "DTYPE";
    DiscriminatorType discriminatorType() default STRING;
    String columnDefinition() default "";
    String options() default "";
    int length() default 31;
    String comment() default "";
}
```

Table 16. DiscriminatorColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of column to be used for the discriminator.	"DTYPE"
DiscriminatorType	discriminatorType	(Optional) The type of object/column to use as a class discriminator.	DiscriminatorType.STRING

Type	Name	Description	Default
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the discriminator column.	Provider-generated SQL to create a column of the specified discriminator type.
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended.
int	length	(Optional) The column length for String-based discriminator types. Ignored for other discriminator types.	31
String	comment	(Optional) Comment for the column. This is only used if table generation is in effect.	No comment

Example:

```
@Entity
@Table(name="CUST")
@DiscriminatorColumn(name="DISC", discriminatorType=STRING, length=20)
public class Customer { ... }

@Entity
public class ValuedCustomer extends Customer { ... }
```

11.2.14. DiscriminatorValue Annotation

The `DiscriminatorValue` annotation specifies the value of a discriminator column for rows represented by entities of a given type. This annotation must be applied to a concrete entity class.

If the `DiscriminatorValue` annotation is not specified for a given concrete entity subclass in an entity inheritance hierarchy which makes use of a discriminator column:

- if the discriminator type is `STRING`, the discriminator value defaults to the entity name, as defined in Section 10.1, or
- otherwise, a provider-specific function is used to generate a default value representing the entity type.

A portable application should explicitly specify the `DiscriminatorValue` for every entity subclass when the discriminator type is anything other than `STRING`.

The inheritance mapping strategy, and discriminator column, if any, are determined by the mapping metadata for the root entity of an entity class hierarchy[5]. On the other hand, the discriminator value, if not defaulted, must be specified for every entity class in the hierarchy.

The `DiscriminatorValue` annotation has only a `value` member, as listed below in Table 17. A discriminator value must be consistent in type with the discriminator type (see Section 11.2.13) of the discriminator column for the entity class hierarchy. The `value` member of `DiscriminatorValue` accepts a string which is interpreted as a literal value of the discriminator type. If the discriminator type is `INTEGER`, the specified `value` string must be convertable to an integer value by calling `Integer.parseInt()`.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface DiscriminatorValue {
    String value();
}
```

Table 17. DiscriminatorValue Annotation Elements

Type	Name	Description	Default
String	value	(Optional) The value that indicates that the row is an entity of the annotated entity type.	If the DiscriminatorValue annotation is not specified, a provider-specific function to generate a value representing the entity type is used for the value of the discriminator column. If the DiscriminatorType is STRING, the discriminator value default is the entity name.

Example:

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="DISC", discriminatorType=STRING,length=20)
@DiscriminatorValue("CUSTOMER")
public class Customer { ... }

@Entity
@DiscriminatorValue("VCUSTOMER")
public class ValuedCustomer extends Customer { ... }
```

11.2.15. ElementCollection Annotation

The `ElementCollection` annotation declares an element collection, as defined in Section 2.8. The elements of the collection must be instances of a basic type or embeddable class.

The `ElementCollection` annotation (or equivalent XML element) must be explicitly specified if such a collection maps to a collection table.[6]

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ElementCollection {
    Class targetClass() default void.class;
    FetchType fetch() default LAZY;
}
```

Table 18 lists the annotation elements that may be specified for the `ElementCollection` annotation, along with their default values.

Table 18. ElementCollection Annotation Elements

Type	Name	Description	Default
Class	targetClass	(Optional) The basic or embeddable class that is the element type of the collection. Optional only if the collection field or property is defined using Java generics. Must be specified otherwise.	The parameterized type of the collection when defined using generics.
FetchType	fetch	(Optional) Whether the collection should be lazily loaded or must be eagerly fetched. The `EAGER` policy is a requirement on the persistence provider runtime that the collection elements must be eagerly fetched. The `LAZY` policy is a hint to the persistence provider runtime.	LAZY

Example:

```
@Entity public class Person {
    @Id
    protected String ssn;

    protected String name;

    @ElementCollection
    protected Set<String> nickNames = new HashSet();

    // ...
}
```

11.2.16. Embeddable Annotation

The `Embeddable` annotation declares that the annotated class is an embeddable class, as defined in Section 2.7, whose instances are stored as an intrinsic part of an owning entity and share the persistent identity of the owning entity.

```
@Documented
@Target({TYPE})
@Retention(RUNTIME)
public @interface Embeddable {

}
```

Example 1:

```
@Embeddable
public class EmploymentPeriod {
    @Temporal(DATE)
    java.util.Date startDate;
```

```
    @Temporal(DATE)
    java.util.Date endDate;

    // ...
}
```

Example 2:

```
@Embeddable
public class PhoneNumber {
    protected String areaCode;
    protected String localNumber;

    @ManyToOne
    PhoneServiceProvider provider;

    // ...
}

@Entity
public class PhoneServiceProvider {
    @Id
    protected String name;

    // ...
}
```

Example 3:

```
@Embeddable
public class Address {
    protected String street;
    protected String city;
    protected String state;

    @Embedded
    protected Zipcode zipcode;
}

@Embeddable
public class Zipcode {
    protected String zip;
    protected String plusFour;
}
```

11.2.17. Embedded Annotation

The `Embedded` annotation specifies that a persistent field or property of an entity or embeddable class is an instance of an embeddable class.[7] Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity or embeddable class. The embeddable class must be annotated as `Embeddable`.[8]

The `AttributeOverride`, `AttributeOverrides`, `AssociationOverride`, and `AssociationOverrides` annotations may be used to override mappings declared or defaulted by the embeddable class.

An implementation is not required to support embedded objects mapped across more than one table. An entity which splits the attributes of a single embeddable object across its primary and secondary tables, or across multiple distinct secondary tables, is not considered portable. Therefore, all `AttributeOverride` and `AssociationOverride` annotations for a given embedded attribute should specify columns belonging to the same `table`.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
```

```
public @interface Embedded {}
```

Example:

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="startDate", column=@Column(name="EMP_START")),
    @AttributeOverride(name="endDate", column=@Column(name="EMP_END"))
})
public EmploymentPeriod getEmploymentPeriod() { ... }
```

11.2.18. EmbeddedId Annotation

The `EmbeddedId` annotation identifies a persistent field or property of an entity class or mapped superclass as holding a composite primary key represented as an embeddable class, as defined in Section 2.4. The embeddable class must be annotated as `Embeddable`.[9]

An embedded id class must not declare a field or property representing a relationship, in the sense of Section 2.11, with another entity.

In a given entity class:

- there must be at most one field or property annotated `EmbeddedId`, and
- if there is a field or property annotated `EmbeddedId`, there must be no field or property annotated `Id`.

This rule is relaxed in the special case where the `EmbeddedId` represents the composite primary key of a parent entity in an entity with a derived identity represented as an `IdClass`, as specified in Section 2.4.2.1.

The `AttributeOverride` annotation may be used to override the column mappings declared within the embeddable class.

The `MapsId` annotation may be used in conjunction with the `EmbeddedId` annotation to specify a derived primary key, as defined in Section 2.4.2.

If an entity has a derived primary key, the `AttributeOverride` annotation may only be used to override those attributes of the embedded id that do not correspond to the relationship with the parent entity.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface EmbeddedId {}
```

Example 1:

```
@Entity
public class Employee {
    @EmbeddedId
    protected EmployeePK empPK;

    String name;

    @ManyToOne
    Set<Department> dept;

    // ...
}
```

Example 2:

```
@Embeddable
public class PaycheckId {
```

```
    int period;
    EmployeeId empPK; // corresponds to PK type of Employee
}

@Entity
public class Paycheck {
    // default column name for "name" attribute is overridden
    @AttributeOverride(name="period", @Column(name="pay_period"))
    @EmbeddedId
    PaycheckId id;

    // ...

    @MapsId("empPK")
    @ManyToOne
    Employee emp;
}
```

11.2.19. Enumerated Annotation

An `Enumerated` annotation specifies that a persistent property or field should be persisted as an enumerated type. The `Enumerated` annotation is optional if the type of a persistent field or property is a Java enum type.

The `Enumerated` annotation may be used in conjunction with the `Basic` annotation. The `Enumerated` annotation may be used in conjunction with the `ElementCollection` annotation[10] when the element type of the collection is an enum type.

An enum can be mapped as either a string or an integer[11]. The `EnumType` enum defines the available options for mapping enumerated types.

```
public enum EnumType {
    ORDINAL,
    STRING
}
```

The `value` member of the `Enumerated` annotation specifies the `EnumType`:

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Enumerated {
    EnumType value() default ORDINAL;
}
```

Table 19 lists the annotation elements that may be specified for the `Enumerated` annotation and their default values.

Table 19. Enumerated Annotation Elements

Type	Name	Description	Default
EnumType	value	(Optional) The type used in mapping an enum type.	ORDINAL

If a persistent field or property of enum type has no explicit `Enumerated` annotation, and if no converter is applied to the field or property:

- if the enum type has a final field of type `java.lang.String` annotated `EnumeratedValue`, the enumerated type is inferred to be `STRING`;
- otherwise, the enumerated type is taken to be `ORDINAL`.

The enum type may have a final field annotated `EnumeratedValue`. This field, if it exists, controls the mapping of enum

values to database column values:

- if the enum type does have a field annotated `EnumeratedValue`, each enum value is mapped to the value of the annotated field, or, otherwise,
- if the enumerated type is `ORDINAL`, each enum value is mapped to the value of the `ordinal` field, but
- if the enumerated type is `STRING`, each enum value is mapped to the value of the `name` field.

Example:

```
public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}
public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}

@Entity
public class Employee {
    // ...

    public EmployeeStatus getStatus() {...}

    @Enumerated(STRING)
    public SalaryRate getPayScale() {...}

    // ...
}
```

If the status property is mapped to a column of integer type, and the payscale property to a column of varchar type, an instance that has a status of `PART_TIME` and a pay rate of `JUNIOR` is stored with `STATUS` set to 1 and `PAYSCALE` set to *"JUNIOR"*.

11.2.20. EnumeratedValue Annotation

The `EnumeratedValue` annotation specifies that an annotated field of a Java enum type is the source of database column values when the enum occurs as the declared type of an `Enumerated` property or field. The annotated field must be declared final, and must be of type:

- `byte`, `short`, or `int` for an `ORDINAL` enumerated type, or
- `java.lang.String` for a `STRING` enumerated type.

The field must not be set to null, and must hold a distinct value for each value of the enum type.

If the type of the field annotated `EnumeratedValue` disagrees with the enumerated type mapping specified by the `Enumerated` annotation, the behavior is undefined. Portable applications should ensure that the type of the field annotated `EnumeratedValue` agrees with the type mapping wherever the enum type is used in a field or property explicitly annotated `Enumerated`.

If a converter is applied to an `Enumerated` field or property, the `EnumeratedValue` annotation is ignored for that field or property.

Example:

```
enum Status {
    OPEN(0), CLOSED(1), CANCELLED(-1);

    @EnumeratedValue
    final int intValue;

    Status(int intValue) {
        this.intValue = intValue;
    }
}
```

11.2.21. ExcludedFromVersioning Annotation

The `ExcludedFromVersioning` annotation specifies that mutation of the annotated field or property does not result in incrementation of the version field or property of the entity.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ExcludedFromVersioning {}
```

This annotation indicates that the application tolerates lost updates to the annotated field or property when concurrent optimistic transactions modify the same entity instance, and that no `OptimisticLockException` should occur in this scenario.

11.2.22. ForeignKey Annotation

The `ForeignKey` annotation is used to specify the handling of foreign key constraints when schema generation is in effect. If this annotation is not specified, the persistence provider's default foreign key strategy is used.

```
@Target({})
@Retention(RUNTIME)
public @interface ForeignKey {
    String name() default "";
    ConstraintMode value() default CONSTRAINT;
    String foreignKeyDefinition() default "";
    String options() default "";
}
```

The `name` element specifies a name for the foreign key constraint.

The `ConstraintMode` enum is used to control the application of constraints.

```
public enum ConstraintMode {CONSTRAINT, NO_CONSTRAINT, PROVIDER_DEFAULT}
```

The enum values have the following semantics: A value of `CONSTRAINT` will cause the persistence provider to generate a foreign key constraint. A value of `NO_CONSTRAINT` will result in no constraint being generated. A value of `PROVIDER_DEFAULT` will result in the provider's default behavior (which may or may not result in the generation of a constraint for any given join column or set of join columns).

The syntax used in the `foreignKeyDefinition` element should follow the SQL syntax used by the target database for foreign key constraints. For example, this may be similar to the following:

```
FOREIGN KEY (<COLUMN expression> {, <COLUMN expression>}... )
REFERENCES <TABLE identifier> [ (<COLUMN expression> {, <COLUMN expression>}... ) ]
[ ON UPDATE <referential action> ]
[ ON DELETE <referential action> ]
```

If the `ForeignKey` annotation is specified with a `ConstraintMode` value of `CONSTRAINT`, but the `foreignKeyDefinition` element is not specified, the provider will generate a foreign key constraint whose update and delete actions it determines most appropriate for the join column(s) to which the foreign key constraint is applied

Table 20 lists the annotation elements that may be specified for the `ForeignKey` annotation.

Table 20. ForeignKey Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the foreign key constraint.	A provider-generated name.
ConstraintMode	value	(Optional) Whether to generate a constraint.	CONSTRAINT
String	foreignKeyDefinition	(Optional) The foreign key constraint definition.	Provider-default. If the value of the `ConstraintMode` element is` NO_CONSTRAINT`, the provider must not generate a foreign key constraint.
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended.

11.2.23. GeneratedValue Annotation

The `GeneratedValue` annotation specifies a generation strategy for the values of a primary key. The `GeneratedValue` annotation may be applied to a primary key property or field of an entity or mapped superclass in conjunction with the `Id` annotation.[12] The persistence provider is only required to support the use of the `GeneratedValue` annotation for simple primary keys. Use of the `GeneratedValue` annotation for derived primary keys is not supported.

Table 21 lists the annotation elements that may be specified for the `GeneratedValue` annotation and their default values.

The types of primary key generation are defined by the `GenerationType` enum:

```
public enum GenerationType { TABLE, SEQUENCE, IDENTITY, UUID, AUTO }
```

The `TABLE` generator type value indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.

The `SEQUENCE` and `IDENTITY` values specify the use of a database sequence or identity column, respectively.[13]

The further specification of table generators and sequence generators is described in Section 11.2.51 and Section 11.2.54.

A `TABLE`, `SEQUENCE`, or `IDENTITY` generator may be used to generate values for a primary key property or field of type `java.lang.Long`, `java.lang.Integer`, `long`, or `int`.

The `UUID` value indicates that the persistence provider should assign an RFC 4122 Universally Unique IDentifier.

A `UUID` generator may be used to generate values for a primary key property or field of type `java.util.UUID` or `java.lang.String`.

The `AUTO` value indicates that the persistence provider should pick an appropriate strategy given the type of the primary key property or field, and the capabilities of the particular database. In the case of a field or property of type `java.util.UUID` or `java.lang.String`, the `AUTO` strategy is equivalent to `UUID`. In the case of a field or property of type `java.lang.Long`, `java.lang.Integer`, `long`, or `int`, the `AUTO` strategy may select between `TABLE`, `SEQUENCE`, or `IDENTITY`.

The `AUTO` generation strategy may expect a database resource to exist, or it may attempt to create one. A vendor may provide documentation on how to create such resources in the event that it does not support schema generation or cannot create the schema resource at runtime.

This specification does not define the exact behavior of these strategies.

However, if the persistence provider stores a value generated according to the `UUID` strategy in a column of type `VARCHAR` or equivalent, the value must be stored in its canonical representation, unless the application explicitly indicates that some other representation is preferred.

The `name` member specifies the name of a generator to use, and defaults to the entity name of the entity in which the `GeneratedValue` annotation occurs. If the `name` is not specified, and if there is no generator with the defaulted name, then the persistence provider supplies a default id generator, of a type compatible with the value of the `strategy` member.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface GeneratedValue {
    GenerationType strategy() default AUTO;
    String generator() default "";
}
```

Table 21. GeneratedValue Annotation Elements

Type	Name	Description	Default
GenerationType	strategy	(Optional) The primary key generation strategy that the persistence provider must use to generate the annotated entity primary key.	GenerationType.AUTO
String	generator	(Optional) The name of the primary key generator to use as specified in the `SequenceGenerator` or `TableGenerator` annotation which declares the generator.	The entity name of the entity in which the annotation occurs.

Example 1:

```
@Id
@GeneratedValue(strategy=SEQUENCE, generator="CUST_SEQ")
@Column(name="CUST_ID")
public Long getId() { return id; }
```

Example 2:

```
@Id
@GeneratedValue(strategy=TABLE, generator="CUST_GEN")
@Column(name="CUST_ID")
Long id;
```

11.2.24. Id Annotation

The `Id` annotation declares a primary key property or field of an entity. The `Id` annotation may be applied to a property or field of:

- an entity class that is the root of an entity hierarchy, or

- a mapped superclass that is a superclass of all entity classes in an entity hierarchy.

The field or property to which the `Id` annotation is applied should have one of the legal simple primary key types listed in Section 2.4.[14][15]

The mapped column for the primary key of the entity is assumed to be the primary key of the primary table. If no `Column` annotation is specified, the primary key column name is assumed to be the name of the primary key property or field.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Id {}
```

Example:

```
@Id
public Long getId() { return id; }
```

11.2.25. IdClass Annotation

The `IdClass` annotation specifies a composite primary key class, as defined in Section 2.4.1, for the annotated entity class or mapped superclass. The `IdClass` annotation is used when—and only when—the primary key encompasses multiple fields or properties of the entity, as specified by Section 2.4. If a composite primary key is mapped by an `EmbeddedId`, the `IdClass` annotation must not be applied to the entity class.

When a composite primary key class is specified via `IdClass`:

- The `Id` annotation must be applied to the primary key fields or properties of the entity class, and
- the names of the fields or properties in the primary key class and the primary key fields or properties of the entity must correspond, and their types must match according to the rules specified in Section 2.4 and Section 2.4.2.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface IdClass {
    Class<?> value();
}
```

The `IdClass` annotation has only a `value` member, as listed in Table 22. The specified class must be a legal primary key class, according to the rules specified in Section 2.4.1.

Table 22. IdClass Annotation Elements

Type	Name	Description	Default
Class	value	(Required) The composite primary key class.	

Example:

```
@IdClass(com.acme.EmployeePK.class)
@Entity
public class Employee {
    @Id
    String empName;

    @Id
    Date birthDay;
```

```
    // ...
}
```

11.2.26. Index Annotation

The `Index` annotation is used in schema generation. Note that it is not necessary to specify an index for a primary key, as the primary key index is created automatically, however, the Index annotation may be used to specify the ordering of the columns in the index for the primary key.

```
@Target({})
@Retention(RUNTIME)
public @interface Index {
    String name() default "";
    String columnList();
    boolean unique() default false;
    String type() default "";
    String using() default "";
    String options() default "";
}
```

The syntax of the `columnList` element depends on the SQL dialect, but is usually a comma-separated list of column names, with an optional ordering specification, `ASC` or `DESC`.

Table 23 lists the annotation elements that may be specified for the `Index` annotation.

Table 23. Index Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the index.	A provider-generated name.
String	columnList	(Required) The names of the columns to be included in the index.	
boolean	unique	(Optional) Whether the index is unique.	false
String	type	(Optional) A SQL fragment representing the index type, usually inserted before the keyword `type` in the generated DDL which creates this index.	Nothing inserted
String	using	(Optional) A SQL fragment representing the indexing method. When specified, it is added into the generated DDL which creates this index, usually following a `USING` keyword.	Nothing appended.

Type	Name	Description	Default
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended.

11.2.27. Inheritance Annotation

The `Inheritance` annotation specifies the inheritance mapping strategy, as defined in Section 2.14, for an entity class hierarchy. This annotation, if it is specified, must be applied to the entity class that is the root of the entity class hierarchy.

The inheritance mapping strategies defined by this specification are enumerated by the `InheritanceType` enum:

```
public enum InheritanceType { SINGLE_TABLE, TABLE_PER_CLASS, JOINED }
```

If the root entity of an entity class hierarchy is not annotated with the `Inheritance` annotation, and if no inheritance mapping strategy is specified for an entity class hierarchy by an XML descriptor, the inheritance mapping strategy for the entity class hierarchy defaults to the `SINGLE_TABLE` mapping strategy.

This specification does not require support for mixing inheritance mapping strategies within a single entity class hierarchy. Therefore, a portable application must not apply the `Inheritance` annotation to a non-root entity in an entity class hierarchy.

This specification does not require support for the `TABLE_PER_CLASS` mapping strategy in this release.

Table 24 lists the annotation elements that may be specified for the `Inheritance` annotation and their default values.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Inheritance {
    InheritanceType strategy() default SINGLE_TABLE;
}
```

Table 24. Inheritance Annotation Elements

Type	Name	Description	Default
InheritanceType	strategy	(Optional) The inheritance strategy to use for the entity inheritance hierarchy.	`InheritanceType.SINGLE_TABLE`

Example:

```
@Entity
@Inheritance(strategy=JOINED)
public class Customer { ... }

@Entity
public class ValuedCustomer extends Customer { ... }
```

11.2.28. JoinColumn Annotation

The `JoinColumn` annotation is used to specify a column for joining an entity association or element collection.

Table 25 lists the annotation elements that may be specified for the `JoinColumn` annotation and their default values.

If the `JoinColumn` annotation itself is defaulted, a single join column is assumed and the default values described in Table 25 apply.

The `name` annotation element defines the name of the foreign key column. The remaining annotation elements (other than `referencedColumnName`) refer to this column and have the same semantics as for the `Column` annotation.

If the `referencedColumnName` element is missing, the foreign key is assumed to refer to the primary key of the referenced table.

Support for referenced columns that are not primary key columns of the referenced table is optional. Applications that use such mappings are not portable.

The `foreignKey` annotation element is used to specify or control the generation of a foreign key constraint when schema generation is in effect. If this element is not specified, the persistence provider's default foreign key strategy will apply.

If more than one `JoinColumn` annotation is applied to a field or property, both the `name` and the `referencedColumnName` elements must be specified in each such `JoinColumn` annotation.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(JoinColumns.class)
public @interface JoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String options() default "";
    String table() default "";
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    CheckConstraint[] check() default {};
    String comment() default "";
}
```

Table 25. JoinColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the foreign key column. The table in which it is found depends upon the context. If the join is for a OneToOne or ManyToOne mapping using a foreign key mapping strategy, the foreign key column is in the table of the source entity or embeddable. If the join is for a unidirectional OneToMany mapping using a foreign key mapping strategy, the foreign key is in the table of the target entity. If the join is for a ManyToMany mapping or for a OneToOne or bidirectional ManyToOne/OneToMany mapping using a join table, the foreign key is in a join table. If the join is for an element collection, the foreign key is in a collection table.	(Default only applies if a single join column is used.) The concatenation of the following: the name of the referencing relationship property or field of the referencing entity or embeddable class; "_"; the name of the referenced primary key column. If there is no such referencing relationship property or field in the entity, or if the join is for an element collection, the join column name is formed as the concatenation of the following: the name of the entity; "_"; the name of the referenced primary key column.

Type	Name	Description	Default
String	referencedColumnName	(Optional) The name of the column referenced by this foreign key column. When used with entity relationship mappings other than the cases described below, the referenced column is in the table of the target entity. When used with a unidirectional OneToMany foreign key mapping, the referenced column is in the table of the source entity. When used inside a `JoinTable` annotation, the referenced key column is in the entity table of the owning entity, or inverse entity if the join is part of the inverse join definition. When used in a collection table mapping, the referenced column is in the table of the entity containing the collection.	(Default only applies if single join column is being used.) The same name as the primary key column of the referenced table.
boolean	unique	(Optional) Whether the property is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint is only a single field. It is not necessary to explicitly specify this for a join column that corresponds to a primary key that is part of a foreign key.	false
boolean	nullable	(Optional) Whether the foreign key column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true

Type	Name	Description	Default
boolean	updatable	(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL for the column.
String	table	(Optional) The name of the table that contains the column.	If the join is for a OneToOne or ManyToOne mapping using a foreign key mapping strategy, the name of the table of the source entity or embeddable. If the join is for a unidirectional OneToMany mapping using a foreign key mapping strategy, the name of the table of the target entity. If the join is for a ManyToMany mapping or for a OneToOne or bidirectional ManyToOne/OneToMany mapping using a join table, the name of the join table. If the join is for an element collection, the name of the collection table.
ForeignKey	foreignKey	(Optional) The foreign key constraint for the join column. This is used only if table generation is in effect.	Provider's default
CheckConstraint[]	check	(Optional) Check constraints for the column. These are only used if table generation is in effect.	No check constraint
String	comment	(Optional) Comment for the column. This is only used if table generation is in effect.	No comment
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended.

Example 1:

```
@ManyToOne
```

```
@JoinColumn(name="ADDR_ID")
public Address getAddress() { return address; }
```

Example 2: Unidirectional One-to-Many association using a foreign key mapping.

In Customer class:

```
@OneToMany
@JoinColumn(name="CUST_ID") // join column is in table for Order
public Set<Order> getOrders() { return orders; }
```

11.2.29. JoinColumns Annotation

Composite foreign keys are supported by means of the `JoinColumns` annotation. The `JoinColumns` annotation groups `JoinColumn` annotations for the same relationship.

When the `JoinColumns` annotation is used, both the `name` and the `referencedColumnName` elements must be specified in each of the grouped `JoinColumn` annotations.

The `foreignKey` annotation element is used to specify or control the generation of a foreign key constraint when schema generation is in effect. If both this element and the `foreignKey` element of any of the `JoinColumn` elements referenced by the `value` element are specified, the behavior is undefined. If no `foreignKey` annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinColumns {
    JoinColumn[] value();
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

Table 26 lists the annotation elements that may be specified for the `JoinColumns` annotation.

Table 26. JoinColumns Annotation Elements

Type	Name	Description	Default
JoinColumn[]	value	(Required) The join columns that map the relationship.	
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default

Example:

```
@ManyToOne
@JoinColumns({
    @JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
    @JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
})
public Address getAddress() { return address; }
```

11.2.30. JoinTable Annotation

The `JoinTable` annotation is used in the mapping of entity associations. A `JoinTable` annotation is specified on the owning side of the association. A join table is typically used in the mapping of many-to-many and unidirectional one-to-many associations. It may also be used to map bidirectional many-to-one/one-to-many associations, unidirectional many-to-one relationships, and one-to-one associations (both bidirectional and unidirectional).

Table 27 lists the annotation elements that may be specified for the `JoinTable` annotation and their default values.

If the `JoinTable` annotation is not explicitly specified for the mapping of a many-to-many or unidirectional one-to-many relationship, the default values of the annotation elements apply.

The name of the join table is assumed to be the table names of the associated primary tables concatenated together (owning side first) using an underscore.

The `foreignKey` element specifies or controls the generation of a foreign key constraint for the columns corresponding to the `joinColumns` element when table generation is in effect. If both this element and the `foreignKey` element of any of the `joinColumns` elements are specified, the behavior is undefined. If no `foreignKey` annotation element is specified in either location, the persistence provider's default foreign key strategy will apply. The `inverseForeignKey` element applies to the generation of a foreign key constraint for the columns corresponding to the `inverseJoinColumns` element, and similar rules apply.

When a join table is used in mapping a relationship with an embeddable class on the owning side of the relationship, the containing entity rather than the embeddable class is considered the owner of the relationship.

This annotation may not be applied to a persistent field or property not annotated `@ManyToOne`, `@OneToOne`, `@ManyToMany`, or `@OneToMany`.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinTable {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    JoinColumn[] joinColumns() default {};
    JoinColumn[] inverseJoinColumns() default {};
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    ForeignKey inverseForeignKey() default @ForeignKey(PROVIDER_DEFAULT);
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
    CheckConstraint[] check() default {};
    String comment() default "";
    String type() default "";
    String options() default "";
}
```

Table 27. JoinTable Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the join table.	The concatenated names of the two associated primary entity tables (owning side first), separated by an underscore.

Type	Name	Description	Default
String	catalog	(Optional) The catalog of the table.	Default catalog.
String	schema	(Optional) The schema of the table.	Default schema for user.
JoinColumn[]	joinColumns	(Optional) The foreign key columns of the join table which reference the primary table of the entity owning the association (i.e. the owning side of the association).	The same defaults as for JoinColumn.
JoinColumn[]	inverseJoinColumns	(Optional) The foreign key columns of the join table which reference the primary table of the entity that does not own the association (i.e. the inverse side of the association).	The same defaults as for JoinColumn.
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider’s default.
ForeignKey	inverseForeignKey	(Optional) The foreign key constraint specification for the inverse join columns. This is used only if table generation is in effect.	Provider’s default.
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes
CheckConstraint[]	check	(Optional) Check constraints for the table. These are only used if table generation is in effect.	No check constraint

Type	Name	Description	Default
String	comment	(Optional) Comment for the table. This is only used if table generation is in effect.	No comment
String	type	(Optional) A SQL fragment indicating the type of table.	Nothing inserted
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Example:

```
@JoinTable(
    name="CUST_PHONE",
    joinColumns=@JoinColumn(name="CUST_ID", referencedColumnName="ID"),
    inverseJoinColumns=@JoinColumn(name="PHONE_ID", referencedColumnName="ID")
)
```

11.2.31. Lob Annotation

A `Lob` annotation specifies that a persistent property or field should be persisted as a large object to a database-supported large object type. Portable applications should use the `Lob` annotation when mapping to a database Lob type. The `Lob` annotation may be used in conjunction with the `Basic` annotation or with the `ElementCollection`[16] annotation when the element collection value is of basic type. A Lob may be either a binary or character type. The Lob type is inferred from the type of the persistent field or property and, except for string and character types, defaults to Blob.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Lob {

}
```

Example 1:

```
@Lob
@Basic(fetch=EAGER)
@Column(name="REPORT")
protected String report;
```

Example 2:

```
@Lob @Basic(fetch=LAZY)
@Column(name="EMP_PIC", columnDefinition="BLOB NOT NULL")
protected byte[] pic;
```

11.2.32. ManyToMany Annotation

A `ManyToMany` annotation defines a many-valued association with many-to-many multiplicity. If the collection is defined using generics to specify the element type, the associated target entity class does not need to be specified; otherwise it must be specified.

Every many-to-many association has two sides, the owning side and the non-owning, or inverse, side. If the association

is bidirectional, either side may be designated as the owning side. If the relationship is bidirectional, the non-owning side must use the `mappedBy` element of the `ManyToMany` annotation to specify the relationship field or property of the owning side.

The join table for the relationship, if not defaulted, is specified on the owning side.

The `ManyToMany` annotation may be used within an embeddable class contained within an entity class to specify a relationship to a collection of entities[17]. If the relationship is bidirectional and the entity containing the embeddable class is the owner of the relationship, the non-owning side must use the `mappedBy` element of the `ManyToMany` annotation to specify the relationship field or property of the embeddable class. The dot (".") notation syntax must be used in the `mappedBy` element to indicate the relationship attribute within the embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

Table 28 lists these annotation elements that may be specified for the `ManyToMany` annotation and their default values.

The `cascade` element specifies the set of cascadable operations that are propagated to the associated entity. The operations that are cascadable are defined by the `CascadeType` enum.

When the collection is a `java.util.Map`, the `cascade` element applies to the map `value`.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ManyToMany {
    Class<?> targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
}
```

Table 28. ManyToMany Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association. Required if the target entity type cannot be inferred from the declared type of the annotated field or property, for example, if the declared type is a raw collection type.	The type argument of the collection type when the declared type of the annotated field or property is a non-raw collection type.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.

Type	Name	Description	Default
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The `EAGER` policy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The `LAZY` policy is a hint to the persistence provider runtime.	LAZY
String	mappedBy	The field or property of the target entity that owns the relationship and specifies its mapping to the database. This element is only specified on the inverse (non-owning) side of the association.	

Example 1:

In Customer class:

```
@ManyToMany
@JoinTable(name="CUST_PHONES")
public Set<PhoneNumber> getPhones() { return phones; }
```

In PhoneNumber class:

```
@ManyToMany(mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

Example 2:

In Customer class:

```
@ManyToMany(targetEntity=com.acme.PhoneNumber.class)
public Set getPhones() { return phones; }
```

In PhoneNumber class:

```
@ManyToMany(targetEntity=com.acme.Customer.class, mappedBy="phones")
public Set getCustomers() { return customers; }
```

Example 3:

In Customer class:

```
@ManyToMany
@JoinTable(
    name="CUST_PHONE",
```

```
        joinColumns=@JoinColumn(name="CUST_ID", referencedColumnName="ID"),
        inverseJoinColumns=@JoinColumn(name="PHONE_ID",referencedColumnName="ID")
    )
    public Set<PhoneNumber> getPhones() { return phones; }
```

In PhoneNumberClass:

```
    @ManyToMany(mappedBy="phones")
    public Set<Customer> getCustomers() { return customers; }
```

Example 4:

Embeddable class used by the Employee entity specifies a many-to-many relationship.

```
    @Entity
    public class Employee {
        @Id
        int id;

        @Embedded
        ContactInfo contactInfo;

        // ...
    }

    @Embeddable
    public class ContactInfo {
        @ManyToOne
        Address address; // Unidirectional

        @ManyToMany
        List<PhoneNumber> phoneNumbers; // Bidirectional
    }

    @Entity
    public class PhoneNumber {
        @Id
        int phNumber;

        @ManyToMany(mappedBy="contactInfo.phoneNumbers")
        Collection<Employee> employees;
    }
```

11.2.33. ManyToOne Annotation

The `ManyToOne` annotation defines a single-valued association to another entity class that has many-to-one multiplicity. It is not normally necessary to specify the target entity explicitly since it can usually be inferred from the type of the object being referenced.

The `ManyToOne` annotation may be used within an embeddable class to specify a relationship from the embeddable class to an entity class. If the relationship is bidirectional, the non-owning `OneToMany` entity side must use the `mappedBy` element of the `OneToMany` annotation to specify the relationship field or property of the embeddable field or property on the owning side of the relationship. The dot (“.”) notation syntax must be used in the `mappedBy` element to indicate the relationship attribute within the embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

Table 29 lists the annotation elements that may be specified for the `ManyToOne` annotation and their default values.

```
    @Target({METHOD, FIELD})
    @Retention(RUNTIME)
```

```
public @interface ManyToOne {
    Class<?> targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default DEFAULT;
    boolean optional() default true;
}
```

The operations that can be cascaded are defined by the `CascadeType` enum defined in Section 3.6.1.

If the annotated field or property is also annotated with `jakarta.annotation.Nonnull`, then `optional=false` is implied and the value of the `optional` annotation member is ignored.

Table 29. ManyToOne Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association. Required if the target entity type cannot be inferred from the declared type of the annotated field or property.	The declared type of the annotated field or property.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The `EAGER` policy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The `LAZY` policy is a hint to the persistence provider runtime. The `DEFAULT` policy selects between `EAGER` and `LAZY` depending on the default fetch type of the persistence unit for one-to-one and many-to-one associations.	`DEFAULT`

Type	Name	Description	Default
boolean	optional	(Optional) Whether the association is optional. By default, the association is optional, and the annotated field or property may be set to a null value. When `optional=false`, a non-null value must be assigned to the non-optional field or property before the persistence provider attempts to write the state of entity instance to the database. May be used in schema generation to infer that the mapped foreign key column is not null. Ignored if the field or property is annotated `@jakarta.annotation.Nonnull`.	true

Example 1:

```
@ManyToOne(optional=false)
@JoinColumn(name="CUST_ID", nullable=false, updatable=false)
public Customer getCustomer() { return customer; }
```

Example 2:

```
@Entity
public class Employee {
    @Id
    int id;

    @Embedded
    JobInfo jobInfo;

    // ...
}

@Embeddable
public class JobInfo {
    String jobDescription;

    @ManyToOne
    ProgramManager pm; // Bidirectional
}

@Entity
public class ProgramManager {
    @Id
    int id;

    @OneToMany(mappedBy="jobInfo.pm")
```

```
    Collection<Employee> manages;
}
```

11.2.34. MapKey Annotation

The `MapKey` annotation specifies the map key for associations of type `java.util.Map` when the map key is itself the primary key or a persistent field or property of the entity that is the value of the map.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKey {
    String value() default "";
    @Deprecated String name() default "";
}
```

The `value` or `name` element designates the name of the persistent field or property of the associated entity that is used as the map key. If neither `value` nor `name` element is specified, the primary key of the associated entity is used as the map key. If the primary key is a composite primary key and is mapped as `IdClass`, an instance of the primary key class is used as the key.

If a persistent field or property other than the primary key is used as a map key, it is expected to be unique within the context of the relationship.

The `MapKeyClass` annotation is not used when `MapKey` is specified and vice versa.

Table 30 lists the annotation elements that may be specified for the `MapKey` annotation.

Table 30. MapKey Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the persistent field or property that is used as the map key.	The primary key is used as the map key.

Example 1:

```
@Entity
public class Department {

    // ...

    @OneToMany(mappedBy="department")
    @MapKey // map key is primary key
    public Map<Integer, Employee> getEmployees() { ... }

    // ...
}

@Entity
public class Employee {

    // ...

    @Id public Integer getEmpId() { ... }
    @ManyToOne
    @JoinColumn(name="dept_id")
    public Department getDepartment() { ... }

    // ...
```

```
}
```

Example 2:

```
@Entity
public class Department {
    // ...

    @OneToMany(mappedBy="department")
    @MapKey(name="name")
    public Map<String, Employee> getEmployees() { ... }

    // ...
}

@Entity
public class Employee {
    @Id
    public Integer getEmpId() { ... }

    // ...

    public String getName() { ... }

    // ...

    @ManyToOne
    @JoinColumn(name="dept_id")
    public Department getDepartment() { ... }

    // ...
}
```

11.2.35. MapKeyClass Annotation

The `MapKeyClass` annotation is used to specify the type of the map key for associations of type `java.util.Map`. The map key can be a basic type, an embeddable class, or an entity. If the map is specified using Java generics, the `MapKeyClass` annotation and associated type need not be specified; otherwise they must be specified.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyClass {
    Class value();
}
```

The `MapKeyClass` annotation is used in conjunction with `ElementCollection` or one of the collection-valued relationship annotations (`OneToMany` or `ManyToMany`).

The `MapKey` annotation is not used when `MapKeyClass` is specified and vice versa.

Table 31 lists the annotation elements that may be specified for the `MapKeyClass` annotation.

Table 31. MapKeyClass Annotation Elements

Type	Name	Description	Default
Class	value	(Required) The type of the map key.	

Example 1:

```
@Entity
public class Item {
    @Id
    int id;

    // ...

    @ElementCollection(targetClass=String.class)
    @MapKeyClass(String.class)
    Map images; // map from image name to image filename

    // ...
}
```

Example 2:

```
// MapKeyClass and target type of relationship can be defaulted
@Entity
public class Item {
    @Id
    int id;

    // ...

    @ElementCollection
    Map<String, String> images;

    // ...
}
```

Example 3:

```
@Entity
public class Company {
    @Id
    int id;

    // ...

    @OneToMany(targetEntity=com.example.VicePresident.class)
    @MapKeyClass(com.example.Division.class)
    Map organization;
}
```

Example 4:

```
// MapKeyClass and target type of relationship are defaulted
@Entity
public class Company {
    @Id
    int id;

    // ...

    @OneToMany
    Map<Division, VicePresident> organization;
}
```

11.2.36. MapKeyColumn Annotation

The `MapKeyColumn` annotation is used to specify the mapping for the key column of a map whose map key is a basic type. If the `name` element is not specified, it defaults to the concatenation of the following: the name of the referencing

relationship field or property; “_”; “KEY”.

`MapKeyColumn` is used with owned collections. It is specified on the owning side of a one-to-many or many-to-many relationship or on an element collection—and always on the side of the relationship that references the map. The map key is not visible as part of the state of the entity or embeddable class. Use of the `MapKeyColumn` annotation on the unowned side of a relationship is not portable between persistence providers.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyColumn {
    String name() default "";
    boolean unique() default false;
    boolean nullable() default false;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String options() default "";
    String table() default "";
    int length() default 255;
    int precision() default 0; // decimal precision
    int scale() default 0; // decimal scale
    CheckConstraint[] check() default {};
    String comment() default "";
}
```

If no `MapKeyColumn` annotation is specified, the default values in Table 32 apply.

Table 32. MapKeyColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the map key column. The table in which it is found depends upon the context. If the map key is for an element collection, the map key column is in the collection table for the map value. If the map key is for a ManyToMany entity relationship or for a OneToMany entity relationship using a join table, the map key column is in a join table. If the map key is for a OneToMany entity relationship using a foreign key mapping strategy, the map key column is in the table of the entity that is the value of the map.	The concatenation of the following: the name of the referencing property or field name; " _ "; " `KEY` ".

Type	Name	Description	Default
boolean	unique	(Optional) Whether the column is a unique key. This is a shortcut for the UniqueConstraint annotation at the table level and is useful for when the unique key constraint corresponds to only a single column. This constraint applies in addition to any constraint entailed by primary key mapping and to constraints specified at the table level.	false
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL to create a column of the inferred type.
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Type	Name	Description	Default
String	table	(Optional) The name of the table that contains the column.	If the map key is for an element collection, the name of the collection table for the map value. If the map key is for a OneToMany or ManyToMany entity relationship using a join table, the name of the join table for the map. If the map key is for a OneToMany entity relationship using a foreign key mapping strategy, the name of the primary table of the entity that is the value of the map.
int	length	(Optional) The column length Applies only to columns whose type is parameterized by length, for example, varchar or varbinary types.	255
int	precision	(Optional) The precision of a column of SQL type `decimal` or `numeric`, or of similar database-native type. Applies only to columns of exact numeric type. The default value `0` indicates that a provider-determined precision should be inferred.	0
int	scale	(Optional) The scale of a column of SQL type `decimal` or `numeric`, or of similar database-native type. Applies only to columns of exact numeric type. The default value `0` indicates that a provider-determined scale should be inferred.	0

Type	Name	Description	Default
CheckConstraint[]	check	(Optional) Check constraints for the column. These are only used if table generation is in effect.	No check constraint
String	comment	(Optional) Comment for the column. This is only used if table generation is in effect.	No comment

Example:

```
@Entity
public class Item {
    @Id
    int id;

    // ...

    @ElementCollection
    @MapKeyColumn(name="IMAGE_NAME")
    @Column(name="IMAGE_FILENAME")
    @CollectionTable(name="IMAGE_MAPPING")
    Map<String, String> images; // map from image name to filename

    // ...
}
```

11.2.37. MapKeyEnumerated Annotation

The `MapKeyEnumerated` annotation is used to specify the enum type for a map key whose basic type is an enumerated type.

The `MapKeyEnumerated` annotation can be applied to an element collection or relationship of type `java.util.Map` , in conjunction with the `ElementCollection`, `OneToMany`, or `ManyToMany` annotation. If the map is specified using Java generics, the `MapKeyClass` annotation and associated type need not be specified; otherwise they must be specified.

If the enumerated type is not specified or the `MapKeyEnumerated` annotation is not used, the enumerated type is assumed to be `ORDINAL`.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyEnumerated {
    EnumType value() default ORDINAL;
}
```

Table 33 lists the annotation elements that may be specified for the `MapKeyEnumerated` annotation and their default values. The `EnumType` enum is defined in Section 11.2.19.

Table 33. MapKeyEnumerated Annotation Elements

Type	Name	Description	Default
EnumType	value	(Optional) The type used in mapping an enum type.	ORDINAL

11.2.38. MapKeyJoinColumn Annotation

The `MapKeyJoinColumn` annotation is used to specify a mapping to an entity that is a map key. The map key join column is in the collection table, join table, or table of the target entity that is used to represent the map.

The `MapKeyJoinColumn` is used with owned collections. It is specified on the owning side of a one-to-many or many-to-many relationship or on an element collection—and always on the side of the relationship that references the map. The map key is not visible as part of the state of the entity or embeddable class.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(MapKeyJoinColumns.class)
public @interface MapKeyJoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default false;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String options() default "";
    String table() default "";
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    CheckConstraint[] check() default {};
    String comment() default "";
}
```

Table 34 lists the annotation elements that may be specified for the `MapKeyJoinColumn` annotation and their default values.

If no `MapKeyJoinColumn` annotation is specified, a single join column is assumed and the default values described below (and in Table 34) apply.

The `name` annotation element defines the name of the foreign key column. The remaining annotation elements (other than `referencedColumnName`) refer to this column.

If there is a single map key join column, and if the name annotation member is missing, the map key join column name is formed as the concatenation of the following: the name of the referencing relationship property or field of the referencing entity or embeddable; " _ "; " `KEY` ".

If the `referencedColumnName` element is missing, the foreign key is assumed to refer to the primary key of the referenced table. Support for referenced columns that are not primary key columns of the referenced table is optional. Applications that use such mappings are not portable.

The `foreignKey` element specifies or controls the generation of a foreign key constraint for the map key join column when table generation is in effect. If the `foreignKey` element is not specified, the persistence provider's default foreign key strategy is used.

If more than one `MapKeyJoinColumn` annotation is applied to a field or property, both the `name` and the `referencedColumnName` elements must be specified in each such `MapKeyJoinColumn` annotation.

Table 34. MapKeyJoinColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the foreign key column for the map key. The table in which it is found depends upon the context. If the join is for a map key for an element collection, the foreign key column is in the collection table for the map value. If the join is for a map key for a `ManyToMany` entity relationship or for a `OneToMany` entity relationship using a join table, the foreign key column is in a join table. If the join is for a `OneToMany` entity relationship using a foreign key mapping strategy, the foreign key column for the map key is in the table of the entity that is the value of the map.	(Default only applies if a single join column is used.) The concatenation of the following: the name of the referencing relationship property or field of the referencing entity or embeddable class; “_”; "KEY".
String	referencedColumnName	(Optional) The name of the column referenced by this foreign key column. The referenced column is in the table of the target entity.	(Default only applies if single join column is being used.) The same name as the primary key column of the referenced table.
boolean	unique	(Optional) Whether the property is a unique key. This is a shortcut for the `UniqueConstraint` annotation at the table level and is useful for when the unique key constraint is only a single field.	false
boolean	nullable	(Optional) Whether the foreign key column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true

Type	Name	Description	Default
boolean	updatable	(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL for the column.
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended
String	table	(Optional) The name of the table that contains the foreign key column. If the join is for a map key for an element collection, the foreign key column is in the collection table for the map value. If the join is for a map key for a `ManyToMany` entity relationship or for a `OneToMany` entity relationship using a join table, the foreign key column is in a join table. If the join is for a `OneToMany` entity relationship using a foreign key mapping strategy, the foreign key column for the map key is in the table of the entity that is the value of the map.	If the map is for an element collection, the name of the collection table for the map value. If the map is for a `OneToMany` or `ManyToMany` entity relationship using a join table, the name of the join table for the map. If the map is for a `OneToMany` entity relationship using a foreign key mapping strategy, the name of the primary table of the entity that is the value of the map.
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join column. This is used only if table generation is in effect.	Provider’s default
CheckConstraint[]	check	(Optional) Check constraints for the column. These are only used if table generation is in effect.	No check constraint
String	comment	(Optional) Comment for the column. This is only used if table generation is in effect.	No comment

Example 1:

```
@Entity
public class Company {
    @Id
    int id;

    // ...

    @OneToMany // unidirectional
    @JoinTable(
        name="COMPANY_ORGANIZATION",
        joinColumns=@JoinColumn(name="COMPANY"),
        inverseJoinColumns=@JoinColumn(name="VICEPRESIDENT")
    )
    @MapKeyJoinColumn(name="DIVISION")
    Map<Division, VicePresident> organization;
}
```

Example 2:

```
@Entity
public class VideoStore {
    @Id
    int id;

    String name;

    Address location;

    // ...

    @ElementCollection
    @CollectionTable(name="INVENTORY", joinColumns=@JoinColumn(name="STORE"))
    @Column(name="COPIES_IN_STOCK")
    @MapKeyJoinColumn(name="MOVIE", referencedColumnName="ID")
    Map<Movie, Integer> videoInventory;

    // ...
}

@Entity
public class Movie {
    @Id
    long id;

    String title;

    // ...
}
```

Example 3:

```
@Entity
public class Student {
    @Id
    int studentId;

    // ...

    @ManyToMany // students and courses are also many-many
    @JoinTable(
        name="ENROLLMENTS",
        joinColumns=@JoinColumn(name="STUDENT"),
        inverseJoinColumns=@JoinColumn(name="SEMESTER")
    )
```

```
    @MapKeyJoinColumn(name="COURSE")
    Map<Course, Semester> enrollment;

    // ...
}
```

11.2.39. MapKeyJoinColumns Annotation

Composite map keys referencing entities are supported by means of the `MapKeyJoinColumns` annotation. The `MapKeyJoinColumns` annotation groups `MapKeyJoinColumn` annotations.

When the `MapKeyJoinColumns` annotation is used, both the `name` and the `referencedColumnName` elements must be specified in each of the grouped `MapKeyJoinColumn` annotations.

The `foreignKey` element specifies or controls the generation of a foreign key constraint for the columns corresponding to the `MapKeyJoinColumn` elements referenced by the `value` element when table generation is in effect. If both this element and the `foreignKey` element of any of the `MapKeyJoinColumn` elements are specified, the behavior is undefined. If no `foreignKey` annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyJoinColumns {
    MapKeyJoinColumn[] value();
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

Table 35 lists the annotation elements that may be specified for the `MapKeyJoinColumns` annotation.

Table 35. MapKeyJoinColumns Annotation Elements

Type	Name	Description	Default
MapKeyJoinColumn[]	value	(Required) The map key join columns that are used to map to the entity that is the map key.	
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default

11.2.40. MapKeyTemporal Annotation

The `MapKeyTemporal` annotation is used to specify the temporal type for a map key whose basic type is a temporal type.

The `MapKeyTemporal` annotation can be applied to an element collection or relationship of type `java.util.Map` , in conjunction with the `ElementCollection`, `OneToMany`, or `ManyToMany` annotation. If the map is specified using Java generics, the `MapKeyClass` annotation and associated type need not be specified; otherwise they must be specified.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKeyTemporal {
    TemporalType value();
```

```
}
```

The `MapKeyTemporal` enum is deprecated, and its use in newly written code is strongly discouraged.

Table 36 lists the annotation elements that may be specified for the `MapKeyTemporal` annotation and their default values. The `TemporalType` enum is defined in Section 11.2.56.

Table 36. MapKeyTemporal Annotation Elements

Type	Name	Description	Default
TemporalType	value	(Required) The type used in mapping java.util.Date or java.util.Calendar.	

11.2.41. MappedSuperclass Annotation

The `MappedSuperclass` annotation designates a class whose mapping information is applied to the entities that inherit from it. A mapped superclass has no separate table defined for it.

A class designated with the `MappedSuperclass` annotation can be mapped in the same way as an entity except that the mappings will apply only to its subclasses since no table exists for the mapped superclass itself. When applied to the subclasses the inherited mappings will apply in the context of the subclass tables. Mapping information may be overridden in such subclasses by using the `AttributeOverride`, `AttributeOverrides`, `AssociationOverride`, and `AssociationOverrides` annotations.

```
@Documented
@Target(TYPE)
@Retention(RUNTIME)
public @interface MappedSuperclass {}
```

11.2.42. MapsId Annotation

The `MapsId` annotation designates a `ManyToOne` or `OneToOne` relationship attribute that provides the mapping for

- an `Id` or `EmbeddedId` attribute of an entity with a derived identity, or
- an attribute of the `EmbeddedId` class of an entity with a derived identity,

as defined in Section 2.4.2.1.

The `value` element listed in Table 37 specifies the attribute within a composite key class to which the relationship attribute corresponds. If the primary key of the entity is of the same Java type as the primary key of the entity referenced by the relationship, the `value` attribute is not specified.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapsId {
    String value() default "";
}
```

Table 37. MapsId Annotation Elements

Type	Name	Description	Default
String	value	(Optional) The name of an attribute within the composite key class to which the relationship attribute corresponds.	The relationship maps the primary key of the entity.

Example:

```
// parent entity has simple primary key
@Entity
public class Employee {
    @Id
    long empId;

    String name;

    // ...
}

// dependent entity uses EmbeddedId for composite key
@Embeddable
public class PaycheckId {
    int period;
    long empid; // corresponds to PK type of Employee
}

@Entity
public class Paycheck {
    @EmbeddedId
    PaycheckId id;

    // ...

    @MapsId("empid") // maps the empid attribute of embedded id
    @ManyToOne
    Employee emp;
}
```

11.2.43. OneToMany Annotation

A `OneToMany` annotation defines a many-valued association with one-to-many multiplicity.

Table 38 lists the annotation elements that may be specified for the `OneToMany` annotation and their default values.

If the collection is defined using generics to specify the element type, the associated target entity class need not be specified; otherwise it must be specified.

The `OneToMany` annotation may be used within an embeddable class contained within an entity class to specify a relationship to a collection of entities[18]. If the relationship is bidirectional, the `mappedBy` element must be used to specify the relationship field or property of the entity that is the owner of the relationship.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToMany {
    Class<?> targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
```

```
    boolean orphanRemoval() default false;
}
```

The operations that can be cascaded are defined by the `CascadeType` enum defined in Section 3.6.1.

When the collection is a `java.util.Map`, the `cascade` element and the `orphanRemoval` element apply to the map `value`.

If `orphanRemoval` is `true` and an entity that is the target of the relationship is removed from the relationship (either by removal from the collection or by setting the relationship to null), the remove operation is applied to the entity being orphaned. If the entity being orphaned is a detached, new, or removed entity, the semantics of `orphanRemoval` do not apply.

If `orphanRemoval` is `true` and the remove operation is applied to the source entity, the remove operation is cascaded to the relationship target in accordance with the rules of Section 3.6.4, (and hence it is not necessary to specify `cascade=REMOVE` for the relationship)[19].

The remove operation is applied at the time of the flush operation. The `orphanRemoval` functionality is intended for entities that are privately "owned" by their parent entity. Portable applications must otherwise not depend upon a specific order of removal, and must not reassign an entity that has been orphaned to another relationship or otherwise attempt to persist it.

The default mapping for unidirectional one-to-many relationships uses a join table as is described in Section 2.12.5. Unidirectional one-to-many relationships may be implemented using one-to-many foreign key mappings, using the `JoinColumn` and `JoinColumns` annotations.

Table 38. OneToMany Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association. Required if the target entity type cannot be inferred from the declared type of the annotated field or property, for example, if the declared type is a raw collection type.	The type argument of the collection type when the declared type of the annotated field or property is a non-raw collection type.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The `EAGER` policy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The `LAZY` policy is a hint to the persistence provider runtime.	LAZY

Type	Name	Description	Default
String	mappedBy	The field or property of the target entity that owns the relationship and specifies its mapping to the database. Required unless the relationship is unidirectional.	
boolean	orphanRemoval	(Optional) Whether to apply the remove operation to entities that have been removed from the relationship and to cascade the remove operation to those entities.	false

Example 1: One-to-Many association using generics

In Customer class:

```
@OneToMany(cascade=ALL, mappedBy="customer", orphanRemoval=true)
public Set<Order> getOrders() { return orders; }
```

In Order class:

```
@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
public Customer getCustomer() { return customer; }
```

Example 2: One-to-Many association without using generics

In Customer class:

```
@OneToMany(
    targetEntity=com.acme.Order.class,
    cascade=ALL,
    mappedBy="customer",
    orphanRemoval=true
)
public Set getOrders() { return orders; }
```

In Order class:

```
@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
protected Customer customer;
```

Example 3: Unidirectional One-to-Many association using a foreign key mapping

In Customer class:

```
@OneToMany(orphanRemoval=true)
@JoinColumn(name="CUST_ID") // join column is in table for Order
public Set<Order> getOrders() { return orders; }
```

11.2.44. OneToOne Annotation

The `OneToOne` annotation defines a single-valued association to another entity that has one-to-one multiplicity. It is not normally necessary to specify the associated target entity explicitly since it can usually be inferred from the type of the object being referenced.

If the relationship is bidirectional, the `mappedBy` element must be used to specify the relationship field or property of the entity that is the owner of the relationship.

The `OneToOne` annotation may be used within an embeddable class to specify a relationship from the embeddable class to an entity class. If the relationship is bidirectional and the entity containing the embeddable class is on the owning side of the relationship, the non-owning side must use the `mappedBy` element of the `OneToOne` annotation to specify the relationship field or property of the embeddable class. The dot (“.”) notation syntax must be used in the `mappedBy` element to indicate the relationship attribute within the embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

Table 39 lists the annotation elements that may be specified for the `OneToOne` annotation and their default values.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToOne {
    Class<?> targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default DEFAULT;
    boolean optional() default true;
    String mappedBy() default "";
    boolean orphanRemoval() default false;
}
```

The operations that can be cascaded are defined by the `CascadeType` enum, defined in Section 3.6.1.

If `orphanRemoval` is `true` and an entity that is the target of the relationship is removed from the relationship (by setting the relationship to null), the remove operation is applied to the entity being orphaned. If the entity being orphaned is a detached, new, or removed entity, the semantics of `orphanRemoval` do not apply.

If `orphanRemoval` is `true` and the remove operation is applied to the source entity, the remove operation is cascaded to the relationship target in accordance with the rules of Section 3.6.4, (and hence it is not necessary to specify `cascade=REMOVE` for the relationship)[20].

The remove operation is applied at the time of the flush operation. The `orphanRemoval` functionality is intended for entities that are privately "owned" by their parent entity. Portable applications must otherwise not depend upon a specific order of removal, and must not reassign an entity that has been orphaned to another relationship or otherwise attempt to persist it.

If the annotated field or property is also annotated with `jakarta.annotation.Nonnull`, then `optional=false` is implied and the value of the `optional` annotation member is ignored.

Table 39. OneToOne Annotation Elements

Type	Name	Description	Default
Class	targetEntity	(Optional) The entity class that is the target of the association. Required if the target entity type cannot be inferred from the declared type of the annotated field or property.	The declared type of the annotated field or property.
CascadeType[]	cascade	(Optional) The operations that must be cascaded to the target of the association.	No operations are cascaded.
FetchType	fetch	(Optional) Whether the association should be lazily loaded or must be eagerly fetched. The `EAGER` policy is a requirement on the persistence provider runtime that the associated entity must be eagerly fetched. The `LAZY` policy is a hint to the persistence provider runtime. The `DEFAULT` policy selects between `EAGER` and `LAZY` depending on the default fetch type of the persistence unit for one-to-one and many-to-one associations.	`DEFAULT`

Type	Name	Description	Default
boolean	optional	(Optional) Whether the association is optional. By default, the association is optional, and the annotated field or property may be set to a null value. When `optional=false`, a non-null value must be assigned to the non-optional field or property before the persistence provider attempts to write the state of entity instance to the database. May be used in schema generation to infer that the mapped foreign key column is not null. Ignored if the field or property is annotated `@jakarta.annotation.Nonnull`.	true
String	mappedBy	The field or property of the target entity that owns the relationship and specifies its mapping to the database. This element is only specified on the inverse (non-owning) side of the association.	
boolean	orphanRemoval	(Optional) Whether to apply the remove operation to entities that have been removed from the relationship and to cascade the remove operation to those entities.	false

Example 1: One-to-one association that maps a foreign key column.

On Customer class:

```
@OneToOne(optional=false)
@JoinColumn(name="CUSTREC_ID", unique=true, nullable=false, updatable=false)
public CustomerRecord getCustomerRecord() { return customerRecord; }
```

On CustomerRecord class:

```
@OneToOne(optional=false, mappedBy="customerRecord")
```

```
    public Customer getCustomer() { return customer; }
```

Example 2: One-to-one association where both source and target share the same primary key values.

On Employee class:

```
@Entity
public class Employee {
    @Id
    Integer id;

    @OneToOne(orphanRemoval=true)
    @MapsId
    EmployeeInfo info;

    // ...
}
```

On EmployeeInfo class:

```
@Entity
public class EmployeeInfo {
    @Id
    Integer id;

    //...
}
```

Example 3: One-to-one association from an embeddable class to another entity.

```
@Entity
public class Employee {
    @Id
    int id;

    @Embedded
    LocationDetails location;

    // ...
}

@Embeddable
public class LocationDetails {
    int officeNumber;

    @OneToOne
    ParkingSpot parkingSpot;

    // ...
}

@Entity
public class ParkingSpot {
    @Id
    int id;

    String garage;

    @OneToOne(mappedBy="location.parkingSpot")
    Employee assignedTo;

    // ...
}
```

11.2.45. OrderBy Annotation

The `OrderBy` annotation specifies the ordering the elements of a collection-valued association or element collection are to have when the association or collection is retrieved.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OrderBy {
    String value() default "";
}
```

The syntax of the `value` ordering element is an `orderby_list`, as follows:

```
orderby_list ::= orderby_item [,orderby_item]*
orderby_item ::= [property_or_field_name] [ASC | DESC]
```

If `orderby_list` is not specified or if `ASC` or `DESC` is not specified, `ASC` (ascending order) is assumed.

If the ordering element is not specified for an entity association, ordering by the primary key of the associated entity is assumed.[21]

A property or field name specified as an `orderby_item` must correspond to a basic persistent property or field of the associated class or embedded class within it. The properties or fields used in the ordering must correspond to columns for which comparison operators are supported.

The dot (".") notation is used to refer to an attribute within an embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

The `OrderBy` annotation may be applied to an element collection. When `OrderBy` is applied to an element collection of basic type, the ordering is by value of the basic objects and the `property_or_field_name` is not used.[22] When specifying an ordering over an element collection of embeddable type, the dot notation must be used to specify the attribute or attributes that determine the ordering.

The `OrderBy` annotation is not used when an order column is specified. See Section 11.2.46.

Table 40 lists the annotation elements that may be specified for the `OrderBy` annotation.

Table 40. OrderBy Annotation Elements

Type	Name	Description	Default
String	value	(Optional) The list of attributes (optionally qualified with ASC or DESC) whose values are used in the ordering.	Ascending ordering by the primary key.

Example 1:

```
@Entity
public class Course {
    // ...

    @ManyToMany
    @OrderBy("lastname ASC")
    public List<Student> getStudents() { ... }

    // ...
```

```
}
```

Example 2:

```
@Entity
public class Student {
    // ...

    @ManyToMany(mappedBy="students")
    @OrderBy // PK is assumed
    public List<Course> getCourses() { ... }

    // ...
}
```

Example 3:

```
@Entity
public class Person {
    // ...

    @ElementCollection
    @OrderBy("zipcode.zip, zipcode.plusFour")
    public Set<Address> getResidences() { ... }

    // ...
}

@Embeddable
public class Address {
    protected String street;
    protected String city;
    protected String state;

    @Embedded
    protected Zipcode zipcode;
}

@Embeddable
public class Zipcode {
    protected String zip;
    protected String plusFour;
}
```

11.2.46. OrderColumn Annotation

The `OrderColumn` annotation specifies a column that is used to maintain the persistent order of a list. The persistence provider is responsible for maintaining the order upon retrieval and in the database. The persistence provider is responsible for updating the ordering upon flushing to the database to reflect any insertion, deletion, or reordering affecting the list.

`OrderColumn` is used with owned collections. It is specified on the owning side of a one-to-many or many-to-many relationship or on an element collection—and always on the side of the relationship that references the collection that is ordered. The order column is not visible as part of the state of the entity or embeddable class. [23] Use of the `OrderColumn` annotation on the unowned side of a relationship is not portable between persistence providers.

The `OrderBy` annotation is not used when `OrderColumn` is specified.

Table 41 lists the annotation elements that may be specified for the `OrderColumn` annotation and their default values.

```
@Target({METHOD, FIELD})
```

```
@Retention(RUNTIME)
public @interface OrderColumn {
    String name() default "";
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String options() default "";
    String comment() default "";
}
```

If `name` is not specified, the column name is the concatenation of the following: the name of the referencing relationship property or field of the referencing entity or embeddable class; " _ "; " `ORDER` ".

The order column must be of integral type. The persistence provider must maintain a contiguous (non-sparse) ordering of the values of the order column when updating the association or element collection. The order column value for the first element of the list must be 0.

Table 41. OrderColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the ordering column.	The concatenation of the name of the referencing property or field; " _ "; " `ORDER` ".
boolean	nullable	(Optional) Whether the database column is nullable.	true
boolean	insertable	(Optional) Whether the column is included in SQL INSERT statements generated by the persistence provider.	true
boolean	updatable	(Optional) Whether the column is included in SQL UPDATE statements generated by the persistence provider.	true
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column.	Generated SQL to create a column of the inferred type.
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Example 1:

```
@Entity
public class CreditCard {
    @Id
    long ccNumber;
```

```
    @OneToMany // unidirectional
    @OrderColumn
    List<CardTransaction> transactionHistory;

    // ...
}
```

Example 2:

```
@Entity
public class Course {
    // ...

    @ManyToMany
    @JoinTable(name="COURSE_ENROLLMENT")
    public Set<Student> getStudents() { ... }

    // ...

    @ManyToMany // unidirectional
    @JoinTable(name="WAIT_LIST")
    @OrderColumn(name="WAITLIST_ORDER")
    public List<Student> getWaitList()  { ... }
}

@Entity
public class Student {
    // ...

    @ManyToMany(mappedBy="students")
    public Set<Course> getCourses() { ... }

    // ...
}
```

Example of querying the ordered list:

```
SELECT w
FROM course c JOIN c.waitlist w
WHERE c.name = "geometry" AND INDEX(w) = 0
```

11.2.47. PrimaryKeyJoinColumn Annotation

The `PrimaryKeyJoinColumn` annotation specifies a primary key column that is used as a foreign key to join to another table.

The `PrimaryKeyJoinColumn` annotation is used to join the primary table of an entity subclass in the `JOINED` mapping strategy to the primary table of its superclass; it is used within a `SecondaryTable` annotation to join a secondary table to a primary table; and it may be used in a `OneToOne` mapping in which the primary key of the referencing entity is used as a foreign key[24] to the referenced entity[25].

The `foreignKey` element specifies or controls the generation of a foreign key constraint for the primary key join column when table generation is in effect. If the `foreignKey` element is not specified, the persistence provider's default foreign key strategy will apply.

Table 42 lists the annotation elements that may be specified for the `PrimaryKeyJoinColumn` annotation and their default values.

If no `PrimaryKeyJoinColumn` annotation is specified for a subclass in the `JOINED` mapping strategy, the foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(PrimaryKeyJoinColumns.class)
public @interface PrimaryKeyJoinColumn {
    String name() default "";
    String referencedColumnName() default "";
    String columnDefinition() default "";
    String options() default "";
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    CheckConstraint[] check() default {};
    String comment() default "";
}
```

Table 42. PrimaryKeyJoinColumn Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the primary key column of the current table.	The same name as the primary key column of the primary table of the superclass (`JOINED` mapping strategy); the same name as the primary key column of the primary table (`SecondaryTable` mapping); or the same name as the primary key column for the table for the referencing entity (`OneToOne` mapping).
String	referencedColumnName	(Optional) The name of the primary key column of the table being joined to.	The same name as the primary key column of the primary table of the superclass (`JOINED` mapping strategy); the same name as the primary key column of the primary table (`SecondaryTable` mapping); or the same name as the primary key column of the table for the referenced entity (`OneToOne` mapping).
String	columnDefinition	(Optional) The SQL fragment that is used when generating the DDL for the column. This should not be specified for a OneToOne primary key association.	Generated SQL to create a column of the inferred type.
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Type	Name	Description	Default
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join column. This is used only if table generation is in effect.	Provider's default
CheckConstraint[]	check	(Optional) Check constraints for the column. These are only used if table generation is in effect.	No check constraint
String	comment	(Optional) Comment for the column. This is only used if table generation is in effect.	No comment

Example: Customer and ValuedCustomer subclass

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=JOINED)
@DiscriminatorValue("CUST")
public class Customer { ... }

@Entity
@Table(name="VCUST")
@DiscriminatorValue("VCUST")
@PrimaryKeyJoinColumn(name="CUST_ID")
public class ValuedCustomer extends Customer { ... }
```

11.2.48. PrimaryKeyJoinColumns Annotation

Composite foreign keys are supported by means of the `PrimaryKeyJoinColumns` annotation. The `PrimaryKeyJoinColumns` annotation groups `PrimaryKeyJoinColumn` annotations.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PrimaryKeyJoinColumns {
    PrimaryKeyJoinColumn[] value();
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
}
```

The `foreignKey` element specifies or controls the generation of a foreign key constraint for the columns corresponding to the `PrimaryKeyJoinColumn` elements referenced by the `value` element when table generation is in effect. If both this element and the `foreignKey` element of any of the `PrimaryKeyJoinColumn` elements are specified, the behavior is undefined. If no `foreignKey` annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

Table 43 lists the annotation elements that may be specified for the `PrimaryKeyJoinColumns` annotation.

Table 43. PrimaryKeyJoinColumns Annotation Elements

Type	Name	Description	Default
PrimaryKeyJoinColumn[]	value	(Required) The primary key join columns.	
ForeignKey	foreignKey	(Optional) The foreign key constraint specification for the join columns. This is used only if table generation is in effect.	Provider's default

Example 1: ValuedCustomer subclass

```
@Entity
@Table(name="VCUST")
@DiscriminatorValue("VCUST")
@PrimaryKeyJoinColumns({
    @PrimaryKeyJoinColumn(name="CUST_ID", referencedColumnName="ID"),
    @PrimaryKeyJoinColumn(name="CUST_TYPE", referencedColumnName="TYPE")
})
public class ValuedCustomer extends Customer { ... }
```

Example 2: OneToOne relationship between Employee and EmployeeInfo classes.[26]

```
public class EmpPK {
    public Integer id;
    public String name;
}

@Entity
@IdClass(com.acme.EmpPK.class)
public class Employee {
    @Id
    Integer id;

    @Id
    String name;

    @OneToOne
    @PrimaryKeyJoinColumns({
        @PrimaryKeyJoinColumn(name="ID", referencedColumnName="EMP_ID"),
        @PrimaryKeyJoinColumn(name="NAME", referencedColumnName="EMP_NAME")
    })
    EmployeeInfo info;

    // ...
}

@Entity
@IdClass(com.acme.EmpPK.class)
public class EmployeeInfo {
    @Id
    @Column(name="EMP_ID")
    Integer id;

    @Id
    @Column(name="EMP_NAME")
    String name;

    // ...
}
```

11.2.49. SecondaryTable Annotation

The `SecondaryTable` annotation is used to specify a secondary table for the annotated entity class.

If no `SecondaryTable` annotation is specified, it is assumed that all persistent fields or properties of the entity are mapped to the primary table. Specifying one or more secondary tables indicates that the data for the entity class is stored across multiple tables.

Table 44 lists the annotation elements that may be specified for the `SecondaryTable` annotation and their default values.

If no primary key join columns are specified, the join columns are assumed to reference the primary key columns of the primary table, and have the same names and types as the referenced primary key columns of the primary table.

The `foreignKey` element specifies or controls the generation of a foreign key constraint for the columns corresponding to the `pkJoinColumns` element when table generation is in effect. If both this element and the `foreignKey` element of any of the `pkJoinColumns` elements are specified, the behavior is undefined. If no `foreignKey` annotation element is specified in either location, the persistence provider's default foreign key strategy will apply.

This annotation may not be applied to a class annotated `@MappedSuperclass` or `@Embeddable`.

```
@Target({TYPE})
@Retention(RUNTIME)
@Repeatable(SecondaryTables.class)
public @interface SecondaryTable {
    String name();
    String catalog() default "";
    String schema() default "";
    PrimaryKeyJoinColumn[] pkJoinColumns() default {};
    ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
    CheckConstraint[] check() default {};
    String comment() default "";
    String type() default "";
    String options() default "";
}
```

Table 44. SecondaryTable Annotation Elements

Type	Name	Description	Default
String	name	(Required) The name of the table.	
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user
PrimaryKeyJoinColumn[]	pkJoinColumns	(Optional) The columns that are used to join with the primary table.	Column(s) of the same name(s) as the primary key column(s) in the primary table

Type	Name	Description	Default
ForeignKey	foreignKey	(Optional) The foreign key constraint for the join column. This is used only if table generation is in effect.	Provider's default
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are typically only used if table generation is in effect. These constraints apply in addition to any constraints specified by the Column and JoinColumn annotations and constraints entailed by primary key mappings.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes
CheckConstraint[]	check	(Optional) Check constraints for the table. These are only used if table generation is in effect.	No check constraint
String	comment	(Optional) Comment for the table. This is only used if table generation is in effect.	No comment
String	type	(Optional) A SQL fragment indicating the type of table.	Nothing inserted
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Example 1: Single secondary table with a single primary key column.

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(
    name="CUST_DETAIL",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="CUST_ID")
)
public class Customer { ... }
```

Example 2: Single secondary table with multiple primary key columns.

```
@Entity
@Table(name="CUSTOMER")
@SecondaryTable(
    name="CUST_DETAIL",
```

```
    pkJoinColumns={
        @PrimaryKeyJoinColumn(name="CUST_ID"),
        @PrimaryKeyJoinColumn(name="CUST_TYPE")
    })
public class Customer { ... }
```

11.2.50. SecondaryTables Annotation

The `SecondaryTables` annotation can be used to specify multiple secondary tables for an entity.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface SecondaryTables {
    SecondaryTable[] value();
}
```

Table 45 lists the annotation elements that may be specified for the `SecondaryTables` annotation.

Table 45. SecondaryTables Annotation Elements

Type	Name	Description	Default
SecondaryTable[]	value	(Required) The secondary tables that are used to map the entity class.	

Example 1: Multiple secondary tables assuming primary key columns are named the same in all tables.

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables({
    @SecondaryTable(name="EMP_DETAIL"),
    @SecondaryTable(name="EMP_HIST")
})
public class Employee { ... }
```

Example 2: Multiple secondary tables with differently named primary key columns.

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables({
    @SecondaryTable(
        name="EMP_DETAIL",
        pkJoinColumns=@PrimaryKeyJoinColumn(name="EMPL_ID")),
    @SecondaryTable(
        name="EMP_HIST",
        pkJoinColumns=@PrimaryKeyJoinColumn(name="EMPLOYEE_ID"))
})
public class Employee { ... }
```

11.2.51. SequenceGenerator Annotation

Defines a primary key generator backed by a database sequence. This annotation may be applied to:

- an entity class, as defined in Section 2.1,
- the primary key field or property of an entity class, as defined in Section 2.4, or
- a package descriptor.

Every sequence generator has a `name`. If this annotation occurs on an entity class or primary key attribute of an entity class, and the `name` member is not explicitly specified by the annotation, the name defaults to the name of the annotated entity. The scope of the generator name is global to the persistence unit (across all generator types).

If this annotation occurs on a package descriptor, and the `name` member is not explicitly specified by the annotation, then the annotation defines a recipe for producing a default generator when a `GeneratedValue` annotation of any program element in the annotated package has `strategy=SEQUENCE` and a defaulted generator name. The name of this default generator is the defaulted generator name, and its other properties are determined by the members of the package `SequenceGenerator` annotation.

A named generator may be referenced by the `generator` element of the `GeneratedValue` annotation.

Table 46 lists the annotation elements that may be specified for the `SequenceGenerator` annotation and their default values.

```
@Target({TYPE, METHOD, FIELD, PACKAGE})
@Retention(RUNTIME)
@Repeatable(SequenceGenerators.class)
public @interface SequenceGenerator {
    String name() default "";
    String sequenceName() default "";
    String catalog() default "";
    String schema() default "";
    int initialValue() default 1;
    int allocationSize() default 50;
    String options() default "";
}
```

Table 46. SequenceGenerator Annotation Elements

Type	Name	Description	Default
String	name	(Optional) A unique generator name that can be referenced by one or more classes to be the generator for primary key values.	See text above
String	sequenceName	(Optional) The name of the database sequence object from which to obtain primary key values.	A provider-chosen sequence name
String	catalog	(Optional) The catalog of the sequence generator.	Default catalog
String	schema	(Optional) The schema of the sequence generator.	Default schema for user
int	initialValue	(Optional) The value from which the sequence object is to start generating.	1

Type	Name	Description	Default
int	allocationSize	(Optional) The amount to increment by when allocating sequence numbers from the sequence.	50
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Example:

```
@SequenceGenerator(name="EMP_SEQ", allocationSize=25)
```

11.2.52. SequenceGenerators Annotation

The `SequenceGenerators` annotation can be used to specify multiple sequence generators.

```
@Target({TYPE, METHOD, FIELD, PACKAGE})
@Retention(RUNTIME)
public @interface SequenceGenerators {
    SequenceGenerator[] value();
}
```

Table 47. SequenceGenerators Annotation Elements

Type	Name	Description	Default
SequenceGenerator[]	value	(Required) The sequence generator mappings	

11.2.53. Table Annotation

The `Table` annotation specifies the primary table for the annotated entity. Additional tables may be specified by using the `SecondaryTable` or `SecondaryTables` annotation.[27]

Table 48 lists the annotation elements that may be specified for the `Table` annotation and their default values.

If no `Table` annotation is specified for an entity class, the default values defined in Table 48 apply.

This annotation may not be applied to a class annotated `@MappedSuperclass` or `@Embeddable`.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Table {
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
    CheckConstraint[] check() default {};
    String comment() default "";
    String type() default "";
    String options() default "";
}
```

Table 48. Table Annotation Elements

Type	Name	Description	Default
String	name	(Optional) The name of the table.	Entity name
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect. These constraints apply in addition to any constraints specified by the `Column` and `JoinColumn` annotations and constraints entailed by primary key mappings.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes
CheckConstraint[]	check	(Optional) Check constraints for the table. These are only used if table generation is in effect.	No check constraint
String	comment	(Optional) Comment for the table. This is only used if table generation is in effect.	No comment
String	type	(Optional) A SQL fragment indicating the type of table.	Nothing inserted
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Example:

```
@Entity
@Table(name="CUST", schema="RECORDS")
public class Customer { ... }
```

11.2.54. TableGenerator Annotation

Defines a primary key generator backed by a row of a database table. This annotation may be applied to:

- an entity class, as defined in Section 2.1,
- the primary key field or property of an entity class, as defined in Section 2.4, or
- a package descriptor.

Every table generator has a `name`. If this annotation occurs on an entity class or primary key attribute of an entity class, and the `name` member is not explicitly specified by the annotation, the name defaults to the name of the annotated entity. The scope of the generator name is global to the persistence unit (across all generator types).

If this annotation occurs on a package descriptor, and the `name` member is not explicitly specified by the annotation, then the annotation defines a recipe for producing a default generator when a `GeneratedValue` annotation of any program element in the annotated package has `strategy=TABLE` and a defaulted generator name. The name of this default generator is the defaulted generator name, and its other properties are determined by the members of the package `TableGenerator` annotation.

A named generator may be referenced by the `generator` element of the `GeneratedValue` annotation.

Table 49 lists the annotation elements that may be specified for the `TableGenerator` annotation and their default values.

The `table` element specifies the name of the table that is used by the persistence provider to store generated primary key values for entities. An entity type will typically use its own row in the table for the generation of primary key values. The primary key values are normally positive integers.

```
@Target({TYPE, METHOD, FIELD, PACKAGE})
@Retention(RUNTIME)
@Repeatable(TableGenerators.class)
public @interface TableGenerator {
    String name() default "";
    String table() default "";
    String catalog() default "";
    String schema() default "";
    String pkColumnName() default "";
    String valueColumnName() default "";
    String pkColumnValue() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
    UniqueConstraint[] uniqueConstraints() default {};
    Index[] indexes() default {};
    String options() default "";
}
```

Table 49. TableGenerator Annotation Elements

Type	Name	Description	Default
String	name	(Optional) A unique generator name that can be referenced by one or more classes to be the generator for primary key values.	See text above
String	table	(Optional) Name of table that stores the generated primary key values.	A provider-chosen table name

Type	Name	Description	Default
String	catalog	(Optional) The catalog of the table.	Default catalog
String	schema	(Optional) The schema of the table.	Default schema for user
String	pkColumnName	(Optional) Name of the primary key column in the table.	A provider-chosen name
String	valueColumnName	(Optional) Name of the column that stores the last value generated.	A provider-chosen name
String	pkColumnValue	(Optional) The primary key value in the generator table that distinguishes this set of generated values from others that may be stored in the table.	A provider-chosen value to store in the primary key column of the generator table
int	initialValue	(Optional) The value used to initialize the column that stores the last value generated.	0
int	allocationSize	(Optional) The amount to increment by when allocating numbers from the generator.	50
UniqueConstraint[]	uniqueConstraints	(Optional) Unique constraints that are to be placed on the table. These are only used if table generation is in effect. These constraints apply in addition to primary key constraints.	No additional constraints
Index[]	indexes	(Optional) Indexes for the table. These are only used if table generation is in effect.	No additional indexes
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Example 1:

```
@Entity
public class Employee {
    // ...
```

```
    @TableGenerator(
        name="empGen",
        table="ID_GEN",
        pkColumnName="GEN_KEY",
        valueColumnName="GEN_VALUE",
        pkColumnValue="EMP_ID",
        allocationSize=1)
    @Id
    @GeneratedValue(strategy=TABLE, generator="empGen")
    int id;

    // ...
}
```

Example 2:

```
@Entity
public class Address {
    // ...

    @TableGenerator(
        name="addressGen",
        table="ID_GEN",
        pkColumnName="GEN_KEY",
        valueColumnName="GEN_VALUE",
        pkColumnValue="ADDR_ID")
    @Id
    @GeneratedValue(strategy=TABLE, generator="addressGen")
    int id;

    // ...
}
```

11.2.55. TableGenerators Annotation

The `TableGenerators` annotation can be used to specify multiple table generators.

```
@Target({TYPE, METHOD, FIELD, PACKAGE})
@Retention(RUNTIME)
public @interface TableGenerators {
    TableGenerator[] value();
}
```

Table 50. TableGenerators Annotation Elements

Type	Name	Description	Default
TableGenerator[]	value	(Required) The table generator mappings	

11.2.56. Temporal Annotation

The `Temporal` annotation must be specified for persistent fields or properties of type `java.util.Date` and `java.util.Calendar` unless a converter is being applied. It may only be specified for fields or properties of these types.

The `Temporal` annotation may be used in conjunction with the `Basic` annotation, the `Id` annotation, or the `ElementCollection`[28] annotation (when the element collection value is of such a temporal type).

The `TemporalType` enum defines the mapping for these temporal types.

```
public enum TemporalType {
    DATE, //java.sql.Date
    TIME, //java.sql.Time
    TIMESTAMP //java.sql.Timestamp
}

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Temporal {
    TemporalType value();
}
```

The `Temporal` annotation and `TemporalType` enum are deprecated, and their use in newly written code is strongly discouraged.

Table 51 lists the annotation elements that may be specified for the `Temporal` annotation and their default values.

Table 51. Temporal Annotation Elements

Type	Name	Description	Default
TemporalType	value	(Required) The type used in mapping java.util.Date or java.util.Calendar.	

Example:

```
@Embeddable
public class EmploymentPeriod {
    @Temporal(DATE)
    java.util.Date startDate;

    @Temporal(DATE)
    java.util.Date endDate;

    // ...
}
```

11.2.57. Transient Annotation

The `Transient` annotation specifies that the annotated property or field of an entity class, mapped superclass, or embeddable class is not persistent.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Transient {}
```

Example:

```
@Entity
public class Employee {
    @Id
    int id;

    @Transient
    User currentUser;

    // ...
}
```

11.2.58. UniqueConstraint Annotation

The `UniqueConstraint` annotation is used to specify that a unique constraint is to be included in the generated DDL for a primary or secondary table.

Table 52 lists the annotation elements that may be specified for the `UniqueConstraint` annotation.

```
@Target({})
@Retention(RUNTIME)
public @interface UniqueConstraint {
    String name() default "";
    String[] columnNames();
    String options() default "";
}
```

Table 52. UniqueConstraint Annotation Elements

Type	Name	Description	Default
String	name	(Optional) Constraint name.	A provider-chosen name.
String[]	columnNames	(Required) An array of the column names that make up the constraint.	
String	options	(Optional) A SQL fragment appended to the generated DDL.	Nothing appended

Example:

```
@Entity
@Table(
    name="EMPLOYEE",
    uniqueConstraints=@UniqueConstraint(columnNames={"EMP_ID", "EMP_NAME"})
)
public class Employee { ... }
```

11.2.59. Version Annotation

The `Version` annotation declares the version field or property of an entity class, as defined in Section 2.5. The version is used to ensure integrity when performing the merge operation, and for optimistic concurrency control, as specified in Section 3.9.2.

The `Version` field or property should be mapped to the primary table of the entity; an application which maps the `Version` property to a table other than the primary table is not portable.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Version {}
```

Example:

```
@Version
@Column(name="OPTLOCK")
protected int getVersionNum() { return versionNum; }
```

11.3. Object/Relational Metadata Used in Schema Generation

The following annotations and XML elements define or control the generation of database objects. If schema generation is in effect, the persistence provider must observe the mapping information specified by these annotations and their corresponding XML elements. Unless otherwise specified, all elements of these annotations are observed in the schema generation process.

- `CollectionTable`
- `Column`
- `DiscriminatorColumn`
- `EmbeddedId`
- `Enumerated`, `MapKeyEnumerated`
- `ForeignKey`
- `GeneratedValue`
- `Id`
- `Index`
- `Inheritance`
- `JoinColumn`
- `JoinTable`
- `Lob`
- `MapKeyColumn`
- `MapKeyJoinColumn`
- `OrderColumn`
- `PrimaryKeyJoinColumn`
- `SecondaryTable`
- `SequenceGenerator`
- `Table`
- `TableGenerator`
- `Temporal`, `MapKeyTemporal`
- `UniqueConstraint`
- `Version`

In some cases, these annotations and elements may be specified explicitly, while in other cases they may be implied by the default values of other annotations or elements. For example, by default a table is generated corresponding to an entity and bears the same name as that assigned to the entity (which in turn may have been defaulted from the name of the entity class).

The naming of database objects is determined by the defaulting rules and the explicit names used in annotations and/or XML. The names of database objects must be treated in conformance with the requirements of Section 2.15.

The metadata annotations and corresponding XML elements that result in generated objects are as follows.

11.3.1. Table-level elements

The following annotations (and corresponding XML elements) specify the creation of tables. The rules for their naming, columns, and other properties are defined in the referenced sections of this specification:

11.3.1.1. Table

By default, a table is created for every top-level entity and, by default, includes columns corresponding to the basic and embedded attributes of the entity and the foreign keys to the tables of related entities. These columns include columns that result from the use of mapped superclasses, if any. The `SecondaryTable` annotation, in conjunction with the use of the `table` element of the `Column` and `JoinColumn` annotations, is used to override this mapping to partition the state of an entity across multiple tables.

The mapping of the columns of a table is controlled by the `Column` and `JoinColumn` annotations. When entity state is inherited from a mapped superclass, the `AttributeOverride` and `AssociationOverride` annotations may be used to further control the column-level mapping of inherited state. The ordering of the columns is not defined by this specification. When it is desirable to control the ordering of columns, DDL scripts should be provided.

See Section 11.2.52 for additional rules that apply to the generation of tables. For the treatment of column-level mappings, see further below.

11.3.1.2. Inheritance

The `Inheritance` annotation defines the inheritance strategy for an entity hierarchy. The inheritance strategy determines whether the table for a top-level entity includes columns for entities that inherit from the entity and whether it includes a discriminator column, or whether separate tables are created for each entity type that inherits from the top-level entity. See Section 2.14 and Section 11.2.27 for rules pertaining to the treatment of entity inheritance.

11.3.1.3. SecondaryTable

A secondary table is created to partition the mapping of entity state across multiple tables. See Section 11.2.49 for the rules that apply to the generation of secondary tables.

11.3.1.4. CollectionTable

A collection table is created for the mapping of an element collection. See Section 11.2.9 for the rules that apply to the generation of collection tables. The `Column`, `AttributeOverride`, and `AssociationOverride` annotations may be used to override `CollectionTable` mappings, as described in Section 11.2.10, Section 11.2.4, and Section 11.2.2 respectively.

11.3.1.5. JoinTable

By default, join tables are created for the mapping of many-to-many relationships and unidirectional one-to-many relationships. See Section 2.12.4, Section 2.12.5.1, and Section 2.12.5.2 for the defaults that apply in such cases. Join tables may also be used to map bidirectional many-to-one/one-to-many associations, unidirectional many-to-one relationships, and one-to-one relationships (both bidirectional and unidirectional). See Section 11.2.30 for the rules that apply to the generation of join tables. The `AssociationOverride` annotation may be used to override join table mappings.

11.3.1.6. TableGenerator

Table generator tables are used to store generated primary key values. See Section 11.2.54 for the rules pertaining to table generators.

11.3.2. Column-level elements

The following annotations and corresponding XML elements control the mapping of columns in generated tables.

The exact mapping of Java language types to database-specific types is not defined by this specification, as databases

vary in the specific types that they support. In general, however, an implementation of this specification should conform to the “Standard Mapping from Java Types to JDBC Types” as defined by the JDBC specification [3]. Unless otherwise explicitly specified, however, `VARCHAR` and `VARBINARY` mappings should be used in preference to `CHAR` and `BINARY` mappings. Applications that are sensitive to the exact database mappings that are generated should use the `columnDefinition` element of the `Column` annotation or include DDL files that specify how the database schema is to be generated.

11.3.2.1. Column

The following elements of the `Column` annotation are used in schema generation:

- `name`
- `unique`
- `nullable`
- `columnDefinition`
- `table`
- `length` (string-valued columns only)
- `precision` (exact numeric (decimal/numeric) columns only)
- `scale` (exact numeric (decimal/numeric) columns only)

See Section 11.2.10 for the rules that apply to these elements and column creation. The `AttributeOverride` annotation may be used to override column mappings.

11.3.2.2. MapKeyColumn

The `MapKeyColumn` annotation specifies the mapping for a key column of a map when the key is of basic type. The following elements of the `MapKeyColumn` annotation are used in schema generation:

- `name`
- `unique`
- `nullable`
- `columnDefinition`
- `table`
- `length` (string-valued columns only)
- `precision` (exact numeric (decimal/numeric) columns only)
- `scale` (exact numeric (decimal/numeric) columns only)

See Section 11.2.36 for the rules that apply to these elements and map key column creation. The `AttributeOverride` annotation may be used to override map key column mappings.

11.3.2.3. Enumerated, MapKeyEnumerated

The `Enumerated` and `MapKeyEnumerated` annotations control whether string- or integer-valued columns are generated for basic attributes of enumerated types and therefore impact the default column mappings for these types. See Section 11.2.19 and Section 11.2.37. The `Column` and `MapKeyColumn` annotations may be used to further control the column mappings for attributes of enumerated types.

11.3.2.4. Temporal, MapKeyTemporal

The `Temporal` and `MapKeyTemporal` annotations control whether date-, time-, or timestamp-value columns are generated for basic attributes of temporal types, and therefore impact the default column mappings for these types. See Section 11.2.56 and Section 11.2.40. The `Column` and `MapKeyColumn` annotations may be used to further control the column mappings for attributes of temporal types.

11.3.2.5. Lob

The `Lob` annotation specifies that a persistent attribute is to be persisted to a database large object type. See Section 11.2.31. In general, however, the treatment of the `Lob` annotation is provider-dependent. Applications that are sensitive to the exact mapping that is used should use the `columnDefinition` element of the `Column` annotation or include DDL files that specify how the database schema is to be generated.

11.3.2.6. OrderColumn

The `OrderColumn` annotation specifies the generation of a column that is used to maintain the persistent ordering of a list that is represented in an element collection, one-to-many, or many-to-many relationship.

The following elements of the `OrderColumn` annotation are used in schema generation:

- `name`
- `nullable`
- `columnDefinition`

See Section 11.2.46 for the rules that pertain to the generation of order columns.

11.3.2.7. DiscriminatorColumn

A discriminator column is generated for the `SINGLE_TABLE` mapping strategy and may optionally be generated by the provider for use with the `JOINED` inheritance strategy. The `DiscriminatorColumn` annotation may be used to control the mapping of the discriminator column. See Section 11.2.13 for the rules that pertain to discriminator columns.

11.3.2.8. Version

The `Version` annotation specifies the generation of a column to serve as an entity's optimistic lock. See Section 11.2.59 for rules that pertain to the version column. The `Column` annotation may be used to further control the column mapping for a version attribute.

11.3.3. Primary Key mappings

Primary keys may be represented by basic or embedded attributes and/or may correspond to foreign key attributes. The `Id` and `EmbeddedId` annotations define attributes whose corresponding columns are the constituents of database primary keys.

11.3.3.1. Id

The `Id` annotation (which may be used in conjunction with the `IdClass` annotation) declares attributes whose database columns correspond to a primary key. Use of the `Id` annotation results in the creation of a primary key consisting of the corresponding column or columns. Rules for the `Id` annotation are described in Section 11.2.24 and Section 2.4.

The `Column` annotation may be used to further control the column mapping for an `Id` attribute that is applied to a basic

type. If the `Id` column was defined in a mapped superclass, the `AttributeOverride` annotation may be used to control the column mapping.

The `JoinColumn` annotation may be used to further control the column mappings for an `Id` attribute that is applied to a relationship that corresponds to a foreign key. If the `Id` attribute was defined in a mapped superclass, the `AssociationOverride` annotation may be used to control the column mapping.

11.3.3.2. EmbeddedId

The `EmbeddedId` annotation specifies an embedded attribute whose corresponding columns correspond to a database primary key. Use of the `EmbeddedId` annotation results in the creation of a primary key consisting of the corresponding columns. Rules for the `EmbeddedId` annotation are described in Section 11.2.18 and Section 2.4.

The `Column` annotation may be used to control the column mapping for an embeddable class. If the `EmbeddedId` attribute is defined in a mapped superclass, the `AttributeOverride` annotation may be used to control the column mappings.

If an `EmbeddedId` attribute corresponds to a relationship attribute, the `MapsId` annotation must be used, and the column mapping is determined by the join column for the relationship. See Section 2.4.2.

11.3.3.3. GeneratedValue

The `GeneratedValue` annotation indicates a primary key whose value is to be generated by the provider. If a strategy is indicated, the provider must use it if it is supported by the target database. Note that specification of the `AUTO` strategy may result in the provider creating a database object for Id generation (e.g., a database sequence). Rules for the `GeneratedValue` annotation are described in Section 11.2.23. The `GeneratedValue` annotation may only be portably used for simple (i.e., non-composite) primary keys.

11.3.4. Foreign Key Column Mappings

11.3.4.1. JoinColumn

The `JoinColumn` annotation is typically used in specifying a foreign key mapping. In general, the foreign key definitions created are provider-dependent and database-dependent. Applications that are sensitive to the exact mapping that is used should use the `foreignKey` element of the `JoinColumn` annotation or include DDL files that specify how the database schemas are to be generated.

The following elements of the `JoinColumn` annotation are used in schema generation:

- `name`
- `referencedColumnName`
- `unique`
- `nullable`
- `columnDefinition`
- `table`
- `foreignKey`

See Section 11.2.28 for rules that apply to these elements and join column creation, and sections Section 2.12 and Section 11.2.9 for the rules that apply for the default mappings of foreign keys for relationships and element collections. The `AssociationOverride` annotation may be used to override relationship mappings. The `PrimaryKeyJoinColumn` annotation is used to join secondary tables and may be used in the mapping of one-to-one relationships. See Section 11.3.4.3 below.

11.3.4.2. MapKeyJoinColumn

The `MapKeyJoinColumn` annotation is to specify foreign key mappings to entities that are map keys in map-valued element collections or relationships. In general, the foreign key definitions created should be expected to be provider-dependent and database-dependent. Applications that are sensitive to the exact mapping that is used should use the `foreignKey` element of the `MapKeyJoinColumn` annotation or include DDL files that specify how the database schemas are to be generated.

The following elements of the `MapKeyJoinColumn` annotation are used in schema generation:

- `name`
- `referencedColumnName`
- `unique`
- `nullable`
- `columnDefinition`
- `table`
- `foreignKey`

See Section 11.2.38 for rules that apply to these elements and map key join column creation. The `AssociationOverride` annotation may be used to override such mappings.

11.3.4.3. PrimaryKeyJoinColumn

The `PrimaryKeyJoinColumn` annotation specifies that a primary key column is to be used as a foreign key. This annotation is used in the specification of the `JOINED` mapping strategy and for joining a secondary table to a primary table in a `OneToOne` relationship mapping. In general, the foreign key definitions created should be expected to be provider-dependent and database-dependent. Applications that are sensitive to the exact mapping that is used should use the `foreignKey` element of the `PrimaryKeyJoinColumn` annotation or include DDL files that specify how the database schemas are to be generated. See Section 11.2.47 for rules pertaining to the `PrimaryKeyJoinColumn` annotation.

11.3.4.4. ForeignKey

The `ForeignKey` annotation may be used within the `JoinColumn`, `JoinColumns`, `MapKeyJoinColumn`, `MapKeyJoinColumns`, `PrimaryKeyJoinColumn`, `PrimaryKeyJoinColumns`, `CollectionTable`, `JoinTable`, `SecondaryTable`, and `AssociationOverride` annotations to specify or override a foreign key constraint. See Section 11.2.22.

11.3.5. Other Elements

11.3.5.1. SequenceGenerator

The `SequenceGenerator` annotation creates a database sequence to be used for Id generation. The use of generators is limited to those databases that support them. See Section 11.2.51.

11.3.5.2. Index

The `Index` annotation generates an index consisting of the specified columns. The ordering of the names in the `columnList` element specified in the `Index` annotation must be observed by the provider when creating the index. See Section 11.2.26.

11.3.5.3. UniqueConstraint

The `UniqueConstraint` annotation generates a unique constraint for the given table. Databases typically implement unique constraints by creating unique indexes. The ordering of the `columnNames` specified in the `UniqueConstraint` annotation must be observed by the provider when creating the constraint. See Section 11.2.58. The `unique` element of the `Column`, `JoinColumn`, `MapKeyColumn`, and `MapKeyJoinColumn` annotations is equivalent to the use of the `UniqueConstraint` annotation when only one column is to be included in the constraint.

11.4. Examples of the Application of Annotations for Object/Relational Mapping

This example shows some simple mappings:

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = AUTO)
    Long id;

    @Version
    protected int version;

    @ManyToOne
    Address address;

    @Basic
    String description;

    @OneToMany(targetEntity = com.acme.Order.class,
               mappedBy = "customer")
    Collection orders = new Vector();

    @ManyToMany(mappedBy = "customers")
    Set<DeliveryService> serviceOptions = new HashSet();

    public Long getId() {
        return id;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address addr) {
        this.address = addr;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String desc) {
        this.description = desc;
    }

    public Collection getOrders() {
        return orders;
    }

    public Set<DeliveryService> getServiceOptions() {
        return serviceOptions;
    }
}
```

```
@Entity
public class Address {
    private Long id;
    private int version;
    private String street;

    @Id
    @GeneratedValue(strategy = AUTO)
    public Long getId() {
        return id;
    }

    protected void setId(Long id) {
        this.id = id;
    }

    @Version
    public int getVersion() {
        return version;
    }

    protected void setVersion(int version) {
        this.version = version;
    }

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }
}

@Entity
public class Order {
    private Long id;
    private int version;
    private String itemName;
    private int quantity;
    private Customer cust;

    @Id
    @GeneratedValue(strategy = AUTO)
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    protected int getVersion() {
        return version;
    }

    protected void setVersion(int version) {
        this.version = version;
    }

    public String getItemName() {
        return itemName;
    }

    public void setItemName(String itemName) {
        this.itemName = itemName;
```

```
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    @ManyToOne
    public Customer getCustomer() {
        return cust;
    }

    public void setCustomer(Customer cust) {
        this.cust = cust;
    }
}

@Entity
@Table(name = "DLVY_SVC")
public class DeliveryService {
    private String serviceName;
    private int priceCategory;
    private Collection customers;

    @Id
    public String getServiceName() {
        return serviceName;
    }

    public void setServiceName(String serviceName) {
        this.serviceName = serviceName;
    }

    public int getPriceCategory() {
        return priceCategory;
    }

    public void setPriceCategory(int priceCategory) {
        this.priceCategory = priceCategory;
    }

    @ManyToMany(targetEntity = com.acme.Customer.class)
    @JoinTable(name = "CUST_DLVRY")
    public Collection getCustomers() {
        return customers;
    }

    public setCustomers(Collection customers) {
        this.customers = customers;
    }
}
```

Next, we have a more complex example:

```
/***** Employee class *****/
@Entity
@Table(name = "EMPL")
@SecondaryTable(name = "EMP_SALARY",
                pkJoinColumns = @PrimaryKeyJoinColumn(name = "EMP_ID", referencedColumnName = "ID"))
public class Employee implements Serializable {
    private Long id;
    private int version;
    private String name;
```

```
private Address address;
private Collection phoneNumbers;
private Collection<Project> projects;
private Long salary;
private EmploymentPeriod period;

@Id
@GeneratedValue(strategy = TABLE)
public Integer getId() {
    return id;
}

protected void setId(Integer id) {
    this.id = id;
}

@Version
@Column(name = "EMP_VERSION", nullable = false)
public int getVersion() {
    return version;
}

protected void setVersion(int version) {
    this.version = version;
}

@Column(name = "EMP_NAME", length = 80)
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

@ManyToOne(cascade = PERSIST, optional = false)
@JoinColumn(name = "ADDR_ID", referencedColumnName = "ID", nullable = false)
public Address getAddress() {
    return address;
}

public void setAddress(Address address) {
    this.address = address;
}

@OneToMany(targetEntity = com.acme.PhoneNumber.class,
           cascade = ALL,
           mappedBy = "employee")
public Collection getPhoneNumbers() {
    return phoneNumbers;
}

public void setPhoneNumbers(Collection phoneNumbers) {
    this.phoneNumbers = phoneNumbers;
}

@ManyToMany(cascade = PERSIST, mappedBy = "employees")
@JoinTable(
    name = "EMP_PROJ",
    joinColumns = @JoinColumn(name = "EMP_ID", referencedColumnName = "ID"),
    inverseJoinColumns = @JoinColumn(name = "PROJ_ID", referencedColumnName = "ID"))
public Collection<Project> getProjects() {
    return projects;
}

public void setProjects(Collection<Project> projects) {
    this.projects = projects;
```

```
    }

    @Column(name = "EMP_SAL", table = "EMP_SALARY")
    public Long getSalary() {
        return salary;
    }

    public void setSalary(Long salary) {
        this.salary = salary;
    }

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "startDate",
                           column = @Column(name = "EMP_START")),
        @AttributeOverride(name = "endDate",
                           column = @Column(name = "EMP_END"))
    })
    public EmploymentPeriod getEmploymentPeriod() {
        return period;
    }

    public void setEmploymentPeriod(EmploymentPeriod period) {
        this.period = period;
    }
}

/***** Address class *****/
@Entity
public class Address implements Serializable {
    private Integer id;
    private int version;
    private String street;
    private String city;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    public Integer getId() {
        return id;
    }

    protected void setId(Integer id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERS", nullable = false)
    public int getVersion() {
        return version;
    }

    protected void setVersion(int version) {
        this.version = version;
    }

    @Column(name = "RUE")
    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    @Column(name = "VILLE")
    public String getCity() {
        return city;
```

```
    }

    public void setCity(String city) {
        this.city = city;
    }
}

/***** PhoneNumber class *****/
@Entity
@Table(name = "PHONE")
public class PhoneNumber implements Serializable {
    private String number;
    private int phoneType;
    private Employee employee;

    @Id
    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }

    @Column(name = "PTYPE")
    public int getPhonetype() {
        return phonetype;
    }

    public void setPhoneType(int phoneType) {
        this.phoneType = phoneType;
    }

    @ManyToOne(optional = false)
    @JoinColumn(name = "EMP_ID", nullable = false)
    public Employee getEmployee() {
        return employee;
    }

    public void setEmployee(Employee employee) {
        this.employee = employee;
    }
}

/***** Project class *****/
@Entity
@Inheritance(strategy = JOINED)
@DiscriminatorValue("Proj")
@DiscriminatorColumn(name = "DISC")
public class Project implements Serializable {
    private Integer projId;
    private int version;
    private String name;
    private Set<Employee> employees;

    @Id
    @GeneratedValue(strategy = TABLE)
    public Integer getId() {
        return projId;
    }

    protected void setId(Integer id) {
        this.projId = id;
    }

    @Version
    public int getVersion() {
```

```
        return version;
    }

    protected void setVersion(int version) {
        this.version = version;
    }

    @Column(name = "PROJ_NAME")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @ManyToMany(mappedBy = "projects")
    public Set<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Set<Employee> employees) {
        this.employees = employees;
    }
}

/***** GovernmentProject subclass *****/
@Entity
@Table(name = "GOVT_PROJECT")
@DiscriminatorValue("GovtProj")
@PrimaryKeyJoinColumn(name = "GOV_PROJ_ID", referencedColumnName = "ID")
public class GovernmentProject extends Project {
    private String fileInfo;

    @Column(name = "INFO")
    public String getFileInfo() {
        return fileInfo;
    }

    public void setFileInfo(String fileInfo) {
        this.fileInfo = fileInfo;
    }
}

/***** CovertProject subclass *****/
@Entity
@Table(name = "C_PROJECT")
@DiscriminatorValue("CovProj")
@PrimaryKeyJoinColumn(name = "COV_PROJ_ID", referencedColumnName = "ID")
public class CovertProject extends Project {
    private String classified;

    public CovertProject() {
        super();
    }

    public CovertProject(String classified) {
        this();
        this.classified = classified;
    }

    @Column(updatable = false)
    public String getClassified() {
        return classified;
    }

    protected void setClassified(String classified) {
```

```
            this.classified = classified;
        }
    }

    /***** EmploymentPeriod class *****/
    @Embeddable
    public class EmploymentPeriod implements Serializable {
        private Date start;
        private Date end;

        @Column(nullable = false)
        public Date getStartDate() {
            return start;
        }

        public void setStartDate(Date start) {
            this.start = start;
        }

        public Date getEndDate() {
            return end;
        }

        public void setEndDate(Date end) {
            this.end = end;
        }
    }
```

11.5. Derivation of schema metadata from Jakarta Validation constraints

Certain Jakarta Validation constraints can override the persistent attribute metadata and influence the schema generation.

The following list describes how constraints affect the persistent attributes they are applied to:

`jakarta.validation.constraints.NotNull`

This constraint is treated as a way to specify that the column mapped by the attribute is not nullable.

Influence on metadata

Overrides the `nullable` attribute and sets it to `false`.

Influence on DDL

Adds a `NOT NULL` constraint to the column.

`jakarta.validation.constraints.NotBlank`

Similar to `jakarta.validation.constraints.NotNull` this constraint is treated as a way to specify that the column mapped by the attribute is not nullable. With difference being that `@NotBlank` constraint is applicable to fewer types compared to `@NotNull` constraint.

Influence on metadata

Overrides the `nullable` attribute and sets it to `false`.

Influence on DDL

Adds a `NOT NULL` constraint to the column.

`jakarta.validation.constraints.NotEmpty`

Same as `jakarta.validation.constraints.NotNull` this constraint is treated as a way to specify that the column mapped by the attribute is not nullable. With difference being that `@NotEmpty` constraint is applicable to fewer

types compared to `@NotNull` constraint.

Influence on metadata

Overrides the `nullable` attribute and sets it to `false`.

Influence on DDL

Adds a `NOT NULL` constraint to the column.

`jakarta.validation.constraints.Digits`

This constraint is treated as a way to specify the precision and scale of a numeric column.

Influence on metadata

Sets the `precision` attribute to the sum of `integer` and `fraction`, and the `scale` attribute to `fraction`.

Influence on DDL

Defines the numeric type precision and scale, e.g. `DECIMAL(precision, scale)`.

`jakarta.validation.constraints.Size`

If this constraint specifies the maximum value, e.g. `@Size(max = {someValue})`, then it is treated as a way to specify the length of the column mapped by the attribute, i.e. `length={someValue}`.

Influence on metadata

Sets the `length` attribute to `{someValue}`.

Influence on DDL

Defines the column length, e.g. `VARCHAR({someValue})`.

`jakarta.validation.constraints.Min`

This constraint is treated as a way to specify a check constraint on a numeric column.

Influence on metadata

None.

Influence on DDL

Adds a check constraint to the column, e.g. `CHECK (column_name >= value)`, where `value` is the value defined in the constraint.

`jakarta.validation.constraints.Max`

This constraint is treated as a way to specify a check constraint on a numeric column.

Influence on metadata

None.

Influence on DDL

Adds a check constraint to the column, e.g. `CHECK (column_name <= value)`, where `value` is the value defined in the constraint.

`jakarta.validation.constraints.DecimalMin`

This constraint is treated as a way to specify a check constraint on a numeric column.

Influence on metadata

None.

Influence on DDL

Adds a check constraint to the column, e.g. `CHECK (column_name >= value)` if `inclusive` is `true` (default), or `CHECK (column_name > value)` otherwise.

`jakarta.validation.constraints.DecimalMax`

This constraint is treated as a way to specify a check constraint on a numeric column.

Influence on metadata

None.

Influence on DDL

Adds a check constraint to the column, e.g. `CHECK (column_name <= value)` if `inclusive` is `true` (default), or `CHECK (column_name < value)` otherwise.

`jakarta.validation.constraints.Negative`

This constraint is handled the same as `jakarta.validation.constraints.Max(-1)` for integral types and `jakarta.validation.constraints.DecimalMax(value = BigDecimal.ZERO, inclusive = false)` for decimal types.

Influence on metadata

None.

Influence on DDL

Adds a check constraint to the column, e.g. `CHECK (column_name < 0)`.

`jakarta.validation.constraints.NegativeOrZero`

This constraint is handled the same as `jakarta.validation.constraints.Max(0)` for integral types and `jakarta.validation.constraints.DecimalMax(value = BigDecimal.ZERO)` for decimal types.

Influence on metadata

None.

Influence on DDL

Adds a check constraint to the column, e.g. `CHECK (column_name <= 0)`.

`jakarta.validation.constraints.Positive`

This constraint is handled the same as `jakarta.validation.constraints.Min(1)` for integral types and `jakarta.validation.constraints.DecimalMin(value = BigDecimal.ZERO, inclusive = false)` for decimal types.

Influence on metadata

None.

Influence on DDL

Adds a check constraint to the column, e.g. `CHECK (column_name > 0)`.

`jakarta.validation.constraints.PositiveOrZero`

This constraint is handled the same as `jakarta.validation.constraints.Min(0)` for integral types and `jakarta.validation.constraints.DecimalMin(value = BigDecimal.ZERO)` for decimal types.

Influence on metadata

None.

Influence on DDL

Adds a check constraint to the column, e.g. `CHECK (column_name >= 0)`.

Persistence providers must comply with the following precedence and resolution rules:

- Jakarta Validation constraint metadata takes precedence over Jakarta Persistence.
- Jakarta Validation constraint metadata takes precedence over `@jakarta.annotation.Nonnull`.
- When multiple Jakarta Validation constraints apply to the same attribute, the most restrictive constraint must be applied.
- It is up to the persistence provider how to process multiple check clauses contributed by different constraints.

> This specification expects that only constraints that belong to the `jakarta.validation.groups.Default` validation group are considered for this additional metadata contribution.
>
> Persistent providers may have a way to override this expectation and allow users to specify a different validation group or groups to be considered for DDL operations.

[1] In particular, lazy fetching might only be available for `Basic` mappings for which property-based access is used.

[2] The use of map keys that contain embeddables that reference entities is not permitted.

[3] Note that `either` the `joinColumns` element `or` the `joinTable` element of the `AssociationOverride` annotation is specified for overriding a given relationship (but never both).

[4] Or of the subhierarchy in which a distinct inheritance mapping strategy is used. The combination of inheritance strategies within a single entity inheritance hierarchy is not defined by this specification.

[5] Or of the subhierarchy in which a distinct inheritance mapping strategy is used

[6] If it is not specified, the rules of Section 2.10 apply.

[7] If the embeddable class is used as a primary key, the `EmbeddedId` rather than the `Embedded` annotation must be used.

[8] Use of the `Embedded` annotation is not required. See Section 2.10.

[9] The `Id` annotation is not applied to any field or property of the embeddable class.

[10] If the element collection is a `Map`, this applies to the map value.

[11] Mapping of stateful enum values is not supported.

[12] Portable applications should not use the `GeneratedValue` annotation on other persistent fields or properties.

[13] Note that `SEQUENCE` and `IDENTITY` are not portable across all databases.

[14] A primary key with a type not listed is not portable.

[15] In general, floating point types should never be used in primary keys.

[16] If the element collection is a `Map`, this applies to the map value.

[17] The `ManyToMany` annotation must not be used within an embeddable class used in an element collection.

[18] The `OneToMany` annotation must not be used within an embeddable class used in an element collection.

[19] If the parent is detached or new or was previously removed before the orphan was associated with it, the remove operation is not applied to the entity being orphaned.

[20] If the parent is detached or new or was previously removed before the orphan was associated with it, the remove operation is not applied to the entity being orphaned.

[21] If the primary key is a composite primary key, the precedence of ordering among the attributes within the primary key is not futher defined. To assign such a precedence within these attributes, each of the individual attributes must be specified as an orderby_item.

[22] In all other cases when OrderBy is applied to an element collection, the `property_or_field_name` must be specified.

[23] The `OrderBy` annotation should be used for ordering that is visible as persistent state and maintained by the

application.

[24] It is not expected that a database foreign key be defined for the `OneToOne` mapping, as the `OneToOne` relationship may be defined as `optional=true`.

[25] The derived id mechanisms described in Section 2.4.2.2 are now to be preferred over `PrimaryKeyJoinColumn` for the `OneToOne` mapping case.

[26] Note that the derived identity mechanisms described in Section 2.4.2.2 is now preferred to the use of `PrimaryKeyJoinColumn` for this case.

[27] When a `JOINED` inheritance strategy is used, the `Table` annotation specifies a primary table for the subclass-specific state if the default is not used.

[28] If the element collection is a Map, this applies to the map value.

Chapter 12. XML Object/Relational Mapping Descriptor

The XML object/relational mapping descriptor serves as both an alternative to and an overriding mechanism for Java language metadata annotations.

12.1. Use of the XML Descriptor

The XML schema for the object relational/mapping descriptor is contained in Section 12.3. The root element of this schema is the `entity-mappings` element. The absence or present of the `xml-mapping-metadata-complete` subelement contained in the `persistence-unit-defaults` subelement of the `entity-mappings` element controls whether the XML object/relational mapping descriptor is used to selectively override annotation values or whether it serves as a complete alternative to Java language metadata annotations.

If the `xml-mapping-metadata-complete` subelement is specified, the complete set of mapping metadata for the persistence unit is contained in the XML mapping files for the persistence unit, and any persistence annotations on the classes are ignored.

If `xml-mapping-metadata-complete` is specified and XML elements are omitted, the default values apply. These default values are the same as the corresponding defaults when annotations are used, except in the cases specified in Section 12.2 below. When the `xml-mapping-metadata-complete` element is specified, any `metadata-complete` attributes specified within the `entity`, `mapped-superclass`, and `embeddable` elements are ignored.

If the `xml-mapping-metadata-complete` subelement is not specified, the XML descriptor overrides the values set or defaulted by the use of annotations, as described below.

The mapping files used by the application developer must conform to the XML schema defined in Section 12.3 or to the previous versions of the XML schema, `orm_3_2.xsd`, `orm_3_1.xsd`, `orm_3_0.xsd` or `orm_2_2.xsd`, defined in a previous version of this specification [1] in accordance with the version specified by the `orm.xml` file.

The Jakarta Persistence persistence provider may support use of older versions of the object/relational mapping schema as well as the object/relational mapping schema defined in Section 12.3, whether singly or in combination when multiple mapping files are used.

12.2. XML Overriding Rules

This section defines the rules that apply when the XML descriptor is used to override annotations, and the rules pertaining to the interaction of XML elements specified as subelements of the `persistence-unit-defaults`, `entity-mappings`, `entity`, `mapped-superclass`, and `embeddable` elements.

12.2.1. persistence-unit-defaults Subelements

12.2.1.1. schema

The `schema` subelement applies to all entities, tables, secondary tables, join tables, collection tables, table generators, and sequence generators in the persistence unit.

The `schema` subelement is overridden by any `schema` subelement of the `entity-mappings` element; any `schema` element explicitly specified in the `Table` or `SecondaryTable` annotation on an entity or any `schema` attribute on any `table` or `secondary-table` subelement defined within an `entity` element; any `schema` element explicitly specified in a `TableGenerator` annotation or `table-generator` subelement; any `schema` element explicitly specified in a `SequenceGenerator` annotation or `sequence-generator` subelement; any `schema` element explicitly specified in a `JoinTable` annotation or `join-table` subelement; and any `schema` element explicitly specified in a `CollectionTable` annotation or `collection-table` subelement.

12.2.1.2. catalog

The `catalog` subelement applies to all entities, tables, secondary tables, join tables, collection tables, table generators, and sequence generators in the persistence unit.

The `catalog` subelement is overridden by any `catalog` subelement of the `entity-mappings` element; any `catalog` element explicitly specified in the `Table` or `SecondaryTable` annotation on an entity or any `catalog` attribute on any `table` or `secondary-table` subelement defined within an `entity` XML element; any `catalog` element explicitly specified in a `TableGenerator` annotation or `table-generator` subelement; any `catalog` element explicitly specified in a `SequenceGenerator` annotation or `sequence-generator` subelement; any `catalog` element explicitly specified in a `JoinTable` annotation or `join-table` subelement; and any `catalog` element explicitly specified in a `CollectionTable` annotation or `collection-table` subelement.

12.2.1.3. delimited-identifiers

The `delimited-identifiers` subelement applies to the naming of database objects, as described in Section 2.15. It specifies that all database table-, schema-, and column-level identifiers in use for the persistence unit be treated as delimited identifiers.

The `delimited-identifiers` subelement cannot be overridden in this release.

12.2.1.4. access

The `access` subelement applies to all managed classes in the persistence unit.

The `access` subelement is overridden by the use of any annotations specifying mapping information on the fields or properties of the entity class; by any `Access` annotation on the entity class, mapped superclass, or embeddable class; by any `access` subelement of the `entity-mappings` element; by any `Access` annotation on a field or property of an entity class, mapped superclass, or embeddable class; by any `access` attribute defined within an `entity`, `mapped-superclass`, or `embeddable` XML element, or by any `access` attribute defined within an `id`, `embedded-id`, `version`, `basic`, `embedded`, `many-to-one`, `one-to-one`, `one-to-many`, `many-to-many`, or `element-collection` element.

12.2.1.5. cascade-persist

The `cascade-persist` subelement applies to all relationships in the persistence unit.

Specifying this subelement adds the cascade persist option to all relationships in addition to any settings specified in annotations or XML.

The `cascade-persist` subelement cannot be overridden in this release.

> The ability to override the `cascade-persist` of the `persistence-unit-defaults` element will be added in a future release of this specification.

12.2.1.6. entity-listeners

The `entity-listeners` subelement defines default entity listeners for the persistence unit. These entity listeners are called before any other entity listeners for an entity unless the entity listener order is overridden within a `mapped-superclass` or `entity` element, or the `ExcludeDefaultListeners` annotation is present on the entity or mapped superclass or the `exclude-default-listeners` subelement is specified within the corresponding `entity` or `mapped-superclass` XML element.

12.2.2. Other Subelements of the entity-mappings element

12.2.2.1. package

The `package` subelement specifies the package of the classes listed within the subelements and attributes of the same mapping file only. The `package` subelement is overridden if the fully qualified class name is specified for a class and the two disagree.

12.2.2.2. schema

The `schema` subelement applies only to the entities, tables, secondary tables, join tables, collection tables, table generators, and sequence generators listed within the same mapping file.

The `schema` subelement is overridden by any `schema` element explicitly specified in the `Table`, `SecondaryTable` , `JoinTable`, or `CollectionTable` annotation on an entity listed within the mapping file or any `schema` attribute on any `table` or `secondary-table` subelement defined within the `entity` element for such an entity, or by any `schema` attribute on any `join-table` or `collection-table` subelement of an attribute defined within the `attributes` subelement of the `entity` element for such an entity, or by the `schema` attribute of any `table-generator` or `sequence-generator` element within the mapping file.

12.2.2.3. catalog

The `catalog` subelement applies only to the entities, tables, secondary tables, join tables, collection tables, table generators, and sequence generators listed within the same mapping file.

The `catalog` subelement is overridden by any `catalog` element explicitly specified in the `Table`, `SecondaryTable` , `JoinTable`, or `CollectionTable` annotation on an entity listed within the mapping file or any `catalog` attribute on any `table` or `secondary-table` subelement defined within the `entity` element for such an entity, or by any `catalog` attribute on any `join-table` or `collection-table` subelement of an attribute defined within the `attributes` subelement of the `entity` element for such an entity, or by the `catalog` attribute of any `table-generator` or `sequence-generator` element within the mapping file.

12.2.2.4. access

The `access` subelement applies to the managed classes listed within the same mapping file.

The `access` subelement is overridden by the use of any annotations specifying mapping information on the fields or properties of the entity class; by any `Access` annotation on the entity class, mapped superclass, or embeddable class; by any `Access` annotation on a field or property of an entity class, mapped superclass, or embeddable class; by any `access` attribute defined within an `entity`, `mapped-superclass`, or `embeddable` XML element, or by any `access` attribute defined within an `id`, `embedded-id`, `version`, `basic`, `embedded`, `many-to-one`, `one-to-one`, `one-to-many`, `many-to-many`, or `element-collection` element.

12.2.2.5. sequence-generator

The generator defined by the `sequence-generator` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain generators of the same name.

The generator defined is added to any generators defined in annotations. If a generator of the same name is defined in annotations, the generator defined by this subelement overrides that definition.

12.2.2.6. table-generator

The generator defined by the `table-generator` subelement applies to the persistence unit. It is undefined if multiple

mapping files for the persistence unit contain generators of the same name.

The generator defined is added to any generators defined in annotations. If a generator of the same name is defined in annotations, the generator defined by this subelement overrides that definition.

12.2.2.7. named-query, named-native-query

A named query defined by a `named-query` or `named-native-query` element is scoped to the persistence unit.

If the mapping files for a persistence unit declare multiple queries with the same name, the behavior is undefined.

A named query declared in XML is added to the named queries declared using annotations. If a named query declared using annotations has the same name as the query declared in XML, the declaration in XML overrides the declaration in annotations.

12.2.2.8. named-statement, named-native-statement

A named statement defined by a `named-statement` or `named-native-statement` element is scoped to the persistence unit.

If the mapping files for a persistence unit declare multiple statements with the same name, the behavior is undefined.

A named statement declared in XML is added to the named statements declared using annotations. If a named statement declared using annotations has the same name as the statement declared in XML, the declaration in XML overrides the declaration in annotations.

12.2.2.9. named-stored-procedure-query

A named stored procedure query defined by `named-stored-procedure-query` element is scoped to the persistence unit.

If the mapping files for a persistence unit declare multiple stored procedure queries with the same name, the behavior is undefined.

A named stored procedure query declared in XML is added to the named stored procedure queries declared using annotations. If a named stored procedure query declared using annotations has the same name as the stored procedure query declared in XML, the declaration in XML overrides the declaration in annotations.

12.2.2.10. sql-result-set-mapping

The SQL result set mapping defined by the `sql-result-set-mapping` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain SQL result set mappings of the same name.

The SQL result set mapping defined is added to the SQL result set mappings defined in annotations. If a SQL result set mapping of the same name is defined in annotations, the SQL result set mapping defined by this subelement overrides that definition.

12.2.2.11. entity

The `entity` subelement defines an entity of the persistence unit. It is undefined if multiple mapping files for the persistence unit contain entries for the same entity.

The entity class may or may not have been annotated as `Entity`. The subelements and attributes of the `entity` element override as specified in Section 12.2.3.

12.2.2.12. mapped-superclass

The `mapped-superclass` subelement defines a mapped superclass of the persistence unit. It is undefined if multiple mapping files for the persistence unit contain entries for the same mapped superclass.

The mapped superclass may or may not have been annotated as `MappedSuperclass`. The subelements and attributes of the `mapped-superclass` element override as specified in Section 12.2.4.

12.2.2.13. embeddable

The `embeddable` subelement defines an embeddable class of the persistence unit. It is undefined if multiple mapping files for the persistence unit contain entries for the same embeddable class.

The `embeddable` class may or may not have been annotated as `Embeddable`. The subelements and attributes of the `embeddable` element override as specified in Section 12.2.5.

12.2.2.14. converter

The converter defined by the `converter` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain converters for the same target type.

The converter defined is added to the converters defined in annotations. If a converter for the same target type is defined in annotations, the converter defined by this subelement overrides that definition.

12.2.3. entity Subelements and Attributes

These apply only to the entity for which they are subelements or attributes, unless otherwise specified below.

12.2.3.1. metadata-complete

If the `metadata-complete` attribute of the `entity` element is specified as `true`, any annotations on the entity class (and its fields and properties) are ignored. When `metadata-complete` is specified as `true` and XML attributes or sub-elements of the `entity` element are omitted, the default values for those attributes and elements are applied.

12.2.3.2. access

The `access` attribute defines the access type for the entity. The `access` attribute overrides any access type specified by the `persistence-unit-defaults` element or `entity-mappings` element for the given entity. The access type for a field or property of the entity may be overridden by specifying by overriding the mapping for that field or property using the appropriate XML subelement, as described in Section 12.2.3.26 below.

Caution must be exercised in overriding an access type that was specified or defaulted using annotations, as doing so may cause applications to break.

12.2.3.3. cacheable

The `cacheable` attribute defines whether the entity should be cached or must not be cached when the `shared-cache-mode` element of the `persistence.xml` file is specified as `ENABLE_SELECTIVE` or `DISABLE_SELECTIVE`. If the `Cacheable` annotation was specified for the entity, its value is overridden by this attribute. The value of the `cacheable` attribute is inherited by subclasses (unless otherwise overridden for a subclass by the `Cacheable` annotation or `cacheable` XML attribute).

12.2.3.4. name

The name attribute specifies the entity name. The `name` attribute overrides the value of the entity name defined by the `name` element of the `Entity` annotation (whether explicitly specified or defaulted). Caution must be exercised in overriding the entity name, as doing so may cause applications to break.

12.2.3.5. table

The `table` subelement overrides any `Table` annotation (including defaulted `Table` values) on the entity. If a `table` subelement is present, and attributes or subelements of that `table` subelement are not explicitly specified, their default values are applied.

12.2.3.6. secondary-table

The `secondary-table` subelement overrides all `SecondaryTable` and `SecondaryTables` annotations (including defaulted `SecondaryTable` values) on the entity. If a `secondary-table` subelement is present, and attributes or subelements of that `secondary - table` subelement are not explicitly specified, their default values are applied.

12.2.3.7. primary-key-join-column

The `primary-key-join-column` subelement of the entity element specifies a primary key column that is used to join the table of an entity subclass to the primary table for the entity when the joined strategy is used. The `primary-key-join-column` subelement overrides all `PrimaryKeyJoinColumn` and `PrimaryKeyJoinColumns` annotations (including defaulted `PrimaryKeyJoinColumn` values) on the entity. If a `primary-key-join-column` subelement is present, and attributes or subelements of that `primary-key-join-column` subelement are not explicitly specified, their default values are applied.

12.2.3.8. id-class

The `id-class` subelement overrides any `IdClass` annotation specified on the entity.

12.2.3.9. inheritance

The `inheritance` subelement overrides any `Inheritance` annotation (including defaulted `Inheritance` values) on the entity. If an `inheritance` subelement is present, and the `strategy` attribute is not explicitly specified, its default value is applied.

This element applies to the entity and its subclasses (unless otherwise overridden for a subclass by an annotation or XML element).

Support for the combination of inheritance strategies is not required by this specification. Portable applications should use only a single inheritance strategy within an entity hierarchy.

12.2.3.10. discriminator-value

The `discriminator-value` subelement overrides any `DiscriminatorValue` annotations (including defaulted `DiscriminatorValue` values) on the entity.

12.2.3.11. discriminator-column

The `discriminator-column` subelement overrides any `DiscriminatorColumn` annotation (including defaulted `DiscriminatorColumn` values) on the entity. If a `discriminator-column` subelement is present, and attributes of that `discriminator-column` subelement are not explicitly specified, their default values are applied.

This element applies to the entity and its subclasses (unless otherwise overridden for a subclass by an annotation or XML element).

12.2.3.12. sequence-generator

The generator defined by the `sequence-generator` subelement is added to any generators defined in annotations and any other generators defined in XML. If a generator of the same name is defined in annotations, the generator defined by this subelement overrides that definition. If a `sequence-generator` subelement is present, and attributes or subelements of that `sequence-generator` subelement are not explicitly specified, their default values are applied.

The generator defined by the `sequence-generator` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain generators of the same name.

12.2.3.13. table-generator

The generator defined by the `table-generator` subelement is added to any generators defined in annotations and any other generators defined in XML. If a generator of the same name is defined in annotations, the generator defined by this subelement overrides that definition. If a `table-generator` subelement is present, and attributes or subelements of that `table-generator` subelement are not explicitly specified, their default values are applied.

The generator defined by the `table-generator` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain generators of the same name.

12.2.3.14. attribute-override

The `attribute-override` subelement is additive to any `AttributeOverride` or `AttributeOverrides` annotations on the entity. It overrides any `AttributeOverride` elements for the same attribute name. If an `attribute-override` subelement is present, and attributes or subelements of that `attribute-override` subelement are not explicitly specified, their default values are applied.

12.2.3.15. association-override

The `association-override` subelement is additive to any `AssociationOverride` or `AssociationOverrides` annotations on the entity. It overrides any `AssociationOverride` elements for the same attribute name. If an `association-override` subelement is present, and attributes or subelements of that `association-override` subelement are not explicitly specified, their default values are applied.

12.2.3.16. convert

The `convert` subelement is additive to any `Convert` or `Converts` annotations on the entity. It overrides any `Convert` annotation for the same attribute name. If a `convert` subelement is present, and attributes or subelements of that `convert` subelement are not explicitly specified, their default values are applied.

12.2.3.17. named-entity-graph

The `named-entity-graph` subelement is additive to any `NamedEntityGraph` annotations on the entity. It overrides any `NamedEntityGraph` annotation with the same name.

12.2.3.18. named-query

The named query defined by the `named-query` subelement is added to any named queries defined in annotations, and

any other named queries defined in XML. If a named query of the same name is defined in annotations, the named query defined by this subelement overrides that definition. If a `named-query` subelement is present, and attributes or subelements of that `named-query` subelement are not explicitly specified, their default values are applied.

The named query defined by the `named-query` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain named queries of the same name.

12.2.3.19. named-native-query

The named query defined by the `named-native-query` subelement is added to any named queries defined in annotations, and any other named queries defined in XML. If a named query of the same name is defined in annotations, the named query defined by this subelement overrides that definition. If a `named-native-query` subelement is present, and attributes or subelements of that `named-native-query` subelement are not explicitly specified, their default values are applied.

The named native query defined by the `named-native-query` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain named queries of the same name.

12.2.3.20. named-stored-procedure-query

The named stored procedure query defined by the `named-stored-procedure-query` subelement is added to any named stored procedure queries defined in annotations, and any other named stored procedure queries defined in XML. If a named stored procedure query of the same name is defined in annotations, the named stored procedure query defined by this subelement overrides that definition. If a `named-stored-procedure-query` subelement is present, and attributes or subelements of that `named-stored-procedure-query` subelement are not explicitly specified, their default values are applied.

The named stored procedure query defined by the `named-stored-procedure-query` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain named stored procedure queries of the same name.

12.2.3.21. sql-result-set-mapping

The SQL result set mapping defined by the `sql-result-set-mapping` is added to the SQL result set mappings defined in annotations, and any other SQL result set mappings defined in XML. If a SQL result set mapping of the same name is defined in annotations, the SQL result set mapping defined by this subelement overrides that definition. If a `sql-result-set-mapping` subelement is present, and attributes or subelements of that `sql-result-set-mapping` subelement are not explicitly specified, their default values are applied.

The SQL result set mapping defined by the `sql-result-set-mapping` subelement applies to the persistence unit. It is undefined if multiple mapping files for the persistence unit contain SQL result set mappings of the same name.

12.2.3.22. exclude-default-listeners

The `exclude-default-listeners` subelement applies whether or not the `ExcludeDefaultListeners` annotation was specified on the entity.

This element causes the default entity listeners to be excluded for the entity and its subclasses.

12.2.3.23. exclude-superclass-listeners

The `exclude-superclass-listeners` subelement applies whether or not the `ExcludeSuperclassListeners` annotation was specified on the entity.

This element causes any superclass listeners to be excluded for the entity and its subclasses.

12.2.3.24. entity-listeners

The `entity-listeners` subelement overrides any `EntityListeners` annotation on the entity.

These listeners apply to the entity and its subclasses unless otherwise excluded.

12.2.3.25. pre-* and post-*

The `pre-persist`, `post-persist`, `pre-remove`, `post-remove`,`pre-merge`, `pre-insert`, `post-insert`, `pre-delete`, `post-delete`, `pre-upsert`, `post-upsert`, `pre-update`, `post-update`, and `post-load` subelements override any lifecycle callback methods defined by the corresponding annotations on the entity.

12.2.3.26. attributes

The `attributes` element groups the mapping subelements for the fields and properties of the entity. It may be sparsely populated to include only a subset of the fields and properties. If the value of `metadata-complete` is `true`, the remainder of the attributes are defaulted according to the default rules. If `metadata-complete` is not specified, or is `false`, the mappings for only those properties and fields that are explicitly specified are overridden.

id

The `id` subelement overrides the mapping for the specified field or property. If an `id` subelement is present, and attributes or subelements of that `id` subelement are not explicitly specified, their default values are applied.

embedded-id

The `embedded-id` subelement overrides the mapping for the specified field or property. If an `embedded-id` subelement is present, and attributes or subelements of that `embedded-id` subelement are not explicitly specified, their default values are applied.

basic

The `basic` subelement overrides the mapping for the specified field or property. If a `basic` subelement is present, and attributes or subelements of that `basic` subelement are not explicitly specified, their default values are applied.

version

The `version` subelement overrides the mapping for the specified field or property. If a `version` subelement is present, and attributes or subelements of that `version` subelement are not explicitly specified, their default values are applied.

many-to-one

The `many-to-one` subelement overrides the mapping for the specified field or property. If a `many-to-one` subelement is present, and attributes or subelements of that `many-to-one` subelement are not explicitly specified, their default values are applied.

one-to-many

The `one-to-many` subelement overrides the mapping for the specified field or property. If a `one-to-many` subelement is present, and attributes or subelements of that `one-to-many` subelement are not explicitly specified, their default values are applied.

one-to-one

The `one-to-one` subelement overrides the mapping for the specified field or property. If a `one-to-one` subelement is present, and attributes or subelements of that `one-to-one` subelement are not explicitly specified, their default values are applied.

many-to-many

The `many-to-many` subelement overrides the mapping for the specified field or property. If a `many-to-many` subelement is present, and attributes or subelements of that `many-to-many` subelement are not explicitly specified, their default values are applied.

element-collection

The `element-collection` subelement overrides the mapping for the specified field or property. If an `element-collection` subelement is present, and attributes or subelements of that `element-collection` subelement are not explicitly specified, their default values are applied.

embedded

The `embedded` subelement overrides the mapping for the specified field or property. If an `embedded` subelement is present, and attributes or subelements of that `embedded` subelement are not explicitly specified, their default values are applied.

excluded-from-versioning

The `excluded-from-versioning` subelement overrides the mapping for the specified field or property.

transient

The `transient` subelement overrides the mapping for the specified field or property.

12.2.4. mapped-superclass Subelements and Attributes

These apply only to the mapped-superclass for which they are subelements or attributes, unless otherwise specified below.

12.2.4.1. metadata-complete

If the `metadata-complete` attribute of the `mapped-superclass` element is specified as `true`, any annotations on the mapped superclass (and its fields and properties) are ignored. When `metadata-complete` is specified as `true` and attributes or subelements of the `mapped-superclass` element are omitted, the default values for those attributes and elements are applied.

12.2.4.2. access

The `access` attribute defines the access type for the mapped superclass. The `access` attribute overrides any access type specified by the `persistence-unit-defaults` element or `entity-mappings` element for the given mapped superclass. The access type for a field or property of the mapped superclass may be overridden by specifying by overriding the mapping for that field or property using the appropriate XML subelement, as described in Section 12.2.4.8 below.

Caution must be exercised in overriding an access type that was specified or defaulted using annotations, as doing so may cause applications to break.

12.2.4.3. id-class

The `id-class` subelement overrides any `IdClass` annotation specified on the mapped superclass.

12.2.4.4. exclude-default-listeners

The `exclude-default-listeners` subelement applies whether or not the `ExcludeDefaultListeners` annotation was specified on the mapped superclass.

This element causes the default entity listeners to be excluded for the mapped superclass and its subclasses.

12.2.4.5. exclude-superclass-listeners

The `exclude-superclass-listeners` subelement applies whether or not the `ExcludeSuperclassListeners` annotation was specified on the mapped superclass.

This element causes any superclass listeners to be excluded for the mapped superclass and its subclasses.

12.2.4.6. entity-listeners

The `entity-listeners` subelement overrides any `EntityListeners` annotation on the mapped superclass.

These listeners apply to the mapped superclass and its subclasses unless otherwise excluded.

12.2.4.7. pre-* and post-*

The `pre-persist`, `post-persist`, `pre-remove`, `post-remove`,`pre-merge`, `pre-insert`, `post-insert`, `pre-delete`, `post-delete`, `pre-upsert`, `post-upsert`, `pre-update`, `post-update`, and `post-load` subelements override any lifecycle callback methods defined by the corresponding annotations on the mapped superclass.

12.2.4.8. attributes

The `attributes` element groups the mapping subelements for the fields and properties defined by the mapped superclass. It may be sparsely populated to include only a subset of the fields and properties. If the value of `metadata-complete` is `true`, the remainder of the attributes are defaulted according to the default rules. If `metadata-complete` is not specified, or is `false`, the mappings for only those properties and fields that are explicitly specified are overridden.

id

The `id` subelement overrides the mapping for the specified field or property. If an `id` subelement is present, and attributes or subelements of that `id` subelement are not explicitly specified, their default values are applied.

embedded-id

The `embedded-id` subelement overrides the mapping for the specified field or property. If an `embedded-id` subelement is present, and attributes or subelements of that `embedded-id` subelement are not explicitly specified, their default values are applied.

basic

The `basic` subelement overrides the mapping for the specified field or property. If a `basic` subelement is present, and attributes or subelements of that `basic` subelement are not explicitly specified, their default values are applied.

version

The `version` subelement overrides the mapping for the specified field or property. If a `version` subelement is present, and attributes or subelements of that `version` subelement are not explicitly specified, their default values are applied.

many-to-one

The `many-to-one` subelement overrides the mapping for the specified field or property. If a `many-to-one` subelement is present, and attributes or subelements of that `many-to-one` subelement are not explicitly specified, their default values are applied.

one-to-many

The `one-to-many` subelement overrides the mapping for the specified field or property. If a `one-to-many` subelement is present, and attributes or subelements of that `one-to-many` subelement are not explicitly specified, their default values are applied.

one-to-one

The `one-to-one` subelement overrides the mapping for the specified field or property. If a `one-to-one` subelement is present, and attributes or subelements of that `one-to-one` subelement are not explicitly specified, their default values are applied.

many-to-many

The `many-to-many` subelement overrides the mapping for the specified field or property. If a `many-to-many` subelement is present, and attributes or subelements of that `many-to-many` subelement are not explicitly specified, their default values are applied.

element-collection

The `element-collection` subelement overrides the mapping for the specified field or property. If an `element-collection` subelement is present, and attributes or subelements of that `element-collection` subelement are not explicitly specified, their default values are applied.

embedded

The `embedded` subelement overrides the mapping for the specified field or property. If an `embedded` subelement is present, and attributes or subelements of that `embedded` subelement are not explicitly specified, their default values are applied.

transient

The `transient` subelement overrides the mapping for the specified field or property.

12.2.5. embeddable Subelements and Attributes

These apply only to the embeddable for which they are subelements or attributes.

12.2.5.1. metadata-complete

If the `metadata-complete` attribute of the `embeddable` element is specified as `true`, any annotations on the embeddable class (and its fields and properties) are ignored. When `metadata-complete` is specified as `true` and attributes and sub-elements of the `embeddable` element are omitted, the default values for those attributes and elements are applied.

12.2.5.2. access

The `access` attribute defines the access type for the embeddable class. The `access` attribute overrides any access type specified by the `persistence-unit-defaults` element or `entity-mappings` element for the given embeddable class. The access type for a field or property of the embeddable class may be overridden by specifying by overriding the mapping for that field or property using the appropriate XML subelement, as described in Section 12.2.5.3 below.

Caution must be exercised in overriding an access type that was specified or defaulted using annotations, as doing so may cause applications to break.

12.2.5.3. attributes

The `attributes` element groups the mapping subelements for the fields and properties defined by the embeddable class. It may be sparsely populated to include only a subset of the fields and properties. If the value of `metadata-complete` is `true`, the remainder of the attributes are defaulted according to the default rules. If `metadata-complete` is not specified, or is `false`, the mappings for only those properties and fields that are explicitly specified are overridden.

basic

The `basic` subelement overrides the mapping for the specified field or property. If a `basic` subelement is present, and attributes or subelements of that `basic` subelement are not explicitly specified, their default values are applied.

many-to-one

The `many-to-one` subelement overrides the mapping for the specified field or property. If a `many-to-one` subelement is present, and attributes or subelements of that `many-to-one` subelement are not explicitly specified, their default values are applied.

one-to-many

The `one-to-many` subelement overrides the mapping for the specified field or property. If a `one-to-many` subelement is present, and attributes or subelements of that `one-to-many` subelement are not explicitly specified, their default values are applied.

one-to-one

The `one-to-one` subelement overrides the mapping for the specified field or property. If a `one-to-one` subelement is present, and attributes or subelements of that `one-to-one` subelement are not explicitly specified, their default values are applied.

many-to-many

The `many-to-many` subelement overrides the mapping for the specified field or property. If a `many-to-many` subelement is present, and attributes or subelements of that `many-to-many` subelement are not explicitly specified, their default values are applied.

element-collection

The `element-collection` subelement overrides the mapping for the specified field or property. If an `element-collection` subelement is present, and attributes or subelements of that `element-collection` subelement are not explicitly specified, their default values are applied.

embedded

The `embedded` subelement overrides the mapping for the specified field or property. If an `embedded` subelement is present, and attributes or subelements of that `embedded` subelement are not explicitly specified, their default values are applied.

transient

The `transient` subelement overrides the mapping for the specified field or property.

12.3. XML Schema

This section provides the XML object/relational mapping schema for use with the Persistence API.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Jakarta Persistence API object/relational mapping file schema -->
<xsd:schema targetNamespace="https://jakarta.ee/xml/ns/persistence/orm"
  xmlns:orm="https://jakarta.ee/xml/ns/persistence/orm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="4.0">
  <xsd:annotation>
     <xsd:documentation><![CDATA[

       This is the XML Schema for the persistence object/relational mapping file.
       The file may be named "META-INF/orm.xml" in the persistence
       archive or it may be named some other name which would be
       used to locate the file as resource on the classpath.

       Object/relational mapping files must indicate the object/relational
       mapping file schema by using the persistence namespace:

       https://jakarta.ee/xml/ns/persistence/orm

       and indicate the version of the schema by
       using the version element as shown below:

      <entity-mappings xmlns="https://jakarta.ee/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence/orm
        https://jakarta.ee/xml/ns/persistence/orm/orm_4_0.xsd"
      version="4.0">
        ...
    </entity-mappings>


   ]]></xsd:documentation>
</xsd:annotation>

<xsd:complexType name="emptyType"/>

<xsd:simpleType name="versionType">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9]+(\.[0-9]+)*"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- ************************************************** -->

<xsd:element name="entity-mappings">
  <xsd:complexType>
    <xsd:annotation>
      <xsd:documentation>

      The entity-mappings element is the root element of a mapping
      file. It contains the following four types of elements:

      1. The persistence-unit-metadata element contains metadata
      for the entire persistence unit. It is undefined if this element
      occurs in multiple mapping files within the same persistence unit.

      2. The package, schema, catalog and access elements apply to all of
      the entity, mapped-superclass and embeddable elements defined in
      the same file in which they occur.

      3. The sequence-generator, table-generator, converter, named-query,
      named-native-query, named-stored-procedure-query, and
      sql-result-set-mapping elements are global to the persistence
      unit. It is undefined to have more than one sequence-generator
      or table-generator of the same name in the same or different
      mapping files in a persistence unit. It is undefined to have
      more than one named-query, named-native-query, sql-result-set-mapping,
      or named-stored-procedure-query of the same name in the same
      or different mapping files in a persistence unit.  It is also
      undefined to have more than one converter for the same target
      type in the same or different mapping files in a persistence unit.

      4. The entity, mapped-superclass and embeddable elements each define
      the mapping information for a managed persistent class. The mapping
      information contained in these elements may be complete or it may
      be partial.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string"
                   minOccurs="0"/>
      <xsd:element name="persistence-unit-metadata"
                   type="orm:persistence-unit-metadata"
                   minOccurs="0"/>
      <xsd:element name="package" type="xsd:string"
                   minOccurs="0"/>
      <xsd:element name="schema" type="xsd:string"
                   minOccurs="0"/>
      <xsd:element name="catalog" type="xsd:string"
                   minOccurs="0"/>
```

```
      <xsd:element name="access" type="orm:access-type"
                   minOccurs="0"/>
      <xsd:element name="sequence-generator" type="orm:sequence-generator"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="table-generator" type="orm:table-generator"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="named-query" type="orm:named-query"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="named-native-query" type="orm:named-native-query"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="named-statement" type="orm:named-statement"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="named-native-statement" type="orm:named-native-statement"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="named-stored-procedure-query"
                   type="orm:named-stored-procedure-query"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="sql-result-set-mapping"
                   type="orm:sql-result-set-mapping"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="mapped-superclass" type="orm:mapped-superclass"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="entity" type="orm:entity"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="embeddable" type="orm:embeddable"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="converter" type="orm:converter"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="version" type="orm:versionType"
                   fixed="4.0" use="required"/>
  </xsd:complexType>
</xsd:element>

<!-- ************************************************** -->

<xsd:complexType name="persistence-unit-metadata">
  <xsd:annotation>
    <xsd:documentation>

      Metadata that applies to the persistence unit and not just to
      the mapping file in which it is contained.

      If the xml-mapping-metadata-complete element is specified,
      the complete set of mapping metadata for the persistence unit
      is contained in the XML mapping files for the persistence unit.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    <xsd:element name="xml-mapping-metadata-complete" type="orm:emptyType"
                 minOccurs="0"/>
    <xsd:element name="persistence-unit-defaults"
                 type="orm:persistence-unit-defaults"
                 minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="persistence-unit-defaults">
  <xsd:annotation>
    <xsd:documentation>

      These defaults are applied to the persistence unit as a whole
      unless they are overridden by local annotation or XML
```

```
      element settings.

      schema - Used as the schema for all tables, secondary tables, join
          tables, collection tables, sequence generators, and table
          generators that apply to the persistence unit
      catalog - Used as the catalog for all tables, secondary tables, join
          tables, collection tables, sequence generators, and table
          generators that apply to the persistence unit
      delimited-identifiers - Used to treat database identifiers as
          delimited identifiers.
      access - Used as the access type for all managed classes in
          the persistence unit
      cascade-persist - Adds cascade-persist to the set of cascade options
          in all entity relationships of the persistence unit
      entity-listeners - List of default entity listeners to be invoked
          on each entity in the persistence unit.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="schema" type="xsd:string"
                   minOccurs="0"/>
      <xsd:element name="catalog" type="xsd:string"
                   minOccurs="0"/>
      <xsd:element name="delimited-identifiers" type="orm:emptyType"
                   minOccurs="0"/>
      <xsd:element name="access" type="orm:access-type"
                   minOccurs="0"/>
      <xsd:element name="cascade-persist" type="orm:emptyType"
                   minOccurs="0"/>
      <xsd:element name="entity-listeners" type="orm:entity-listeners"
                   minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="entity">
  <xsd:annotation>
    <xsd:documentation>

      Defines the settings and mappings for an entity. Is allowed to be
      sparsely populated and used in conjunction with the annotations.
      Alternatively, the metadata-complete attribute can be used to
      indicate that no annotations on the entity class (and its fields
      or properties) are to be processed. If this is the case then
      the defaulting rules for the entity and its subelements will
      be recursively applied.

      @Target(TYPE) @Retention(RUNTIME)
      public @interface Entity {
        String name() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    <xsd:element name="table" type="orm:table"
                 minOccurs="0"/>
    <xsd:element name="secondary-table" type="orm:secondary-table"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:sequence>
      <xsd:element name="primary-key-join-column"
                   type="orm:primary-key-join-column"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="primary-key-foreign-key"
```

```
              type="orm:foreign-key"
              minOccurs="0"/>
</xsd:sequence>
<xsd:element name="id-class" type="orm:id-class" minOccurs="0"/>
<xsd:element name="inheritance" type="orm:inheritance" minOccurs="0"/>
<xsd:element name="discriminator-value" type="orm:discriminator-value"
             minOccurs="0"/>
<xsd:element name="discriminator-column"
             type="orm:discriminator-column"
             minOccurs="0"/>
<xsd:element name="sequence-generator" type="orm:sequence-generator"
             minOccurs="0"/>
<xsd:element name="table-generator" type="orm:table-generator"
             minOccurs="0"/>
<xsd:element name="named-query" type="orm:named-query"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="named-native-query" type="orm:named-native-query"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="named-statement" type="orm:named-statement"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="named-native-statement" type="orm:named-native-statement"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="named-stored-procedure-query"
             type="orm:named-stored-procedure-query"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="sql-result-set-mapping"
             type="orm:sql-result-set-mapping"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="exclude-default-listeners" type="orm:emptyType"
             minOccurs="0"/>
<xsd:element name="exclude-superclass-listeners" type="orm:emptyType"
             minOccurs="0"/>
<xsd:element name="entity-listeners" type="orm:entity-listeners"
             minOccurs="0"/>
<xsd:element name="pre-merge" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreMerge {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-persist" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PrePersist {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-persist" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostPersist {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-remove" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>
```

```
      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreRemove {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-remove" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostRemove {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-update" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreUpdate {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-update" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostUpdate {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-upsert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreUpsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-upsert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostUpsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-insert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreInsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
```

```
      <xsd:element name="post-insert" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PostInsert {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="pre-delete" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PreDelete {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="post-delete" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PostDelete {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="post-load" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PostLoad {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="attribute-override" type="orm:attribute-override"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="association-override"
                   type="orm:association-override"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="convert" type="orm:convert"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="named-entity-graph" type="orm:named-entity-graph"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="attributes" type="orm:attributes" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="class" type="xsd:string" use="required"/>
    <xsd:attribute name="access" type="orm:access-type"/>
    <xsd:attribute name="cacheable" type="xsd:boolean"/>
    <xsd:attribute name="metadata-complete" type="xsd:boolean"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:simpleType name="access-type">
    <xsd:annotation>
      <xsd:documentation>

        This element determines how the persistence provider accesses the
        state of an entity or embedded object.
```

```
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="PROPERTY"/>
      <xsd:enumeration value="FIELD"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- ***************************************************** -->

  <xsd:complexType name="association-override">
    <xsd:annotation>
      <xsd:documentation>

        @Repeatable(AssociationOverrides.class)
        @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
        public @interface AssociationOverride {
          String name();
          JoinColumn[] joinColumns() default{};
          ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
          JoinTable joinTable() default @JoinTable;
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="join-column" type="orm:join-column"
                       minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="foreign-key" type="orm:foreign-key"
                       minOccurs="0"/>
        </xsd:sequence>
        <xsd:element name="join-table" type="orm:join-table"
                     minOccurs="0"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="attribute-override">
    <xsd:annotation>
      <xsd:documentation>

        @Repeatable(AttributeOverrides.class)
        @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
        public @interface AttributeOverride {
          String name();
          Column column();
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="column" type="orm:column"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="attributes">
    <xsd:annotation>
```

```
      <xsd:documentation>

        This element contains the entity field or property mappings.
        It may be sparsely populated to include only a subset of the
        fields or properties. If metadata-complete for the entity is true
        then the remainder of the attributes will be defaulted according
        to the default rules.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:choice>
        <xsd:element name="id" type="orm:id"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="embedded-id" type="orm:embedded-id"
                     minOccurs="0"/>
      </xsd:choice>
      <xsd:element name="basic" type="orm:basic"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="version" type="orm:version"
                   minOccurs="0"/>
      <xsd:element name="many-to-one" type="orm:many-to-one"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="one-to-many" type="orm:one-to-many"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="one-to-one" type="orm:one-to-one"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="many-to-many" type="orm:many-to-many"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="element-collection" type="orm:element-collection"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="embedded" type="orm:embedded"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="excluded-from-versioning"
                   type="orm:excluded-from-versioning"
                   minOccurs="0"/>
      <xsd:element name="transient" type="orm:transient"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="basic">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Basic {
          FetchType fetch() default FetchType.EAGER;
          boolean optional() default true;
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="column" type="orm:column" minOccurs="0"/>
      <xsd:choice>
        <xsd:element name="lob" type="orm:lob" minOccurs="0"/>
        <xsd:element name="temporal" type="orm:temporal" minOccurs="0"/>
        <xsd:element name="enumerated" type="orm:enumerated" minOccurs="0"/>
        <xsd:element name="convert" type="orm:convert" minOccurs="0"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="fetch" type="orm:fetch-type"/>
```

```
    <xsd:attribute name="optional" type="xsd:boolean"/>
    <xsd:attribute name="access" type="orm:access-type"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="cascade-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum CascadeType { ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="cascade-all" type="orm:emptyType"
                   minOccurs="0"/>
      <xsd:element name="cascade-persist" type="orm:emptyType"
                   minOccurs="0"/>
      <xsd:element name="cascade-merge" type="orm:emptyType"
                   minOccurs="0"/>
      <xsd:element name="cascade-remove" type="orm:emptyType"
                   minOccurs="0"/>
      <xsd:element name="cascade-refresh" type="orm:emptyType"
                   minOccurs="0"/>
      <xsd:element name="cascade-detach" type="orm:emptyType"
                   minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="check-constraint">
    <xsd:annotation>
      <xsd:documentation>

        @Target({}) @Retention(RUNTIME)
        public @interface CheckConstraint {
          String name() default "";
          String constraint();
          String options() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="constraint" type="xsd:string" use="required"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="collection-table">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface CollectionTable {
          String name() default "";
          String catalog() default "";
          String schema() default "";
          JoinColumn[] joinColumns() default {};
          ForeignKey foreignKey() default @ForeignKey(ConstraintMode.PROVIDER_DEFAULT);
          UniqueConstraint[] uniqueConstraints() default {};
          Index[] indexes() default {};
          CheckConstraint[] check() default {};
          String comment() default "";
```

```
        String type() default "";
        String options() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:sequence>
      <xsd:element name="comment" type="xsd:string" minOccurs="0" />
      <xsd:element name="join-column" type="orm:join-column"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="foreign-key" type="orm:foreign-key"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:element name="unique-constraint" type="orm:unique-constraint"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="index" type="orm:index"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="check-constraint" type="orm:check-constraint"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="catalog" type="xsd:string"/>
  <xsd:attribute name="schema" type="xsd:string"/>
  <xsd:attribute name="type" type="xsd:string"/>
  <xsd:attribute name="options" type="xsd:string"/>
</xsd:complexType>


<!-- ************************************************** -->

<xsd:complexType name="column">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Column {
        String name() default "";
        boolean unique() default false;
        boolean nullable() default true;
        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String options() default "";
        String table() default "";
        int length() default 255;
        int precision() default 0; // decimal precision
        int scale() default 0; // decimal scale
        int secondPrecision() default -1; //fractional second precision
        CheckConstraint[] check() default {};
        String comment() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="comment" type="xsd:string" minOccurs="0" />
    <xsd:element name="check-constraint" type="orm:check-constraint"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="unique" type="xsd:boolean"/>
  <xsd:attribute name="nullable" type="xsd:boolean"/>
  <xsd:attribute name="insertable" type="xsd:boolean"/>
  <xsd:attribute name="updatable" type="xsd:boolean"/>
  <xsd:attribute name="column-definition" type="xsd:string"/>
  <xsd:attribute name="options" type="xsd:string"/>
```

```
    <xsd:attribute name="table" type="xsd:string"/>
    <xsd:attribute name="length" type="xsd:int"/>
    <xsd:attribute name="precision" type="xsd:int"/>
    <xsd:attribute name="scale" type="xsd:int"/>
    <xsd:attribute name="second-precision" type="xsd:int"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="column-result">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target(METHOD) @Retention(RUNTIME)
        public @interface ColumnResult {
          String name();
          Class<?> type() default void.class;
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="class" type="xsd:string"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:simpleType name="constraint-mode">
    <xsd:annotation>
      <xsd:documentation>

       public enum ConstraintMode { CONSTRAINT, NO_CONSTRAINT, PROVIDER_DEFAULT }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="CONSTRAINT"/>
      <xsd:enumeration value="NO_CONSTRAINT"/>
      <xsd:enumeration value="PROVIDER_DEFAULT"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- ***************************************************** -->


  <xsd:complexType name="constructor-result">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target(METHOD) @Retention(RUNTIME)
        public @interface ConstructorResult {
          Class<?> targetClass();
          ColumnResult[] columns() default {};
          EntityResult[] entities() default {};
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="column" type="orm:column-result"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="entity" type="orm:entity-result"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="target-class" type="xsd:string" use="required"/>
  </xsd:complexType>
```

```
<!-- ***************************************************** -->

  <xsd:complexType name="convert">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Repeatable(Converts.class)
        @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Convert {
          Class<? extends AttributeConverter> converter() default AttributeConverter.class;
          String attributeName() default "";
          boolean disableConversion() default false;
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="converter" type="xsd:string"/>
    <xsd:attribute name="attribute-name" type="xsd:string"/>
    <xsd:attribute name="disable-conversion" type="xsd:boolean"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="converter">
    <xsd:annotation>
      <xsd:documentation>

        @Target({TYPE}) @Retention(RUNTIME)
        public @interface Converter {
          boolean autoApply() default false;
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="class" type="xsd:string" use="required"/>
    <xsd:attribute name="auto-apply" type="xsd:boolean"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="discriminator-column">
    <xsd:annotation>
      <xsd:documentation>

        @Target({TYPE}) @Retention(RUNTIME)
        public @interface DiscriminatorColumn {
          String name() default "DTYPE";
          DiscriminatorType discriminatorType() default STRING;
          String columnDefinition() default "";
          String options() default "";
          int length() default 31;
          String comment() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="comment" type="xsd:string" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="discriminator-type" type="orm:discriminator-type"/>
```

```
    <xsd:attribute name="column-definition" type="xsd:string"/>
    <xsd:attribute name="options" type="xsd:string"/>
    <xsd:attribute name="length" type="xsd:int"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:simpleType name="discriminator-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum DiscriminatorType { STRING, CHAR, INTEGER }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="STRING"/>
      <xsd:enumeration value="CHAR"/>
      <xsd:enumeration value="INTEGER"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- ************************************************** -->

  <xsd:simpleType name="discriminator-value">
    <xsd:annotation>
      <xsd:documentation>

        @Target({TYPE}) @Retention(RUNTIME)
        public @interface DiscriminatorValue {
          String value();
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>

<!-- ************************************************** -->

 <xsd:complexType name="element-collection">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface ElementCollection {
          Class<?> targetClass() default void.class;
          FetchType fetch() default FetchType.LAZY;
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="order-by" type="orm:order-by"
                     minOccurs="0"/>
        <xsd:element name="order-column" type="orm:order-column"
                     minOccurs="0"/>
      </xsd:choice>
      <xsd:choice>
        <xsd:element name="map-key" type="orm:map-key"
                     minOccurs="0"/>
        <xsd:sequence>
          <xsd:element name="map-key-class" type="orm:map-key-class"
                       minOccurs="0"/>
          <xsd:choice>
            <xsd:element name="map-key-temporal"
```

```
                        type="orm:temporal"
                        minOccurs="0"/>
          <xsd:element name="map-key-enumerated"
                        type="orm:enumerated"
                        minOccurs="0"/>
          <xsd:sequence>
             <xsd:element name="map-key-attribute-override"
                          type="orm:attribute-override"
                          minOccurs="0" maxOccurs="unbounded"/>
             <xsd:element name="map-key-convert" type="orm:convert"
                          minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:choice>
        <xsd:choice>
          <xsd:element name="map-key-column"
                        type="orm:map-key-column"
                        minOccurs="0"/>
          <xsd:sequence>
            <xsd:element name="map-key-join-column"
                          type="orm:map-key-join-column"
                          minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="map-key-foreign-key"
                          type="orm:foreign-key"
                          minOccurs="0"/>
          </xsd:sequence>
        </xsd:choice>
      </xsd:sequence>
    </xsd:choice>
    <xsd:choice>
      <xsd:sequence>
        <xsd:element name="column" type="orm:column" minOccurs="0"/>
        <xsd:choice>
          <xsd:element name="temporal"
                        type="orm:temporal"
                        minOccurs="0"/>
          <xsd:element name="enumerated"
                        type="orm:enumerated"
                        minOccurs="0"/>
          <xsd:element name="lob"
                        type="orm:lob"
                        minOccurs="0"/>
        </xsd:choice>
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name="attribute-override"
                      type="orm:attribute-override"
                      minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="association-override"
                      type="orm:association-override"
                      minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="convert" type="orm:convert"
                      minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:choice>
    <xsd:element name="collection-table" type="orm:collection-table"
                 minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="target-class" type="xsd:string"/>
  <xsd:attribute name="fetch" type="orm:fetch-type"/>
  <xsd:attribute name="access" type="orm:access-type"/>
</xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="embeddable">
    <xsd:annotation>
```

```
      <xsd:documentation>

        Defines the settings and mappings for embeddable objects. Is
        allowed to be sparsely populated and used in conjunction with
        the annotations. Alternatively, the metadata-complete attribute
        can be used to indicate that no annotations are to be processed
        in the class. If this is the case then the defaulting rules will
        be recursively applied.

        @Target({TYPE}) @Retention(RUNTIME)
        public @interface Embeddable {}

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="attributes" type="orm:embeddable-attributes"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="class" type="xsd:string" use="required"/>
    <xsd:attribute name="access" type="orm:access-type"/>
    <xsd:attribute name="metadata-complete" type="xsd:boolean"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="embeddable-attributes">
    <xsd:sequence>
      <xsd:element name="basic" type="orm:basic"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="many-to-one" type="orm:many-to-one"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="one-to-many" type="orm:one-to-many"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="one-to-one" type="orm:one-to-one"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="many-to-many" type="orm:many-to-many"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="element-collection" type="orm:element-collection"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="embedded" type="orm:embedded"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="transient" type="orm:transient"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <!-- ************************************************** -->

  <xsd:complexType name="embedded">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Embedded {}

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="attribute-override" type="orm:attribute-override"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="association-override"
                   type="orm:association-override"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="convert" type="orm:convert"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
```

```
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="access" type="orm:access-type"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="embedded-id">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface EmbeddedId {}

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="attribute-override" type="orm:attribute-override"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="access" type="orm:access-type"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="entity-listener">
    <xsd:annotation>
      <xsd:documentation>

        Defines an entity listener to be invoked at lifecycle events
        for the entities that list this listener.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="pre-merge" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PreMerge {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="pre-persist" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PrePersist {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="post-persist" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PostPersist {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="pre-remove" type="orm:lifecycle-callback" minOccurs="0">
```

```
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreRemove {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-remove" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostRemove {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-update" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreUpdate {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-update" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostUpdate {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-upsert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreUpsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-upsert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostUpsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-insert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreInsert {}

    </xsd:documentation>
```

```
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="post-insert" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PostInsert {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="pre-delete" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PreDelete {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="post-delete" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PostDelete {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="post-load" type="orm:lifecycle-callback" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            @Target({METHOD}) @Retention(RUNTIME)
            public @interface PostLoad {}

          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="class" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="entity-listeners">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({TYPE}) @Retention(RUNTIME)
        public @interface EntityListeners {
          Class<?>[] value();
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="entity-listener" type="orm:entity-listener"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

<!-- ***************************************************** -->
```

```
  <xsd:complexType name="entity-result">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target(METHOD) @Retention(RUNTIME)
        public @interface EntityResult {
          Class<?> entityClass();
          LockModeType lockMode() default LockModeType.NONE;
          FieldResult[] fields() default {};
          String discriminatorColumn() default "";
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="lock-mode" type="orm:lock-mode-type" minOccurs="0"/>
      <xsd:element name="field-result" type="orm:field-result"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="entity-class" type="xsd:string" use="required"/>
    <xsd:attribute name="discriminator-column" type="xsd:string"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:simpleType name="enum-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum EnumType { ORDINAL, STRING }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="ORDINAL"/>
      <xsd:enumeration value="STRING"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- ***************************************************** -->

  <xsd:simpleType name="enumerated">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Enumerated {
          EnumType value() default ORDINAL;
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="orm:enum-type"/>
  </xsd:simpleType>

<!-- ***************************************************** -->

  <xsd:simpleType name="fetch-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum FetchType { LAZY, EAGER }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="LAZY"/>
```

```
      <xsd:enumeration value="EAGER"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- ************************************************** -->

  <xsd:complexType name="field-result">
    <xsd:annotation>
      <xsd:documentation>

        @Target({}) @Retention(RUNTIME)
        public @interface FieldResult {
          String name();
          String column();
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="column" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="foreign-key">
    <xsd:annotation>
      <xsd:documentation>

        @Target({}) @Retention(RUNTIME)
        public @interface ForeignKey {
          String name() default "";
          ConstraintMode value() default CONSTRAINT;
          String foreign-key-definition() default "";
          String options() default "";
        }

        Note that the elements that embed the use of the annotation
        default this use as @ForeignKey(PROVIDER_DEFAULT).

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="constraint-mode" type="orm:constraint-mode"/>
    <xsd:attribute name="foreign-key-definition" type="xsd:string"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="generated-value">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface GeneratedValue {
          GenerationType strategy() default AUTO;
          String generator() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="strategy" type="orm:generation-type"/>
    <xsd:attribute name="generator" type="xsd:string"/>
  </xsd:complexType>
```

```
<!-- ***************************************************** -->

  <xsd:simpleType name="generation-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum GenerationType { TABLE, SEQUENCE, IDENTITY, UUID, AUTO }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="TABLE"/>
      <xsd:enumeration value="SEQUENCE"/>
      <xsd:enumeration value="IDENTITY"/>
      <xsd:enumeration value="UUID"/>
      <xsd:enumeration value="AUTO"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- ***************************************************** -->

  <xsd:complexType name="id">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Id {}

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="column" type="orm:column"
                   minOccurs="0"/>
      <xsd:element name="generated-value" type="orm:generated-value"
                   minOccurs="0"/>
      <xsd:element name="temporal" type="orm:temporal"
                   minOccurs="0"/>
      <xsd:element name="table-generator" type="orm:table-generator"
                   minOccurs="0"/>
      <xsd:element name="sequence-generator" type="orm:sequence-generator"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="access" type="orm:access-type"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="id-class">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({TYPE}) @Retention(RUNTIME)
        public @interface IdClass {
          Class<?> value();
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="class" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="index">
    <xsd:annotation>
```

```
      <xsd:documentation>

        @Target({}) @Retention(RUNTIME)
        public @interface Index {
          String name() default "";
          String columnList();
          boolean unique() default false;
          String type() default "";
          String using() default "";
          String options() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="column-list" type="xsd:string" use="required"/>
    <xsd:attribute name="unique" type="xsd:boolean"/>
    <xsd:attribute name="type" type="xsd:string"/>
    <xsd:attribute name="using" type="xsd:string"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="inheritance">
    <xsd:annotation>
      <xsd:documentation>

        @Target({TYPE}) @Retention(RUNTIME)
        public @interface Inheritance {
          InheritanceType strategy() default InheritanceType.SINGLE_TABLE;
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="strategy" type="orm:inheritance-type"/>
  </xsd:complexType>

  <!-- ************************************************** -->

  <xsd:simpleType name="inheritance-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum InheritanceType { SINGLE_TABLE, TABLE_PER_CLASS, JOINED }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="SINGLE_TABLE"/>
      <xsd:enumeration value="JOINED"/>
      <xsd:enumeration value="TABLE_PER_CLASS"/>
    </xsd:restriction>
  </xsd:simpleType>

<!-- ************************************************** -->

  <xsd:complexType name="join-column">
    <xsd:annotation>
      <xsd:documentation>

        @Repeatable(JoinColumns.class)
        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface JoinColumn {
```

```
        String name() default "";
        String referencedColumnName() default "";
        boolean unique() default false;
        boolean nullable() default true;
        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String options() default "";
        String table() default "";
        ForeignKey foreignKey() default @ForeignKey();
        CheckConstraint[] check() default {};
        String comment() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="comment" type="xsd:string" minOccurs="0" />
    <xsd:element name="foreign-key" type="orm:foreign-key"
                 minOccurs="0"/>
    <xsd:element name="check-constraint" type="orm:check-constraint"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="referenced-column-name" type="xsd:string"/>
  <xsd:attribute name="unique" type="xsd:boolean"/>
  <xsd:attribute name="nullable" type="xsd:boolean"/>
  <xsd:attribute name="insertable" type="xsd:boolean"/>
  <xsd:attribute name="updatable" type="xsd:boolean"/>
  <xsd:attribute name="column-definition" type="xsd:string"/>
  <xsd:attribute name="options" type="xsd:string"/>
  <xsd:attribute name="table" type="xsd:string"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="join-table">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface JoinTable {
        String name() default "";
        String catalog() default "";
        String schema() default "";
        JoinColumn[] joinColumns() default {};
        JoinColumn[] inverseJoinColumns() default {};
        ForeignKey foreignKey() default @ForeignKey(ConstraintMode.PROVIDER_DEFAULT);
        ForeignKey inverseForeignKey() default @ForeignKey(ConstraintMode.PROVIDER_DEFAULT);
        UniqueConstraint[] uniqueConstraints() default {};
        Index[] indexes() default {};
        CheckConstraint[] check() default {};
        String comment() default "";
        String type() default "";
        String options() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="comment" type="xsd:string" minOccurs="0" />
    <xsd:sequence>
      <xsd:element name="join-column" type="orm:join-column"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="foreign-key" type="orm:foreign-key"
                   minOccurs="0"/>
    </xsd:sequence>
```

```
      <xsd:sequence>
        <xsd:element name="inverse-join-column" type="orm:join-column"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="inverse-foreign-key" type="orm:foreign-key"
                     minOccurs="0"/>
      </xsd:sequence>
      <xsd:element name="unique-constraint" type="orm:unique-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="index" type="orm:index"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="check-constraint" type="orm:check-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="catalog" type="xsd:string"/>
    <xsd:attribute name="schema" type="xsd:string"/>
    <xsd:attribute name="type" type="xsd:string"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="lob">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Lob {}

      </xsd:documentation>
    </xsd:annotation>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:simpleType name="lock-mode-type">
    <xsd:annotation>
      <xsd:documentation>

        public enum LockModeType implements FindOption, RefreshOption { READ, WRITE, OPTIMISTIC,
OPTIMISTIC_FORCE_INCREMENT, PESSIMISTIC_READ, PESSIMISTIC_WRITE, PESSIMISTIC_FORCE_INCREMENT, NONE}

      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="READ"/>
      <xsd:enumeration value="WRITE"/>
      <xsd:enumeration value="OPTIMISTIC"/>
      <xsd:enumeration value="OPTIMISTIC_FORCE_INCREMENT"/>
      <xsd:enumeration value="PESSIMISTIC_READ"/>
      <xsd:enumeration value="PESSIMISTIC_WRITE"/>
      <xsd:enumeration value="PESSIMISTIC_FORCE_INCREMENT"/>
      <xsd:enumeration value="NONE"/>

    </xsd:restriction>
  </xsd:simpleType>

  <!-- ***************************************************** -->

  <xsd:simpleType name="pessimistic-lock-scope">
    <xsd:annotation>
      <xsd:documentation>

        public enum PessimisticLockScope implements FindOption, RefreshOption, LockOption { NORMAL, EXTENDED,
FETCHED }

      </xsd:documentation>
```

```
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="NORMAL"/>
      <xsd:enumeration value="EXTENDED"/>
      <xsd:enumeration value="FETCHED"/>

    </xsd:restriction>
  </xsd:simpleType>

  <!-- ************************************************** -->

<xsd:complexType name="many-to-many">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface ManyToMany {
          Class<?> targetEntity() default void.class;
          CascadeType[] cascade() default {};
          FetchType fetch() default FetchType.LAZY;
          String mappedBy() default "";
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="order-by" type="orm:order-by"
                   minOccurs="0"/>
        <xsd:element name="order-column" type="orm:order-column"
                   minOccurs="0"/>
      </xsd:choice>
      <xsd:choice>
        <xsd:element name="map-key" type="orm:map-key"
                   minOccurs="0"/>
        <xsd:sequence>
          <xsd:element name="map-key-class" type="orm:map-key-class"
                   minOccurs="0"/>
          <xsd:choice>
            <xsd:element name="map-key-temporal"
                         type="orm:temporal"
                         minOccurs="0"/>
            <xsd:element name="map-key-enumerated"
                         type="orm:enumerated"
                         minOccurs="0"/>
            <xsd:sequence>
               <xsd:element name="map-key-attribute-override"
                            type="orm:attribute-override"
                            minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="map-key-convert" type="orm:convert"
                            minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:choice>
          <xsd:choice>
            <xsd:element name="map-key-column" type="orm:map-key-column"
                   minOccurs="0"/>
            <xsd:sequence>
              <xsd:element name="map-key-join-column"
                     type="orm:map-key-join-column"
                     minOccurs="0" maxOccurs="unbounded"/>
              <xsd:element name="map-key-foreign-key"
                           type="orm:foreign-key"
                           minOccurs="0"/>
            </xsd:sequence>
          </xsd:choice>
        </xsd:sequence>
      </xsd:choice>
```

```
      <xsd:element name="join-table" type="orm:join-table"
                   minOccurs="0"/>
      <xsd:element name="cascade" type="orm:cascade-type"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="target-entity" type="xsd:string"/>
    <xsd:attribute name="fetch" type="orm:fetch-type"/>
    <xsd:attribute name="access" type="orm:access-type"/>
    <xsd:attribute name="mapped-by" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="many-to-one">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface ManyToOne {
          Class<?> targetEntity() default void.class;
          CascadeType[] cascade() default {};
          FetchType fetch() default FetchType.EAGER;
          boolean optional() default true;
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="join-column" type="orm:join-column"
                       minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="foreign-key" type="orm:foreign-key"
                       minOccurs="0"/>
        </xsd:sequence>
        <xsd:element name="join-table" type="orm:join-table"
                     minOccurs="0"/>
      </xsd:choice>
      <xsd:element name="cascade" type="orm:cascade-type"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="target-entity" type="xsd:string"/>
    <xsd:attribute name="fetch" type="orm:fetch-type"/>
    <xsd:attribute name="optional" type="xsd:boolean"/>
    <xsd:attribute name="access" type="orm:access-type"/>
    <xsd:attribute name="maps-id" type="xsd:string"/>
    <xsd:attribute name="id" type="xsd:boolean"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="map-key">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface MapKey {
          String value() default "";
          @Deprecated
          String name() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <!-- no need to reflect annotation member name change here -->
```

```
    <xsd:attribute name="name" type="xsd:string"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="map-key-class">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface MapKeyClass {
          Class<?> value();
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="class" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="map-key-column">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface MapKeyColumn {
          String name() default "";
          boolean unique() default false;
          boolean nullable() default false;
          boolean insertable() default true;
          boolean updatable() default true;
          String columnDefinition() default "";
          String options() default "";
          String table() default "";
          int length() default 255;
          int precision() default 0; // decimal precision
          int scale() default 0; // decimal scale
          CheckConstraint[] check() default {};
          String comment() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="comment" type="xsd:string" minOccurs="0" />
      <xsd:element name="check-constraint" type="orm:check-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="unique" type="xsd:boolean"/>
    <xsd:attribute name="nullable" type="xsd:boolean"/>
    <xsd:attribute name="insertable" type="xsd:boolean"/>
    <xsd:attribute name="updatable" type="xsd:boolean"/>
    <xsd:attribute name="column-definition" type="xsd:string"/>
    <xsd:attribute name="options" type="xsd:string"/>
    <xsd:attribute name="table" type="xsd:string"/>
    <xsd:attribute name="length" type="xsd:int"/>
    <xsd:attribute name="precision" type="xsd:int"/>
    <xsd:attribute name="scale" type="xsd:int"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="map-key-join-column">
    <xsd:annotation>
      <xsd:documentation>
```

```
      @Repeatable(MapKeyJoinColumns.class)
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface MapKeyJoinColumn {
        String name() default "";
        String referencedColumnName() default "";
        boolean unique() default false;
        boolean nullable() default false;
        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String options() default "";
        String table() default "";
        ForeignKey foreignKey() default @ForeignKey(ConstraintMode.PROVIDER_DEFAULT);
        CheckConstraint[] check() default {};
        String comment() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="comment" type="xsd:string" minOccurs="0" />
    <xsd:element name="foreign-key" type="orm:foreign-key"
                 minOccurs="0"/>
    <xsd:element name="check-constraint" type="orm:check-constraint"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="referenced-column-name" type="xsd:string"/>
  <xsd:attribute name="unique" type="xsd:boolean"/>
  <xsd:attribute name="nullable" type="xsd:boolean"/>
  <xsd:attribute name="insertable" type="xsd:boolean"/>
  <xsd:attribute name="updatable" type="xsd:boolean"/>
  <xsd:attribute name="column-definition" type="xsd:string"/>
  <xsd:attribute name="options" type="xsd:string"/>
  <xsd:attribute name="table" type="xsd:string"/>
</xsd:complexType>


<!-- ***************************************************** -->

<xsd:complexType name="mapped-superclass">
  <xsd:annotation>
    <xsd:documentation>

      Defines the settings and mappings for a mapped superclass. Is
      allowed to be sparsely populated and used in conjunction with
      the annotations. Alternatively, the metadata-complete attribute
      can be used to indicate that no annotations are to be processed
      If this is the case then the defaulting rules will be recursively
      applied.

      @Target(TYPE) @Retention(RUNTIME)
      public @interface MappedSuperclass {}

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    <xsd:element name="id-class" type="orm:id-class" minOccurs="0"/>
    <xsd:element name="exclude-default-listeners" type="orm:emptyType"
                 minOccurs="0"/>
    <xsd:element name="exclude-superclass-listeners" type="orm:emptyType"
                 minOccurs="0"/>
    <xsd:element name="entity-listeners" type="orm:entity-listeners"
                 minOccurs="0"/>
    <xsd:element name="pre-merge" type="orm:lifecycle-callback" minOccurs="0">
```

```
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreMerge {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-persist" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PrePersist {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-persist" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostPersist {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-remove" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreRemove {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-remove" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostRemove {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-update" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreUpdate {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-update" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostUpdate {}

    </xsd:documentation>
```

```
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-upsert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreUpsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-upsert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostUpsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-insert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreInsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-insert" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostInsert {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="pre-delete" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreDelete {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-delete" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostDelete {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="post-load" type="orm:lifecycle-callback" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
```

```
          public @interface PostLoad {}

        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="attributes" type="orm:attributes" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="class" type="xsd:string" use="required"/>
  <xsd:attribute name="access" type="orm:access-type"/>
  <xsd:attribute name="metadata-complete" type="xsd:boolean"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="named-attribute-node">
  <xsd:annotation>
    <xsd:documentation>

      @Target({}) @Retention(RUNTIME)
      public @interface NamedAttributeNode {
        String value();
        String subgraph() default "";
        String keySubgraph() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="subgraph" type="xsd:string"/>
  <xsd:attribute name="key-subgraph" type="xsd:string"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="named-entity-graph">
  <xsd:annotation>
    <xsd:documentation>

      @Repeatable(NamedEntityGraphs.class)
      @Target({TYPE}) @Retention(RUNTIME)
      public @interface NamedEntityGraph {
        String name() default "";
        NamedAttributeNode[] attributeNodes() default {};
        boolean includeAllAttributes() default false;
        NamedSubgraph[] subgraphs() default {};
        NamedSubGraph[] subclassSubgraphs() default {};
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="named-attribute-node"
                 type="orm:named-attribute-node"
                 minOccurs="0"
                 maxOccurs="unbounded"/>
    <xsd:element name="subgraph"
                 type="orm:named-subgraph"
                 minOccurs="0"
                 maxOccurs="unbounded"/>
    <xsd:element name="subclass-subgraph"
                 type="orm:named-subgraph"
                 minOccurs="0"
                 maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="include-all-attributes" type="xsd:boolean"/>
</xsd:complexType>
```

```
<!-- ***************************************************** -->

  <xsd:complexType name="named-native-query">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Repeatable(NamedNativeQueries.class)
        @Target({TYPE}) @Retention(RUNTIME)
        public @interface NamedNativeQuery {
          String name();
          String query();
          QueryHint[] hints() default {};
          Class<?> resultClass() default void.class;
          String resultSetMapping() default ""; //named SqlResultSetMapping
          EntityResult[] entities() default {};
          ConstructorResult[] classes() default {};
          ColumnResult[] columns() default {};
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="query" type="xsd:string"/>
      <xsd:element name="hint" type="orm:query-hint"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="entity-result" type="orm:entity-result"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="constructor-result" type="orm:constructor-result"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="column-result" type="orm:column-result"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="result-class" type="xsd:string"/>
    <xsd:attribute name="result-set-mapping" type="xsd:string"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="named-query">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Repeatable(NamedQueries.class)
        @Target({TYPE}) @Retention(RUNTIME)
        public @interface NamedQuery {
          String name();
          String query();
          Class<?> resultClass() default void.class;
          LockModeType lockMode() default LockModeType.NONE;
          PessimisticLockScope lockScope() default PessimisticLockScope.NORMAL;
          QueryHint[] hints() default {};
          String entityGraph() default "";
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="query" type="xsd:string"/>
      <xsd:element name="lock-mode" type="orm:lock-mode-type" minOccurs="0"/>
      <xsd:element name="lock-scope" type="orm:pessimistic-lock-scope" minOccurs="0"/>
      <xsd:element name="hint" type="orm:query-hint"
                   minOccurs="0" maxOccurs="unbounded"/>
```

```
      <xsd:element name="entity-graph" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="result-class" type="xsd:string"/>
</xsd:complexType>

  <!-- ************************************************** -->

  <xsd:complexType name="named-native-statement">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Repeatable(NamedNativeStatements.class)
        @Target({TYPE}) @Retention(RUNTIME)
        public @interface NamedNativeStatement {
          String name();
          String statement();
          QueryHint[] hints() default {};
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="statement" type="xsd:string"/>
      <xsd:element name="hint" type="orm:query-hint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>

  <!-- ************************************************** -->

  <xsd:complexType name="named-statement">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Repeatable(NamedStatements.class)
        @Target({TYPE}) @Retention(RUNTIME)
        public @interface NamedStatement {
          String name();
          String statement();
          QueryHint[] hints() default {};
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="statement" type="xsd:string"/>
      <xsd:element name="hint" type="orm:query-hint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>

  <!-- ************************************************** -->

  <xsd:complexType name="named-stored-procedure-query">
    <xsd:annotation>
      <xsd:documentation>

        @Repeatable(NamedStoredProcedureQueries.class)
        @Target({TYPE}) @Retention(RUNTIME)
        public @interface NamedStoredProcedureQuery {
          String name();
          String procedureName();
```

```
          StoredProcedureParameter[] parameters() default {};
          Class[] resultClasses() default {};
          String[] resultSetMappings() default{};
          QueryHint[] hints() default {};
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="parameter"
                   type="orm:stored-procedure-parameter"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="result-class" type="xsd:string"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="result-set-mapping" type="xsd:string"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="hint" type="orm:query-hint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="procedure-name" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="named-subgraph">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({}) @Retention(RUNTIME)
        public @interface NamedSubgraph {
          String name();
          Class<?> type() default void.class;
          NamedAttributeNode[] attributeNodes();
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="named-attribute-node"
                   type="orm:named-attribute-node"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="class" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="one-to-many">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface OneToMany {
          Class<?> targetEntity() default void.class;
          CascadeType[] cascade() default {};
          FetchType fetch() default FetchType.LAZY;
          String mappedBy() default "";
          boolean orphanRemoval() default false;
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
```

```
      <xsd:choice>
        <xsd:element name="order-by" type="orm:order-by"
                   minOccurs="0"/>
        <xsd:element name="order-column" type="orm:order-column"
                   minOccurs="0"/>
      </xsd:choice>
      <xsd:choice>
        <xsd:element name="map-key" type="orm:map-key"
                   minOccurs="0"/>
        <xsd:sequence>
          <xsd:element name="map-key-class" type="orm:map-key-class"
                   minOccurs="0"/>
          <xsd:choice>
            <xsd:element name="map-key-temporal"
                         type="orm:temporal"
                         minOccurs="0"/>
            <xsd:element name="map-key-enumerated"
                         type="orm:enumerated"
                         minOccurs="0"/>
            <xsd:sequence>
               <xsd:element name="map-key-attribute-override"
                            type="orm:attribute-override"
                            minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="map-key-convert" type="orm:convert"
                            minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:choice>
          <xsd:choice>
            <xsd:element name="map-key-column" type="orm:map-key-column"
                   minOccurs="0"/>
            <xsd:sequence>
              <xsd:element name="map-key-join-column"
                           type="orm:map-key-join-column"
                           minOccurs="0" maxOccurs="unbounded"/>
              <xsd:element name="map-key-foreign-key"
                           type="orm:foreign-key"
                           minOccurs="0"/>
            </xsd:sequence>
          </xsd:choice>
        </xsd:sequence>
      </xsd:choice>
      <xsd:choice>
        <xsd:element name="join-table" type="orm:join-table"
                     minOccurs="0"/>
        <xsd:sequence>
          <xsd:element name="join-column" type="orm:join-column"
                       minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="foreign-key" type="orm:foreign-key"
                       minOccurs="0"/>
        </xsd:sequence>
      </xsd:choice>
      <xsd:element name="cascade" type="orm:cascade-type"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="target-entity" type="xsd:string"/>
    <xsd:attribute name="fetch" type="orm:fetch-type"/>
    <xsd:attribute name="access" type="orm:access-type"/>
    <xsd:attribute name="mapped-by" type="xsd:string"/>
    <xsd:attribute name="orphan-removal" type="xsd:boolean"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="one-to-one">
    <xsd:annotation>
      <xsd:documentation><![CDATA[
```

```
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface OneToOne {
        Class<?> targetEntity() default void.class;
        CascadeType[] cascade() default {};
        FetchType fetch() default FetchType.EAGER;
        boolean optional() default true;
        String mappedBy() default "";
        boolean orphanRemoval() default false;
      }

    ]]></xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:choice>
      <xsd:sequence>
        <xsd:element name="primary-key-join-column"
                     type="orm:primary-key-join-column"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="primary-key-foreign-key"
                     type="orm:foreign-key"
                     minOccurs="0"/>
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name="join-column" type="orm:join-column"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="foreign-key" type="orm:foreign-key"
                     minOccurs="0"/>
      </xsd:sequence>
      <xsd:element name="join-table" type="orm:join-table"
                   minOccurs="0"/>
    </xsd:choice>
    <xsd:element name="cascade" type="orm:cascade-type"
                 minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="target-entity" type="xsd:string"/>
  <xsd:attribute name="fetch" type="orm:fetch-type"/>
  <xsd:attribute name="optional" type="xsd:boolean"/>
  <xsd:attribute name="access" type="orm:access-type"/>
  <xsd:attribute name="mapped-by" type="xsd:string"/>
  <xsd:attribute name="orphan-removal" type="xsd:boolean"/>
  <xsd:attribute name="maps-id" type="xsd:string"/>
  <xsd:attribute name="id" type="xsd:boolean"/>
</xsd:complexType>

<!-- ***************************************************** -->

<xsd:simpleType name="order-by">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface OrderBy {
        String value() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<!-- ***************************************************** -->

<xsd:complexType name="order-column">
  <xsd:annotation>
    <xsd:documentation>
```

```
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface OrderColumn {
        String name() default "";
        boolean nullable() default true;
        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String options() default "";
        String comment() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="comment" type="xsd:string" minOccurs="0" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="nullable" type="xsd:boolean"/>
  <xsd:attribute name="insertable" type="xsd:boolean"/>
  <xsd:attribute name="updatable" type="xsd:boolean"/>
  <xsd:attribute name="column-definition" type="xsd:string"/>
  <xsd:attribute name="options" type="xsd:string"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:simpleType name="parameter-mode">
  <xsd:annotation>
    <xsd:documentation>

      public enum ParameterMode { IN, INOUT, OUT, REF_CURSOR }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="IN"/>
    <xsd:enumeration value="INOUT"/>
    <xsd:enumeration value="OUT"/>
    <xsd:enumeration value="REF_CURSOR"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- ************************************************** -->

<xsd:complexType name="lifecycle-callback">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ************************************************** -->

<xsd:complexType name="primary-key-join-column">
  <xsd:annotation>
    <xsd:documentation>

      @Repeatable(PrimaryKeyJoinColumns.class)
      @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
      public @interface PrimaryKeyJoinColumn {
        String name() default "";
        String referencedColumnName() default "";
        String columnDefinition() default "";
        String options() default "";
        ForeignKey foreignKey() default @ForeignKey(PROVIDER_DEFAULT);
        CheckConstraint[] check() default {};
```

```
        String comment() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="comment" type="xsd:string" minOccurs="0" />
    <xsd:element name="foreign-key" type="orm:foreign-key"
                 minOccurs="0"/>
    <xsd:element name="check-constraint" type="orm:check-constraint"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="referenced-column-name" type="xsd:string"/>
  <xsd:attribute name="column-definition" type="xsd:string"/>
  <xsd:attribute name="options" type="xsd:string"/>
</xsd:complexType>

<!-- ***************************************************** -->

<xsd:complexType name="query-hint">
  <xsd:annotation>
    <xsd:documentation>

      @Target({}) @Retention(RUNTIME)
      public @interface QueryHint {
        String name();
        String value();
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="value" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***************************************************** -->

<xsd:complexType name="secondary-table">
  <xsd:annotation>
    <xsd:documentation>

      @Repeatable(SecondaryTables.class)
      @Target({TYPE}) @Retention(RUNTIME)
      public @interface SecondaryTable {
        String name();
        String catalog() default "";
        String schema() default "";
        PrimaryKeyJoinColumn[] pkJoinColumns() default {};
        ForeignKey foreignKey() default @ForeignKey(ConstraintMode.PROVIDER_DEFAULT);
        UniqueConstraint[] uniqueConstraints() default {};
        Index[] indexes() default {};
        CheckConstraint[] check() default {};
        String comment() default "";
        String type() default "";
        String options() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:sequence>
      <xsd:element name="comment" type="xsd:string" minOccurs="0" />
      <xsd:element name="primary-key-join-column"
```

```
                    type="orm:primary-key-join-column"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="primary-key-foreign-key"
                    type="orm:foreign-key"
                    minOccurs="0"/>
      </xsd:sequence>
      <xsd:element name="unique-constraint" type="orm:unique-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="index" type="orm:index"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="foreign-key" type="orm:foreign-key"
                   minOccurs="0"/>
      <xsd:element name="check-constraint" type="orm:check-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="catalog" type="xsd:string"/>
    <xsd:attribute name="schema" type="xsd:string"/>
    <xsd:attribute name="type" type="xsd:string"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="sequence-generator">
    <xsd:annotation>
      <xsd:documentation>

        @Repeatable(SequenceGenerators.class)
        @Target({TYPE, METHOD, FIELD, PACKAGE}) @Retention(RUNTIME)
        public @interface SequenceGenerator {
          String name() default "";
          String sequenceName() default "";
          String catalog() default "";
          String schema() default "";
          int initialValue() default 1;
          int allocationSize() default 50;
          String options() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="sequence-name" type="xsd:string"/>
    <xsd:attribute name="catalog" type="xsd:string"/>
    <xsd:attribute name="schema" type="xsd:string"/>
    <xsd:attribute name="initial-value" type="xsd:int"/>
    <xsd:attribute name="allocation-size" type="xsd:int"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="sql-result-set-mapping">
    <xsd:annotation>
      <xsd:documentation>

        @Repeatable(SqlResultSetMappings.class)
        @Target({TYPE}) @Retention(RUNTIME)
        public @interface SqlResultSetMapping {
          String name();
          EntityResult[] entities() default {};
          ConstructorResult[] classes() default{};
          ColumnResult[] columns() default {};
```

```
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="entity-result" type="orm:entity-result"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="constructor-result" type="orm:constructor-result"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="column-result" type="orm:column-result"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="stored-procedure-parameter">
    <xsd:annotation>
      <xsd:documentation><![CDATA[

        @Target({}) @Retention(RUNTIME)
        public @interface StoredProcedureParameter {
          String name() default "";
          ParameterMode mode() default ParameterMode.IN;
          Class<?> type();
        }

      ]]></xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string"
                   minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="class" type="xsd:string" use="required"/>
    <xsd:attribute name="mode" type="orm:parameter-mode"/>
  </xsd:complexType>

<!-- ***************************************************** -->

  <xsd:complexType name="table">
    <xsd:annotation>
      <xsd:documentation>

        @Target({TYPE}) @Retention(RUNTIME)
        public @interface Table {
          String name() default "";
          String catalog() default "";
          String schema() default "";
          UniqueConstraint[] uniqueConstraints() default {};
          Index[] indexes() default {};
          CheckConstraint[] check() default {};
          String comment() default "";
          String type() default "";
          String options() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="comment" type="xsd:string" minOccurs="0" />
      <xsd:element name="unique-constraint" type="orm:unique-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="index" type="orm:index"
                   minOccurs="0" maxOccurs="unbounded"/>
```

```
      <xsd:element name="check-constraint" type="orm:check-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="catalog" type="xsd:string"/>
    <xsd:attribute name="schema" type="xsd:string"/>
    <xsd:attribute name="type" type="xsd:string"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="table-generator">
    <xsd:annotation>
      <xsd:documentation>

        @Repeatable(TableGenerators.class)
        @Target({TYPE, METHOD, FIELD, PACKAGE}) @Retention(RUNTIME)
        public @interface TableGenerator {
          String name() default "";
          String table() default "";
          String catalog() default "";
          String schema() default "";
          String pkColumnName() default "";
          String valueColumnName() default "";
          String pkColumnValue() default "";
          int initialValue() default 0;
          int allocationSize() default 50;
          UniqueConstraint[] uniqueConstraints() default {};
          Indexes[] indexes() default {};
          String options() default "";
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" minOccurs="0"/>
      <xsd:element name="unique-constraint" type="orm:unique-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="index" type="orm:index"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="check-constraint" type="orm:check-constraint"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="table" type="xsd:string"/>
    <xsd:attribute name="catalog" type="xsd:string"/>
    <xsd:attribute name="schema" type="xsd:string"/>
    <xsd:attribute name="pk-column-name" type="xsd:string"/>
    <xsd:attribute name="value-column-name" type="xsd:string"/>
    <xsd:attribute name="pk-column-value" type="xsd:string"/>
    <xsd:attribute name="initial-value" type="xsd:int"/>
    <xsd:attribute name="allocation-size" type="xsd:int"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:simpleType name="temporal">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Temporal {
          TemporalType value();
        }
```

```
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="orm:temporal-type"/>
  </xsd:simpleType>

  <!-- ************************************************** -->

  <xsd:simpleType name="temporal-type">
    <xsd:annotation>
      <xsd:documentation>

        @Deprecated(since = "3.2")
        public enum TemporalType {
          DATE, // java.sql.Date
          TIME, // java.sql.Time
          TIMESTAMP // java.sql.Timestamp
        }

      </xsd:documentation>
    </xsd:annotation>
      <xsd:restriction base="xsd:token">
        <xsd:enumeration value="DATE"/>
        <xsd:enumeration value="TIME"/>
        <xsd:enumeration value="TIMESTAMP"/>
     </xsd:restriction>
  </xsd:simpleType>

    <!-- ************************************************** -->

    <xsd:complexType name="excluded-from-versioning">
        <xsd:annotation>
            <xsd:documentation>

                @Target({METHOD, FIELD})
                @Retention(RUNTIME)
                public @interface ExcludedFromVersioning {}

            </xsd:documentation>
        </xsd:annotation>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>

    <!-- ************************************************** -->

  <xsd:complexType name="transient">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Transient {}

      </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="unique-constraint">
    <xsd:annotation>
      <xsd:documentation>

        @Target({}) @Retention(RUNTIME)
        public @interface UniqueConstraint {
          String name() default "";
          String[] columnNames();
          String options() default "";
```

```
        }

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="column-name" type="xsd:string"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="options" type="xsd:string"/>
  </xsd:complexType>

<!-- ************************************************** -->

  <xsd:complexType name="version">
    <xsd:annotation>
      <xsd:documentation>

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface Version {}

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="column" type="orm:column" minOccurs="0"/>
      <xsd:element name="temporal" type="orm:temporal" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="access" type="orm:access-type"/>
  </xsd:complexType>

</xsd:schema>
```

Related Documents

- [1] Jakarta Persistence, v. 2.2. *https://jakarta.ee/specifications/persistence/2.2/*.
- [2] SQL 2003, Part 2, Foundation (SQL/Foundation). ISO/IEC 9075-2:2003.
- [3] JDBC 4.3 Specification. http://jcp.org/en/jsr/detail?id=221.
- [4] Enterprise JavaBeans, v. 2.1.
- [5] Jakarta Validation, v. 4.0. *https://jakarta.ee/specifications/bean-validation/4.0/*.
- [6] Jakarta EE Platform, v. 11.0. *https://jakarta.ee/specifications/platform/11/*.
- [7] Jakarta Contexts and Dependency Injection, v. 4.1. *https://jakarta.ee/specifications/cdi/4.1/*.
- [8] Jakarta Transactions, v. 2.0. *https://jakarta.ee/specifications/transactions/2.0/*.
- [9] Jakarta Data, v. 1.0. *https://jakarta.ee/specifications/data/1.0/*.

Appendix A: Revision History

This appendix lists the significant changes that have been made during the development of the Jakarta Persistence specification.

A.1. Jakarta Persistence 4.0

A.1.1. Changes to API

Added `EntityAgent` and `EntityHandler`

Added `@JakartaQuery`, `@NativeQuery`, `@QueryOptions`, and `StaticStatementReference` for declaring static queries at the method level, along with additional members of `TypedQueryReference`

Added a new API for programmatic SQL result set mappings

Added `Statement`, `CriteriaStatement`, `@NamedStatement` and `@NamedNativeStatement` to clearly distinguish statements which modify data from queries which return data; added `ofType()`, `withEntityGraph()`, and `asStatement()` to `Query`; added `StatementOrTypedQuery`

Added `@Fetch` for more elegant definition of named entity graph nodes and subgraphs, and for specifying per-attribute fetch options

Added the ability to select `LAZY` fetching as the default for `@OneToOne` and `@ManyToOne` associations via `<default-to-one-fetch-type>` in `persistence.xml` or `PersistenceConfiguration.defaultToOneFetchType()`

Added specialized expression types `ComparableExpression`, `NumericExpression`, `TextExpression`, `TemporalExpression`, `BooleanExpression` to criteria API, along with matching attribute types `ComparableAttribute`, `NumericAttribute`, `TextAttribute`, `TemporalAttribute`, `BooleanAttribute` to metamodel API

Added `get()` operation as an alternative to `find()`; added `findMultiple()` and `getMultiple()`

Added `getResultCount()` to `TypedQuery`

Added `registerResultParameter()` to `StoredProcedureQuery`

For much improved type safety, overloaded `getResultList()`, `getSingleResult()`, and `getSingleResultOrNull()` on `StoredProcedureQuery`; added `registerParameter()`; and added an overload of `getOutputParameterValue()`

Made `StoredProcedureQuery` a subtype of `AutoCloseable`

Added `@ExcludedFromVersioning`

Added `FETCHED` to `PessimisticLockScope`, `lockScope` to `@NamedQuery`, and `setLockScope()` to `Query`

Added new lifecycle callback types: `@PreMerge`, `@PreInsert`, `@PostInsert`, `@PreUpsert`, `@PostUpsert`, `@PreDelete`, `@PostDelete`

Added `addListener()` to `EntityManagerFactory`

Added `@EntityListener` annotation

Allow `@EntityListeners` on a package descriptor or module descriptor

Specify that an entity listener may have multiple callbacks of the same type

Added `FetchOption` and `AttributeNode.addOption()`

Added `BatchSize`

Added `withEntityGraph()` to `StatementOrTypedQuery`, `entityGraph` to `@NamedQuery`, and overloaded `createQuery(String, EntityGraph)` on `EntityHandler`.

Specified that the graph returned by `getEntityGraph()` is now mutable, and added `getEntityGraph(Class, String)`

Added `createEntityGraph()` and `getNamedEntityGraphs()` to `EntityType`

Added `Graph.getGraphedType()` and `AttributeNode.getAttribute()`

Added `addSubgraph()` and `addTreatedSubgraph()` to `AttributeNode`

Overloaded `addNamedQuery()` to take `TypedQuery` and return `TypedQueryReference` and added `addNamedStatement()`

Added overloads of `setParameter()` which accept a disambiguating `Class` or `Type`; added `setConvertedParameter()` to `Query` and `registerConvertedParameter()` to `StoredProcedureQuery`; added `convertedParameter()` to `CriteriaBuilder`

Allow named queries to be declared by a module or package descriptor

Improved `@Index` and `@Table` to allow more customization of the generated DDL

Added operations to `CriteriaBuilder` accepting JPQL query strings, returning corresponding criteria objects

Added `augment()` methods to `CriteriaBuilder` for use in Jakarta Data implementations

Added additional overload of `CriteriaBuilder.between()`

Added `QueryFlushMode`, `Query.setQueryFlushMode()`, and `flush` to `@NamedQuery` and `@NamedNativeQuery`

Added `FlushModeType.EXPLICIT`

Added specialized `Option` types to `EntityManager`, `EntityAgent`, `TypedQuery`, `Statement`, `StoredProcedureQuery`, and `CreationOption` types to `EntityManager` and `EntityAgent`

A.1.2. Changes to Query Language

Made `select new X(…)` optional in JPQL queries with an explicit result class of type `X`, and allowed the result class of a native SQL query to be interpreted as a constructor result

Specify that entity names must be legal Java identifiers

Made `ELSE` branch in `CASE` expressions optional in JPQL

A.1.3. Changes to Configuration and Bootstrap

Added `@PersistenceAgent` for injection of Jakarta EE container-managed `EntityAgent`

Allow named queries and statements to be defined on non-managed classes and interfaces

Overloaded `Persistence.generateSchema()` to accept `PersistenceConfiguration` and added related functionality to `PersistenceConfiguration`: `exportSchema()`, `schemaManagementDatabaseAction()`, `schemaManagementScriptsAction()`, `SchemaManagementAction`

Added `jakarta.persistence.jdbc.fetchSize` and `jakarta.persistence.jdbc.batchSize`, and promoted schema export specific properties `jakarta.persistence.database-*` to API

Added static lists of standard property names to inner interfaces of `Persistence`

Fixed `SCHEMAGEN_CREATE_TARGET`, `SCHEMAGEN_DROP_TARGET`, and added `SCHEMAGEN_LOAD_SCRIPT_SOURCE` to `PersistenceConfiguration`

Added `SchemaManager.populate()` and the new schema generation action `populate()`

Redefine automatic bean validation. By default, automatic validation is now expected after pre-insert, pre-update and pre-upsert lifecycle events instead of previously used pre-persist, pre-update ones. Automatic bean validation can be enabled after the following lifecycle events: pre-persist, pre-merge, pre-remove, pre-delete.

Added `@Discoverable` annotation

Introduced the requirement to derive additional schema metadata from Jakarta Validation constraints

A.1.4. Minor Changes and Fixes

Relaxed requirement that exceptions validating method arguments cause the transaction to be marked for rollback

Added `jakarta.annotation.Nullable` and `jakarta.annotation.Nonnull` to APIs

`jakarta.annotation.Nonnull` on an entity attribute now implies `optional=false`

Allow `@EntityResult` as a subelement of `@ConstructorResult` in named SQL result set mappings

Allow `value` member of `@JoinColumns` to be defaulted

Added `comment` member to `@DiscriminatorColumn`, `@MapKeyColumn`, `@MapKeyJoinColumn`, `@PrimaryKeyJoinColumn` and `@CollectionTable`

Added `check` member to `@MapKeyColumn`, `@MapKeyJoinColumn`, `@PrimaryKeyJoinColumn` and `@CollectionTable`

Change default `lockMode` of `EntityResult` to `NONE`

Fixed some incorrect generic method signatures in `From`

Fixed use of raw types in `AttributeNode`

Added `PersistenceUnitInfo.getAllManagedClassNames()` and `PersistenceProvider.getClassTransformer()`

Redefined semantics of `EntityExistsException`

Clarified that null collection-valued attributes are interpreted as empty collections

Specified that changes to an owned collection should result in an entity lifecycle callback

Clarify behavior with duplicate persistent field names in entity hierarchy

Clarified usage of `@OrderColumn` and `@MapKeyColumn`

Added additional overloads of operations accepting restriction lists in criteria API

Deleted `CriteriaBuilder.keys()`, `CriteriaBuilder.values()`

A.1.5. Deprecations

Deprecated many operations of `Query`, since those operations should now be executed via a `Statement` or `TypedQuery`

Deprecated `createEntityGraph(String)`

Deprecated `EntityManager.getDelegate()` in favor of `unwrap()`

Deprecated `name` member of `@MapKey` in favor of `value`

A.1.6. Removals

Deleted APIs marked deprecated for removal

Removed redundant overloads of `find()` and `refresh()` accepting `LockModeType`

Removed deprecated support for use of `Byte[]` and `Character[]` as basic types

A.2. Jakarta Persistence 3.2

Added support for Java record types as embeddable classes

Added support for `java.time.Instant` and `java.time.Year` and clarified JDBC mappings for basic types

Added `union`, `intersect`, `except`, `cast`, `left`, `right`, and `replace` for Jakarta Persistence QL and criteria queries

Added `||` string concatenation operator and `id` and `version` functions to Jakarta Persistence QL

Added `CriteriaSelect` to the Criteria API

Added `extract()` to `CriteriaBuilder`

Added `subquery(EntityType)` to `CommonAbstractCriteria`

Added support for specifying null precedence when ordering Jakarta Persistence QL and criteria queries

Added `getSingleResultOrNull()` to `Query`, `TypedQuery`, `StoredProcedureQuery`

Added `entities()`, `classes()` and `columns()` to `NamedNativeQuery`

Added `lockMode()` to `EntityResult` with the default being `OPTIMISTIC`

Added `getVersion()`, `isLoaded()`, `load()`, `isInstance()` and `getClass()` methods to `PersistenceUnitUtil`

Added overload of `entity()` accepting an entity name to `Metamodel`

Added `javax.annotation.processing.Generated` to the list of defined annotations on `StaticMetamodel`

Added joins on `EntityTypes`

Added constants for managed types, named queries, named graphs and named result set mappings to generated `StaticMetamodel`

Added `LocalDateTime` and `Instant` to supported `Version` types

Added `where()`, `having()`, `and()`, `or()`, `array()`, `tuple()` overloads accepting `List` to `CriteriaQuery` and `CriteriaBuilder`

Added `equalTo()` and `notEqualTo()` to `Expression`

Added `concat()` overload accepting list of expressions to `CriteriaBuilder`

Added `Graph` interface as parent of `EntityGraph` and `Subgraph` and moved common operations there

Added `addAttributeNode()`, `removeAttributeNode()`, `addTreatedSubgraph()`, `addElementSubgraph()`, `addTreatedElementSubgraph()`, `addMapKeySubgraph()`, and `addTreatedMapKeySubgraph()` methods to `Graph`

Added `find()`, `refresh()`, `lock()` overloads to `EntityManager` taking newly introduced `FindOptions`, `RefreshOptions`, and `LockOptions` respectively

Added `setCacheStoreMode()`, and `setCacheRetreiveMode()` methods to `EntityManager` and `Query`

Added `getReference` overload to `EntityManager`

Added `runWithConnection()` and `callWithConnection()` to `EntityManager`

Added `runInTransaction()` and `callInTransaction()` to `EntityManagerFactory`

Added `getName()` to `EntityManagerFactory`

Added `setTimeout(Integer)` and `getTimeout()` to `EntityTransaction`

Added programmatic API to obtain `EntityManagerFactory` using `PersistenceConfiguration`

Added constants for properties defined by the specification to the `PersistenceConfiguration`

Added `SchemaManager` API

Added options member to elements which result in DDL generation

Added `EnumeratedValue` allowing custom mapping of fields of Java enums

Added `comment` and `check` members to `@Table` and `@Column` annotations, along with `@CheckConstraint`

Added `secondPrecision` to `@Column` annotation and clarified semantics of `@Column` members

Added factory-level access to named queries and named entity graphs, along with `TypedQueryReference`

Added integration points for dependency injection

Allowed scalar expressions in the `ORDER BY` clause in Jakarta Persistence QL

Made the `name` member of `TableGenerator` and `SequenceGenerator` optional and allow usage of these annotations at the java package level

Made identification variables and the `SELECT` clause in Jakarta Persistence QL optional

Clarified the primary key types supported for each `GenerationType`

Clarified availability of `SEQUENCE`, `TABLE` and `UUID` generated IDs on `PrePersist`

Clarified semantics of numeric literals and numeric type promotions, and added support for `bi` and `bd` suffixes

Clarified semantics of `Convert`/`Converter` annotations

Clarified rules around distinction of entity names and identification variables and case-sensitivity in Jakarta Persistence QL queries

Clarified the semantics of `Bindable.ENTITY_TYPE` in javadoc

Clarified the semantics of collection-valued query parameters

Entity and embeddable classes may now be static inner classes

Primary key classes are no longer required to be public and serializable

Pulled `getParameters()` up from `CriteriaQuery` to `CommonAbstractCriteria`

Updated persistence and object/relational mapping schemas

Fixed wildcard types in `addSubgraph` and `addAttributeNode` in `Graph`

Fixed lower type bounds to the `Path.get` entity argument `X`

Fixed example code in the javadoc of `AttributeOverrides`

Fixed `maxOccurs` of the version element in ORM schema to allow at most one element

Partially fixed raw types warnings through the API

Improved AsciiDoc formatting and fixed typos through the specification document

A.2.1. Deprecations

Deprecated usage of `Calendar`, `Date`, `Time`, `Timestamp`, `Temporal`, `MapKeyTemporal` and `TemporalType` in new applications in favour of `java.time` API

Deprecated use of `Byte[]` and `Character[]` arrays types for basic attributes, in favor of primitive array types

Deprecated `multiselect` methods in `CriteriaQuery`. The preference is to use `array` or `tuple` method defined in `CriteriaBuilder`

A.2.2. Deprecations for removal

Deprecated `addSubclassSubgraph()` in `EntityGraph` for removal; `addTreatedSubgraph()` method should be used as direct replacement

Deprecated `addSubgraph(Attribute, Class)` and `addKeySubgraph()` in `Graph`/`EntityGraph`/`SubGraph` for removal; `addTreatedSubgraph(Attribute, Class)` and `addMapKeySubgraph()` methods should be used as direct replacements

Deprecated `jakarta.persistence.spi.PersistenceUnitTransactionType` for removal; `jakarta.persistence.PersistenceUnitTransactionType` methods should be used as direct replacement

Deprecated default public no-arg constructor in `jakarta.persistence.Persistence` and `PERSISTENCE_PROVIDER` and `providers` fields in this class for removal with no replacement. This class is not designed for extensibility

A.3. Jakarta Persistence 3.1

`EntityManagerFactory` and `EntityManager` interfaces extend `java.lang.AutoCloseable` interface

Fixed `ClassTransformer.transform` to throw Persistence API specific exception

Added support for `java.util.UUID` and `GenerationType.UUID`

Added `CEILING`, `EXP`, `FLOOR`, `LN`, `POWER`, `ROUND`, and `SIGN` numeric functions to Jakarta Persistence QL and `ceiling()`, `exp()`, `floor()`, `ln()`, `power()`, `round()`, and `sign()` to Criteria API

Added `LOCAL DATE`, `LOCAL DATETIME`, and `LOCAL TIME` functions to Jakarta Persistence QL and corresponding `localDate()`, `localDateTime()`, and `localTime()` to Criteria API

Added `EXTRACT` function to Jakarta Persistence QL

Added support for `Expressions` as conditions in `Criteria CASE` expressions

Clarified mixing types of query input parameters

Added missing definition of `single_valued_embeddable_object_field` in Jakarta Persistence QL BNF

Clarified definition of the `Basic` type

Clarified the order of parameters in the `LOCATE` function

Clarified `SqlResultSetMapping` with multiple `EntityResults` and conflicting aliases

A.4. Jakarta Persistence 3.0

Created document from Java Persistence 2.2 Final Release specification.

The document was converted to `Asciidoc` format.

Packages of all API classes were changed to `jakarta.persistence`. These changes are reflected in the specification document.

Schema namespaces were changed from http://xmlns.jcp.org/xml/ns/persistence and http://xmlns.jcp.org/xml/ns/persistence/orm to https://jakarta.ee/xml/ns/persistence and https://jakarta.ee/xml/ns/persistence/orm

References to schema versions lower than 2.2 were removed.

A.5. Java Persistence 2.2 (Maintenance Release Draft)

Created document from Java Persistence 2.1 Final Release specification.

The following annotations have been marked `@Repeatable`:

- `AssociationOverride`
- `AttributeOverride`
- `Convert`
- `JoinColumn`
- `MapKeyJoinColumn`
- `NamedEntityGraph`
- `NamedNativeQuery`
- `NamedQuery`
- `NamedStoredProcedureQuery`
- `PersistenceContext`
- `PersistenceUnit`
- `PrimaryKeyJoinColumn`
- `SecondaryTable`
- `SqlResultSetMapping`
- `SequenceGenerator`
- `TableGenerator`

Added `SequenceGenerators` and `TableGenerators` annotations.

Added support for CDI injection into `AttributeConverter` classes.

Added support for the mapping of the following java.time types:

- `java.time.LocalDate`
- `java.time.LocalTime`
- `java.time.LocalDateTime`
- `java.time.OffsetTime`
- `java.time.OffsetDateTime`

Added default `Stream getResultStream()` method to Query interface.

Added default `Stream<X> getResultStream()` method to TypedQuery interface.

Replaced reference to JAR file specification in persistence provider bootstrapping section with more general reference to Java SE service provider requirements.

Updated persistence.xml and orm.xml schemas to 2.2 versions.

Updated Related Documents.

Appendix B: Persistence API Interfaces

The following APIs are defined in the package `jakarta.persistence`.

B.1. `EntityHandler`

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.CriteriaSelect;
import jakarta.persistence.criteria.CriteriaStatement;
import jakarta.persistence.metamodel.Metamodel;
import jakarta.persistence.sql.ResultSetMapping;

import java.util.List;
import java.util.Map;

/**
 * Declares operations common to {@link EntityManager} and
 * {@link EntityAgent}. The status of an entity instance
 * returned by an operation declared by this interface
 * depends on the semantics of the subtype.
 * <ul>
 * <li>For an {@link EntityAgent}, there is no associated
 *     persistence context, and so every entity instance
 *     returned by any operation of the agent itself, or of
 *     any query object obtained from the agent, is in the
 *     detached state. Operations on an entity agent never
 *     return the same entity instance twice.
 * <li>For an {@link EntityManager}, every entity instance
 *     returned by any operation of the manager itself, or
 *     of any query object obtained from the manager, is
 *     in the managed state and belongs to the persistence
 *     context associated with the entity manager. The
 *     instance remains in the managed state until that
 *     entity manager is closed and its persistence context
 *     is destroyed unless the instance is explicitly
 *     detached via a call to {@link EntityManager#detach
 *     detach()} or {@link EntityManager#clear clear()}.
 *     An {@code EntityManager} is required to ensure that
 *     if an entity instance representing a given record
 *     already exists in its associated persistence context,
 *     then exactly that instance is returned by every
 *     operation that returns an entity representing the
 *     record.
 * </ul>
 * <p>Note that, in a Jakarta EE container environment, a
 * transaction-scoped entity manager which is accessed
 * outside the scope of a container transaction is destroyed
 * immediately after it is called. And so such an entity
 * manager may appear to return unmanaged instances to its
 * client. But this is merely a consequence of the extremely
 * short lifetime of its persistence context.
 *
 * <p>Operations which return entity instances are, by default,
 * permitted to retrieve state from the second-level cache, if
 * any, or, in the case of an {@code EntityManager}, directly
 * from the persistence context, without directly accessing
 * the database. However, any such operation must respect:
 * <ul>
 * <li>any {@linkplain LockModeType lock mode} requested as an
 *     argument to the operation or by calling
 *     {@link TypedQuery#setLockMode(LockModeType)}, as well as
```

```
 * <li>any {@linkplain CacheRetrieveMode cache mode} given
 *     as an {@linkplain FindOption optional argument} or by
 *     calling {@link #setCacheRetrieveMode} or
 *     {@link TypedQuery#setCacheRetrieveMode}.
 * </ul>
 * <p>Thus, access to the database might be required even when
 * the entity is available in the persistence context or in the
 * second-level cache.
 *
 * <p>Use of the second-level cache, if any, may be controlled
 * by calling {@link #setCacheRetrieveMode(CacheRetrieveMode)}
 * and {@link #setCacheStoreMode(CacheStoreMode)}.
 *
 * <p>Outside the Jakarta EE container environment, the safest
 * and simplest way to obtain an {@code EntityHandler} is by
 * calling {@link EntityManagerFactory#runInTransaction} or
 * {@link EntityManagerFactory#callInTransaction}.
 * {@snippet :
 * Book book =
 *         factory.callInTransaction(EntityAgent.class,
 *                 agent -> agent.get(Book.class, isbn))
 * }
 *
 * @since 4.0
 */
public interface EntityHandler extends AutoCloseable {

    /**
     * Retrieve an entity representing the record with the
     * given identifier.
     *
     * @param entityClass The class of the entity to retrieve
     * @param id The identifier of the entity to retrieve
     * @param <T> The entity type
     *
     * @return an entity instance with the given identifier
     *
     * @throws IllegalArgumentException if the given class is
     *         not an entity class belonging to the persistence
     *         unit, or if the given identifier is not a
     *         non-null instance of the identifier type of the
     *         given entity class
     * @throws EntityNotFoundException if no record with the
     *         given identifier exists in the database
     * @throws PersistenceException if the record could not be
     *         read from the database
     *
     * @apiNote Conceptually similar to {@linkplain #find},
     * except this form throws {@linkplain EntityNotFoundException}
     * instead of returning null.
     *
     * @since 4.0
     */
    @Nonnull
    <T> T get(@Nonnull Class<T> entityClass, @Nonnull Object id);

    /**
     * Retrieve an entity representing the record with the
     * given identifier, using the specified
     * {@linkplain FindOption options}. If the given options
     * include a {@link LockModeType}, obtain the lock level
     * specified by the given lock mode.
     * <p>If an implementation does not recognize a given
     * vendor-specific {@linkplain FindOption option}, it
     * must silently ignore the option.
     * <p>Portable applications should not rely on the
     * standard {@linkplain Timeout timeout option}.
```

```
 * Depending on the database in use and the locking
 * mechanisms used by the provider, this option may or
 * may not be observed.
 *
 * @param entityClass The class of the entity to retrieve
 * @param id The identifier of the entity to retrieve
 * @param options Standard and vendor-specific options
 * @param <T> The entity type
 *
 * @return an entity instance with the given identifier
 *
 * @throws IllegalArgumentException if the given class is
 *         not an entity class belonging to the persistence
 *         unit, or if the given identifier is not a
 *         non-null instance of the identifier type of the
 *         given entity class
 * @throws TransactionRequiredException if any lock mode
 *         other than {@link LockModeType#NONE NONE} is
 *         specified and there is no transaction
 *         associated with this handler
 * @throws OptimisticLockException if an optimistic version
 *         check fails
 * @throws PessimisticLockException if a pessimistic lock
 *         could not be obtained and the transaction is
 *         rolled back
 * @throws LockTimeoutException if a pessimistic lock
 *         could not be obtained and only the statement
 *         is rolled back
 * @throws EntityNotFoundException if no record with the
 *         given identifier exists in the database
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if the
 *         record could not be read from the database
 *
 * @apiNote Conceptually similar to {@linkplain #find},
 * except this form throws {@linkplain EntityNotFoundException}
 * instead of returning null.
 *
 * @since 4.0
 */
@Nonnull
<T> T get(@Nonnull Class<T> entityClass,
          @Nonnull Object id,
          @Nullable FindOption... options);

/**
 * Retrieve an entity representing the record with the
 * given identifier, fetching associations specified by
 * the given {@linkplain EntityGraph load graph}, and
 * using the specified {@linkplain FindOption options}.
 * If the given options include a {@link LockModeType},
 * obtain the lock level specified by the given lock
 * mode.
 * <p>If an implementation does not recognize a given
 * vendor-specific {@linkplain FindOption option}, it
 * must silently ignore the option.
 * <p>Portable applications should not rely on the
 * standard {@linkplain Timeout timeout option}.
 * Depending on the database in use and the locking
 * mechanisms used by the provider, this option may or
 * may not be observed.
 *
 * @param graph The {@linkplain EntityGraph load graph}
 * @param id The identifier of the entity to retrieve
 * @param options Standard and vendor-specific options
 * @param <T> The entity type
```

```
 *
 * @return an entity instance with the given identifier
 *
 * @throws IllegalArgumentException if the given class is
 *         not an entity class belonging to the persistence
 *         unit, or if the given identifier is not a
 *         non-null instance of the identifier type of the
 *         given entity class
 * @throws TransactionRequiredException if any lock mode
 *         other than {@link LockModeType#NONE NONE} is
 *         specified and there is no transaction
 *         associated with this handler
 * @throws OptimisticLockException if an optimistic version
 *         check fails
 * @throws PessimisticLockException if a pessimistic lock
 *         could not be obtained and the transaction is
 *         rolled back
 * @throws LockTimeoutException if a pessimistic lock
 *         could not be obtained and only the statement
 *         is rolled back
 * @throws EntityNotFoundException if no record with the
 *         given identifier exists in the database
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if the
 *         record could not be read from the database
 *
 * @apiNote Conceptually similar to {@linkplain #find},
 * except this form throws {@linkplain EntityNotFoundException}
 * instead of returning null.
 *
 * @since 4.0
 */
@Nonnull
<T> T get(@Nonnull EntityGraph<T> graph,
          @Nonnull Object id,
          @Nullable FindOption... options);

/**
 * Retrieve entity instances representing the records
 * with the given identifiers, using the specified
 * {@linkplain FindOption options}, and returning the
 * instances in a list where the position of an instance
 * in the list matches the position of its identifier in
 * the given array. If the given options include a
 * {@link LockModeType}, obtain the lock level specified
 * by the given lock mode.
 * <p>If an implementation does not recognize a given
 * vendor-specific {@linkplain FindOption option}, it
 * must silently ignore the option.
 * <p>Portable applications should not rely on the
 * standard {@linkplain Timeout timeout option}.
 * Depending on the database in use and the locking
 * mechanisms used by the provider, this option may or
 * may not be observed.
 *
 * @param entityClass The class of the entity to retrieve
 * @param ids The identifiers of the entities to retrieve
 * @param options Standard and vendor-specific options
 * @param <T> The entity type
 * @return an ordered list of entity instances
 * @throws IllegalArgumentException if the given class is
 *         not an entity class belonging to the persistence
 *         unit, or if one of the given identifiers is not
 *         a non-null instance of the identifier type of
 *         the given entity class
 * @throws TransactionRequiredException if any lock mode
```

```
 *         other than {@link LockModeType#NONE NONE} is
 *         specified and there is no transaction
 *         associated with this handler
 * @throws OptimisticLockException if an optimistic version
 *         check fails
 * @throws PessimisticLockException if a pessimistic lock
 *         could not be obtained and the transaction is
 *         rolled back
 * @throws LockTimeoutException if a pessimistic lock
 *         could not be obtained and only the statement
 *         is rolled back
 * @throws EntityNotFoundException if no record exists in
 *         the database for one of the given identifiers
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if a
 *         record could not be read from the database
 * @apiNote Conceptually similar to {@linkplain #findMultiple},
 * except this form throws {@linkplain EntityNotFoundException}
 * instead of returning nulls.
 * @since 4.0
 */
@Nonnull
<T> List<T> getMultiple(@Nonnull Class<T> entityClass,
                        @Nonnull List<?> ids,
                        @Nullable FindOption... options);

/**
 * Retrieve entity instances representing the records with
 * the given identifiers of the root entity of the given
 * {@linkplain EntityGraph load graph}, using the specified
 * {@linkplain FindOption options}, and returning the
 * instances in a list where the position of an instance
 * in the list matches the position of its identifier in
 * the given array, and fetching associations specified by
 * the load graph.
 * <p>If an implementation does not recognize a given
 * vendor-specific {@linkplain FindOption option}, it
 * must silently ignore the option.
 * <p>Portable applications should not rely on the
 * standard {@linkplain Timeout timeout option}.
 * Depending on the database in use and the locking
 * mechanisms used by the provider, this option may or
 * may not be observed.
 *
 * @param graph The {@linkplain EntityGraph load graph}
 * @param ids The identifiers of the entities to retrieve
 * @param options Standard and vendor-specific options
 * @param <T> The entity type
 * @return an ordered list of entity instances
 * @throws IllegalArgumentException if the given class is
 *         not an entity class belonging to the persistence
 *         unit, or if one of the given identifiers is not
 *         a non-null instance of the identifier type of
 *         the given entity class
 * @throws TransactionRequiredException if any lock mode
 *         other than {@link LockModeType#NONE NONE} is
 *         specified and there is no transaction
 *         associated with this handler
 * @throws OptimisticLockException if an optimistic version
 *         check fails
 * @throws PessimisticLockException if a pessimistic lock
 *         could not be obtained and the transaction is
 *         rolled back
 * @throws LockTimeoutException if a pessimistic lock
 *         could not be obtained and only the statement
 *         is rolled back
```

```
 * @throws EntityNotFoundException if no record exists in
 *         the database for one of the given identifiers
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if a
 *         record could not be read from the database
 * @apiNote Conceptually similar to {@linkplain #findMultiple},
 * except this form throws {@linkplain EntityNotFoundException}
 * instead of returning nulls.
 * @since 4.0
 */
@Nonnull
<T> List<T> getMultiple(@Nonnull EntityGraph<T> graph,
                        @Nonnull List<?> ids,
                        @Nullable FindOption... options);

/**
 * Retrieve an entity representing the record with the
 * given identifier, or return {@code null} if there is
 * no such record in the database.
 *
 * @param entityClass The class of the entity to retrieve
 * @param id The identifier of the entity to retrieve
 * @param <T> The entity type
 * @return an entity instance with the given identifier,
 *         or {@code null} if there is no matching record
 *         in the database
 *
 * @throws IllegalArgumentException if the given class is
 *         not an entity class belonging to the persistence
 *         unit, or if the given identifier is not a
 *         non-null instance of the identifier type of the
 *         given entity class
 * @throws PersistenceException if the record could not be
 *         read from the database
 *
 * @since 1.0
 */
@Nullable
<T> T find(@Nonnull Class<T> entityClass, @Nonnull Object id);

/**
 * Retrieve an entity representing the record with the
 * given identifier, or return {@code null} if there is
 * no such record in the database, using the specified
 * {@linkplain FindOption options}. If the given options
 * include a {@link LockModeType}, obtain the lock level
 * specified by the given lock mode.
 * <p>If an implementation does not recognize a given
 * vendor-specific {@linkplain FindOption option}, it
 * must silently ignore the option.
 * <p>Portable applications should not rely on the
 * standard {@linkplain Timeout timeout option}.
 * Depending on the database in use and the locking
 * mechanisms used by the provider, this option may or
 * may not be observed.
 *
 * @param entityClass The class of the entity to retrieve
 * @param id The identifier of the entity to retrieve
 * @param options Standard and vendor-specific options
 * @param <T> The entity type
 * @return an entity instance with the given identifier,
 *         or {@code null} if there is no matching record
 *         in the database
 *
 * @throws IllegalArgumentException if the given class is
 *         not an entity class belonging to the persistence
```

```
 *         unit, if the given identifier is not a non-null
 *         instance of the identifier type of the given
 *         entity class, or if the given options are
 *         contradictory
 * @throws TransactionRequiredException if any lock mode
 *         other than {@link LockModeType#NONE NONE} is
 *         specified and there is no transaction
 *         associated with this handler
 * @throws OptimisticLockException if an optimistic version
 *         check fails
 * @throws PessimisticLockException if a pessimistic lock
 *         could not be obtained and the transaction is
 *         rolled back
 * @throws LockTimeoutException if a pessimistic lock
 *         could not be obtained and only the statement
 *         is rolled back
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if the
 *         record could not be read from the database
 *
 * @since 3.2
 */
@Nullable
<T> T find(@Nonnull Class<T> entityClass,
           @Nonnull Object id,
           @Nullable FindOption... options);

/**
 * Retrieve an instance of the root entity of the given
 * {@link EntityGraph} representing the record with the
 * given identifier, or return {@code null} if there is
 * no such record in the database, using the specified
 * {@linkplain FindOption options} and interpreting the
 * {@code EntityGraph} as a load graph. If the given
 * options include a {@link LockModeType}, obtain the
 * lock level specified by the given lock mode.
 * <p>If an implementation does not recognize a given
 * vendor-specific {@linkplain FindOption option}, it
 * must silently ignore the option.
 * <p>Portable applications should not rely on the
 * standard {@linkplain Timeout timeout option}.
 * Depending on the database in use and the locking
 * mechanisms used by the provider, this option may or
 * may not be observed.
 *
 * @param graph The {@linkplain EntityGraph load graph}
 * @param id The identifier of the entity to retrieve
 * @param options Standard and vendor-specific options
 * @param <T> The entity type
 * @return an entity instance with the given identifier,
 *         or {@code null} if there is no matching record
 *         in the database
 *
 * @throws IllegalArgumentException if the root entity
 *         of the given graph is not an entity class
 *         belonging to the persistence unit, if the
 *         given identifier is not a non-null instance of
 *         the identifier type of the given entity class,
 *         or if the given options are contradictory
 * @throws TransactionRequiredException if any lock mode
 *         other than {@link LockModeType#NONE NONE} is
 *         specified and there is no transaction
 *         associated with this handler
 * @throws OptimisticLockException if an optimistic version
 *         check fails
 * @throws PessimisticLockException if a pessimistic lock
```

```
 *         could not be obtained and the transaction is
 *         rolled back
 * @throws LockTimeoutException if a pessimistic lock
 *         could not be obtained and only the statement
 *         is rolled back
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the root entity of the given graph
 *         or if the record could not be read from the database
 *
 * @since 3.2
 */
@Nullable
<T> T find(@Nonnull EntityGraph<T> graph,
           @Nonnull Object id,
           @Nullable FindOption... options);

/**
 * Retrieve entity instances representing the records
 * with the given identifiers, using the specified
 * {@linkplain FindOption options}, returning the
 * instances in a list where the position of an instance
 * in the list matches the position of its identifier in
 * the given array, and the list contains a null value
 * if there is no record matching a given identifier.
 * If the given options include a {@link LockModeType},
 * obtain the lock level specified by the given lock mode.
 * <p>If an implementation does not recognize a given
 * vendor-specific {@linkplain FindOption option}, it
 * must silently ignore the option.
 * <p>Portable applications should not rely on the
 * standard {@linkplain Timeout timeout option}.
 * Depending on the database in use and the locking
 * mechanisms used by the provider, this option may or
 * may not be observed.
 *
 * @param entityClass The class of the entity to retrieve
 * @param ids The identifiers of the entities to retrieve
 * @param options Standard and vendor-specific options
 * @param <T> The entity type
 * @return an ordered list of entity instances with the
 *         given identifiers, with {@code null} in
 *         positions where there is no matching record
 *         in the database
 * @throws IllegalArgumentException if the given class is
 *         not an entity class belonging to the persistence
 *         unit, if one of the given identifiers is not a
 *         non-null instance of the identifier type of the
 *         given entity class, or if the given options are
 *         contradictory
 * @throws TransactionRequiredException if any lock mode
 *         other than {@link LockModeType#NONE NONE} is
 *         specified and there is no transaction
 *         associated with this handler
 * @throws OptimisticLockException if an optimistic version
 *         check fails
 * @throws PessimisticLockException if a pessimistic lock
 *         could not be obtained and the transaction is
 *         rolled back
 * @throws LockTimeoutException if a pessimistic lock
 *         could not be obtained and only the statement
 *         is rolled back
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if a
 *         record could not be read from the database
 * @since 4.0
```

```
 */
@Nonnull
<T> List<T> findMultiple(@Nonnull Class<T> entityClass,
                         @Nonnull List<?> ids,
                         @Nullable FindOption... options);

/**
 * Retrieve entity instances representing the records with
 * the given identifiers of the root entity of the given
 * {@link EntityGraph}, using the specified
 * {@linkplain FindOption options} and interpreting the
 * {@code EntityGraph} as a load graph, returning entity
 * instances in a list where the position of an instance
 * in the list matches the position of its identifier in
 * the given array, and the list contains a null value
 * if there is no record matching a given identifier.
 * If the given options include a {@link LockModeType},
 * obtain the lock level specified by the given lock mode.
 * <p>If an implementation does not recognize a given
 * vendor-specific {@linkplain FindOption option}, it
 * must silently ignore the option.
 * <p>Portable applications should not rely on the
 * standard {@linkplain Timeout timeout option}.
 * Depending on the database in use and the locking
 * mechanisms used by the provider, this option may or
 * may not be observed.
 *
 * @param graph The {@linkplain EntityGraph load graph}
 * @param ids The identifiers of the entities to retrieve
 * @param options Standard and vendor-specific options
 * @param <T> The entity type
 * @return an ordered list of entity instances with the
 *         given identifiers, with {@code null} in
 *         positions where there is no matching record
 *         in the database
 * @throws IllegalArgumentException if the root entity
 *         of the given graph is not an entity class
 *         belonging to the persistence unit, if one of
 *         the given identifiers is not a non-null
 *         instance of the identifier type of the given
 *         entity class, or if the given options are
 *         contradictory
 * @throws TransactionRequiredException if any lock mode
 *         other than {@link LockModeType#NONE NONE} is
 *         specified and there is no transaction
 *         associated with this handler
 * @throws OptimisticLockException if an optimistic version
 *         check fails
 * @throws PessimisticLockException if a pessimistic lock
 *         could not be obtained and the transaction is
 *         rolled back
 * @throws LockTimeoutException if a pessimistic lock
 *         could not be obtained and only the statement
 *         is rolled back
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the root entity of the given graph
 *         or if a record could not be read from the database
 * @since 4.0
 */
@Nonnull
<T> List<T> findMultiple(@Nonnull EntityGraph<T> graph,
                         @Nonnull List<?> ids,
                         @Nullable FindOption... options);

/**
 * Set the default {@linkplain CacheRetrieveMode cache retrieval
```

```
 * mode} for this {@code EntityHandler}.
 * @param cacheRetrieveMode The new default cache retrieval mode
 * @since 3.2
 */
void setCacheRetrieveMode(@Nonnull CacheRetrieveMode cacheRetrieveMode);

/**
 * Set the default {@linkplain CacheStoreMode cache storage mode}
 * for this {@code EntityHandler}.
 * @param cacheStoreMode The new default cache storage mode
 * @since 3.2
 */
void setCacheStoreMode(@Nonnull CacheStoreMode cacheStoreMode);

/**
 * The cache retrieval mode for this {@code EntityHandler}.
 *
 * @since 3.2
 */
@Nonnull
CacheRetrieveMode getCacheRetrieveMode();

/**
 * The cache storage mode for this {@code EntityHandler}.
 *
 * @since 3.2
 */
@Nonnull
CacheStoreMode getCacheStoreMode();

/**
 * Set an {@code EntityHandler}-scoped property or hint. If a
 * vendor-specific property or hint is not recognized, it is
 * silently ignored.
 *
 * @param propertyName The name of the property or hint
 * @param value The value for the property or hint
 * @throws IllegalArgumentException if the given property or
 *         hint name is recognized by the provider, but the
 *         second argument is not a legal value for the given
 *         property or hint
 * @apiNote The use of named properties and hints lacks type
 *          safety compared to the use of
 *          {@linkplain EntityManager.Option manager options}
 *          or {@linkplain EntityAgent.Option agent options}.
 * @since 2.0
 */
void setProperty(@Nonnull String propertyName, @Nullable Object value);

/**
 * The properties and hints and their associated values which
 * are in effect for this {@code EntityHandler}. Modifying the
 * contents of the returned map does not change the configuration
 * in effect.
 *
 * @return a copy of the properties and hints currently in effect
 * @since 2.0
 */
@Nonnull
Map<String, Object> getProperties();

/**
 * Create an instance of {@link Statement} for executing a
 * Jakarta Persistence {@code UPDATE} or {@code DELETE}
 * statement.
 *
 * @param qlString A Jakarta Persistence statement string
```

```
 * @return An instance of {@link Statement} which may be
 *         used to execute the given statement
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid
 * @since 4.0
 */
@Nonnull
Statement createStatement(@Nonnull String qlString);

/**
 * Create an instance of {@link StatementOrTypedQuery}
 * for executing a Jakarta Persistence query language
 * statement.
 * <p>If the given query is a {@code SELECT} statement,
 * the query results might be packaged as arrays:
 * <ul>
 * <li>if the query contains a single item in its select
 *     list, each query result is the value of that item,
 *     or
 * <li>otherwise, if the query contains multiple items in
 *     its select list, each query result is packaged in
 *     an array of type {@code Object[]}, with the array
 *     elements corresponding by position with the items
 *     of the select list.
 * </ul>
 *
 * @param qlString A Jakarta Persistence query string
 * @return An instance of {@link StatementOrTypedQuery}
 *         which may be used to obtain a {@link TypedQuery}
 *         or {@link Statement} or to directly execute the
 *         given query or statement
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid
 * @apiNote For backward compatibility, the returned object
 * may be used to directly execute any kind of statement or
 * query via its deprecated methods. Newly written code
 * should call:
 * <ul>
 * <li>{@link #createStatement(String)} to obtain a
 * {@link Statement},
 * <li>an overload which specifies a
 *     {@linkplain #createQuery(String, Class) result class}
 *     or {@linkplain #createQuery(String, EntityGraph)
 *     entity graph} obtain a {@link TypedQuery}, or
 * <li>{@link StatementOrTypedQuery#asStatement asStatement()}
 *     or {@link StatementOrTypedQuery#ofType(Class) ofType()}
 *     to obtain a {@link Statement} or {@link TypedQuery},
 *     respectively, from the object returned by this method.
 * </ul>
 */
@Nonnull
StatementOrTypedQuery createQuery(@Nonnull String qlString);

/**
 * Create an instance of {@link TypedQuery} for executing a
 * {@linkplain CriteriaSelect criteria select}, which may be
 * top-level query or a union or intersection of top-level
 * queries.
 *
 * @param selectQuery A criteria select query object
 * @param <T> The query result type
 * @return An instance of {@link TypedQuery} which may be used
 *         to execute the given query
 * @throws IllegalArgumentException if the criteria query is
 *         found to be invalid
 * @since 3.2
 */
```

```
@Nonnull
<T> TypedQuery<T> createQuery(@Nonnull CriteriaSelect<T> selectQuery);

/**
 * Create an instance of {@link Statement} for executing a
 * {@linkplain CriteriaStatement criteria statement}.
 *
 * @param statement A criteria statement object
 * @return An instance of {@link Statement} which may be
 *         used to execute the given statement
 * @throws IllegalArgumentException if the statement is
 *         found to be invalid
 * @since 4.0
 */
@Nonnull
Statement createStatement(@Nonnull CriteriaStatement<?> statement);

/**
 * Create an instance of {@link TypedQuery} for executing
 * a Jakarta Persistence query language {@code SELECT}
 * statement and returning instances of the given result
 * class. Either:
 * <ol>
 * <li>the select list of the query contains only a single
 *     item, which must be assignable to the result class,
 * <li>the result class is {@code Object[].class} or
 *     {@link Tuple Tuple.class}, or
 * <li>the result class is a non-abstract class or record
 *     type with a constructor with the same number of
 *     parameters as the query has items in its select list,
 *     and the constructor parameter types exactly match the
 *     types of the corresponding items in the select list.
 * </ol>
 * <p>In the first case, each query result is returned
 * directly to the caller. In the second case, each query
 * result is packaged in an array with the array elements
 * corresponding by position with the items of the query
 * select list or in an instance of {@link Tuple} with the
 * indices and aliases of the tuple elements determined by
 * the positions and aliases of the corresponding items in
 * the select list. In the third case, each query result is
 * automatically packaged in a new instance of the result
 * class by calling the matching constructor.
 *
 * @param qlString A Jakarta Persistence query string
 * @param resultClass The result class
 * @param <T> The query result type
 * @return An instance of {@link TypedQuery} which may be used
 *         to execute the given query
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid, or if the query result is
 *         found to not be assignable to the specified type
 *         and the specified type does not have a suitable
 *         constructor
 * @since 2.0
 */
@Nonnull
<T> TypedQuery<T> createQuery(@Nonnull String qlString,
                              @Nonnull Class<T> resultClass);

/**
 * Create an instance of {@link TypedQuery} for executing
 * a Jakarta Persistence query language {@code SELECT}
 * statement, specifying an {@link EntityGraph} which is
 * interpreted as a load graph. The select list of the
 * query must contain only a single item, which must be
 * assignable to the root type of the given entity graph.
```

```
 *
 * @param qlString A Jakarta Persistence query string
 * @param resultGraph The {@linkplain EntityGraph load graph}
 * @param <T> The query result type
 * @return An instance of {@link TypedQuery} which may be used
 *         to execute the given query
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid, or if the query result is
 *         found to not be assignable to the root type of
 *         the entity graph
 * @since 4.0
 */
@Nonnull
<T> TypedQuery<T> createQuery(@Nonnull String qlString,
                              @Nonnull EntityGraph<T> resultGraph);

/**
 * Create an instance of {@link Statement} for executing a
 * named {@code UPDATE} or {@code DELETE} statement written
 * in the Jakarta Persistence query language, or a named native
 * SQL statement which returns a row count.
 * @param name The name of a statement defined in metadata
 * @return An instance of {@link Statement} which may be used
 *         to execute the named statement
 * @throws IllegalArgumentException if a query has not been
 *         defined with the given name, or if the query string
 *         is found to be invalid
 * @see NamedStatement
 * @see NamedNativeStatement
 * @see EntityManagerFactory#addNamedStatement(String, Statement)
 *
 * @since 4.0
 */
Statement createNamedStatement(String name);

/**
 * Create an instance of {@link StatementOrTypedQuery} for
 * executing a named query written in the Jakarta Persistence
 * query language or in native SQL.
 * <ul>
 * <li>If the named query is a {@code SELECT} statement
 *     written in the Jakarta Persistence query language,
 *     the query results are packaged according to the
 *     {@linkplain NamedQuery#resultClass result class}
 *     specified by the {@link NamedQuery} annotation, or
 * <li>If the named query is written in native SQL and
 *     returns a result set, the query results are
 *     interpreted and packaged according to the
 *     {@linkplain NamedNativeQuery#resultClass result class}
 *     and {@linkplain NamedNativeQuery#resultSetMapping
 *     result set mapping} specified by the
 *     {@link NamedNativeQuery} annotation.
 * </ul>
 *
 * @param name The name of a query defined in metadata
 * @return An instance of {@link StatementOrTypedQuery} which
 * may be used to obtain a {@link TypedQuery} or
 * {@link Statement} or to directly execute the
 * given query or statement
 * @throws IllegalArgumentException if a query has not been
 *                                  defined with the given name, or if the query string
 *                                  is found to be invalid
 * @apiNote For backward compatibility, the returned object
 * may be used to directly execute any kind of statement or
 * query via its deprecated methods. Newly written code
 * should call:
 * <ul>
```

```
 * <li>{@link #createNamedStatement(String)} to obtain a
 * {@link Statement},
 * <li>the overload which specifies a
 *     {@linkplain #createNamedQuery(String, Class) result
 *     class} to obtain a {@link TypedQuery}, or
 * <li>{@link StatementOrTypedQuery#asStatement asStatement()}
 *     or {@link StatementOrTypedQuery#ofType(Class) ofType()}
 *     to obtain a {@link Statement} or {@link TypedQuery},
 *     respectively, from the object returned by this method.
 * </ul>
 * @see NamedQuery
 * @see NamedNativeQuery
 * @since 1.0
 */
@Nonnull
StatementOrTypedQuery createNamedQuery(@Nonnull String name);

/**
 * Create an instance of {@link TypedQuery} for executing a
 * named query written in the Jakarta Persistence query language
 * or in native SQL, returning instances of the given result class.
 * <ul>
 * <li>If the named query is written in the Jakarta Persistence
 *     query language, the result class is interpreted as if it
 *     were an argument of {@link #createQuery(String, Class)}.
 * <li>If the named query is written in native SQL, the result
 *     class is interpreted as if it were an argument of
 *     {@link #createNativeQuery(String, Class)}.
 * </ul>
 * <p>The given result class overrides any result class specified
 * by the {@link NamedQuery} annotation or {@link NamedNativeQuery}
 * annotation which declares the named query.
 *
 * @param name The name of a query defined in metadata
 * @param resultClass The type of the query result
 * @param <T> The query result type
 * @return An instance of {@link TypedQuery} which may be used
 *         to execute the given query
 * @throws IllegalArgumentException if a query has not been
 *         defined with the given name, if the query string is
 *         found to be invalid, or if the query result is found to
 *         not be assignable to the specified type
 * @since 2.0
 */
@Nonnull
<T> TypedQuery<T> createNamedQuery(@Nonnull String name,
                                   @Nonnull Class<T> resultClass);

/**
 * Create an instance of {@link Statement} for executing a named
 * statement written in the Jakarta Persistence query language or
 * in native SQL.
 *
 * @param reference a reference to the query defined in metadata
 * @return the new query instance
 * @throws IllegalArgumentException if a named query has not been
 *         defined, or if the query string is found to be invalid
 * @see EntityManagerFactory#getNamedStatements()
 * @see NamedQuery
 * @see NamedNativeQuery
 * @since 4.0
 */
@Nonnull
Statement createStatement(@Nonnull StatementReference reference);

/**
 * Create an instance of {@link TypedQuery} for executing a
```

```
 * typed named query written in the Jakarta Persistence query
 * language or in native SQL.
 * <ul>
 * <li>If the named query is a {@code SELECT} statement
 *     written in the Jakarta Persistence query language,
 *     the query results are packaged according to the
 *     {@linkplain NamedQuery#resultClass result class}
 *     specified by the {@link NamedQuery} annotation, or
 * <li>If the named query is written in native SQL and
 *     returns a result set, the query results are
 *     interpreted and packaged according to the
 *     {@linkplain NamedNativeQuery#resultClass result class}
 *     and {@linkplain NamedNativeQuery#resultSetMapping
 *     result set mapping} specified by the
 *     {@link NamedNativeQuery} annotation.
 * </ul>
 *
 * @param reference A reference to the query defined in metadata
 * @param <T> The query result type
 * @return An instance of {@link TypedQuery} which may be used
 *         to execute the given query
 * @throws IllegalArgumentException if a query has not been
 *         defined, if the query string is found to be
 *         invalid, or if the query result is found to not be
 *         assignable to the specified type
 * @see EntityManagerFactory#getNamedQueries(Class)
 * @see NamedQuery
 * @see NamedNativeQuery
 * @since 3.2
 */
@Nonnull
<T> TypedQuery<T> createQuery(@Nonnull TypedQueryReference<T> reference);

/**
 * Create an instance of {@link Statement} for executing a
 * native SQL statement which returns a row count, usually
 * an {@code INSERT}, {@code UPDATE}, {@code MERGE}, or
 * {@code DELETE} statement.
 *
 * @param sqlString A native SQL statement string
 * @return An instance of {@link Statement} which may be
 *         used to execute the given statement
 * @since 4.0
 */
@Nonnull
Statement createNativeStatement(@Nonnull String sqlString);

/**
 * Create an instance of {@link StatementOrTypedQuery} for
 * executing a native SQL query or statement.
 * <p>If the given query produces a result set, the query
 * results might be packaged as arrays:
 * <ul>
 * <li>if the query contains a single column in its result
 *     set, each query result is the value of that column,
 *     or
 * <li>otherwise, if the query contains multiple columns
 *     in its result set, each query result is packaged in
 *     an array of type {@code Object[]}, with the array
 *     elements corresponding by position with the columns
 *     of the select list.
 * </ul>
 * <p>Column values are obtained according to the default
 * type mappings defined by the JDBC specification.
 *
 * @param sqlString A native SQL query string
 * @return An instance of {@link StatementOrTypedQuery} which
```

```
 * may be used to obtain a {@link TypedQuery} or
 * {@link Statement} or to directly execute the
 * given query or statement
 * @apiNote For backward compatibility, the returned object
 * may be used to directly execute any kind of statement or
 * query via its deprecated methods. Newly written code
 * should call:
 * <ul>
 * <li>{@link #createNativeStatement(String)} to obtain a
 * {@link Statement},
 * <li>the overload which specifies a
 *     {@linkplain #createNativeQuery(String, Class) result
 *     class} to obtain a {@link TypedQuery}, or
 * <li>{@link StatementOrTypedQuery#asStatement asStatement()}
 *     or {@link StatementOrTypedQuery#ofType(Class) ofType()}
 *     to obtain a {@link Statement} or {@link TypedQuery},
 *     respectively, from the object returned by this method.
 * </ul>
 * @since 1.0
 */
@Nonnull
StatementOrTypedQuery createNativeQuery(@Nonnull String sqlString);

/**
 * Create an instance of {@link TypedQuery} for executing a native
 * SQL query which produces a result set, returning instances of the
 * given result class. Either:
 * <ul>
 * <li>the result class is an entity class and is interpreted as a
 *     managed {@linkplain EntityResult entity result} with implicit
 *     field mappings determined by the names of the columns in the
 *     result set and the object/relational mapping of the entity,
 * <li>the result class is the class of a {@linkplain Basic basic}
 *     type and the result set must have a single column which is
 *     interpreted as a {@linkplain ColumnResult scalar result},
 * <li>the result class is a non-{@code abstract} class or record
 *     type with a constructor with the same number of parameters
 *     as the result set has columns, and is interpreted as a
 *     {@linkplain ConstructorResult constructor result} including
 *     all the columns of the result set, or
 * <li>the result class is {@code Object[].class} and each query
 *     result is packaged in an array of type {@code Object[]},
 *     with the array elements corresponding by position with the
 *     columns of the select list and column values obtained
 *     according to the default type mappings defined by the JDBC
 *     specification.
 * </ul>
 *
 * @param sqlString A native SQL query string
 * @param resultClass The type of the query result
 * @param <T> The query result type
 * @return An instance of {@link TypedQuery} which may be used
 *         to execute the given query
 * @since 1.0
 */
@Nonnull
<T> TypedQuery<T> createNativeQuery(@Nonnull String sqlString,
                                    @Nonnull Class<T> resultClass);

/**
 * Create an instance of {@link StatementOrTypedQuery} for
 * executing a native SQL query, using the named
 * {@linkplain SqlResultSetMapping result set mapping} with the
 * given {@linkplain SqlResultSetMapping#name name} to interpret
 * the JDBC result set.
 *
 * @param sqlString A native SQL query string
```

```
 * @param resultSetMapping The name of the result set mapping
 * @return An instance of {@link StatementOrTypedQuery} which
 *         may be used to obtain a {@link TypedQuery} or to
 *         directly execute the given query
 * @apiNote For backward compatibility, the returned object
 * may be used to directly execute a query via its deprecated
 * methods. Newly written code should call:
 * <ul>
 * <li>the overload which specifies a typed
 *     {@linkplain #createNativeQuery(String, ResultSetMapping)
 *     result set mapping} to obtain a {@link TypedQuery}, or
 * <li>{@link StatementOrTypedQuery#ofType(Class) ofType()} to
 *     obtain a {@link TypedQuery} from the object returned
 *     by this method.
 * </ul>
 * <p>A typed reference to the result set mapping may be
 * obtained from the static metamodel.
 * @see SqlResultSetMapping
 * @since 1.0
 */
@Nonnull
StatementOrTypedQuery createNativeQuery(@Nonnull String sqlString,
                                        @Nonnull String resultSetMapping);

/**
 * Create an instance of {@link TypedQuery} for executing a native
 * SQL query, using the given {@link ResultSetMapping} to interpret
 * the JDBC result set.
 *
 * @param sqlString A native SQL query string
 * @param resultSetMapping The result set mapping
 * @param <T> The query result type
 * @return An instance of {@link TypedQuery} which may be used
 *         to execute the given query
 * @since 4.0
 */
@Nonnull
<T> TypedQuery<T> createNativeQuery(@Nonnull String sqlString,
                                    @Nonnull ResultSetMapping<T> resultSetMapping);

/**
 * Create an instance of {@link StoredProcedureQuery} for executing
 * a stored procedure in the database.
 * <p>Parameters must be registered before the stored procedure can
 * be executed.
 * <p>If the stored procedure returns one or more result sets, any
 * result set is returned as a list of type {@code Object[]}.
 *
 * @param name The name assigned to the stored procedure query in
 *             metadata
 * @return An instance of {@link StoredProcedureQuery} which may be
 *         used to execute the stored procedure
 * @throws IllegalArgumentException if no query has been defined
 *         with the given name
 * @since 2.1
 */
@Nonnull
StoredProcedureQuery createNamedStoredProcedureQuery(@Nonnull String name);

/**
 * Create an instance of {@link StoredProcedureQuery} for executing a
 * stored procedure in the database.
 * <p>Parameters must be registered before the stored procedure can
 * be executed.
 * <p>If the stored procedure returns one or more result sets, any
 * result set is returned as a list of type {@code Object[]}.
 *
```

```
 * @param procedureName The name of the stored procedure in the
 *                      database
 * @return An instance of {@link StoredProcedureQuery} which may be
 *         used to execute the stored procedure
 * @throws IllegalArgumentException if a stored procedure of the
 *         given name does not exist (or if query execution will
 *         fail)
 * @since 2.1
 */
@Nonnull
StoredProcedureQuery createStoredProcedureQuery(@Nonnull String procedureName);

/**
 * Create an instance of {@link StoredProcedureQuery} for executing
 * a stored procedure in the database, explicitly specifying the
 * result class for every result set returned by the stored procedure.
 * <p>Parameters must be registered before the stored procedure can
 * be executed.
 * <p>The given result classes must be specified in the order in which
 * the corresponding result sets are returned by the stored procedure
 * invocation. For each given result class, either:
 * <ul>
 * <li>the result class is an entity class and is interpreted as a
 *     managed {@linkplain EntityResult entity result} with implicit
 *     field mappings determined by the names of the columns in the
 *     result set and the object/relational mapping of the entity,
 * <li>the result class is the class of a {@linkplain Basic basic}
 *     type and the result set must have a single column which is
 *     interpreted as a {@linkplain ColumnResult scalar result}, or
 * <li>the result class must be a non-abstract class or record type
 *     with a constructor with the same number of parameters as the
 *     result set has columns, and is interpreted as a
 *     {@linkplain ConstructorResult constructor result} including
 *     all the columns of the result set.
 * </ul>
 *
 * @param procedureName The name of the stored procedure in the
 *                      database
 * @param resultClasses The classes to which the result sets
 *                      produced by the stored procedure are mapped
 * @return An instance of {@link StoredProcedureQuery} which may be
 *         used to execute the stored procedure
 * @throws IllegalArgumentException if a stored procedure of the
 *         given name does not exist (or if query execution will
 *         fail)
 * @apiNote Use of this overloaded form of the method results
 * in a redundant type cast of each query result in client code.
 * Instead, use {@link #createStoredProcedureQuery(String)} and
 * pass each result class individually as an argument to
 * {@link StoredProcedureQuery#getResultList(Class)},
 * {@link StoredProcedureQuery#getSingleResult(Class)}, or
 * {@link StoredProcedureQuery#getSingleResultOrNull(Class)}.
 * @since 2.1
 */
@Nonnull
StoredProcedureQuery createStoredProcedureQuery(
        @Nonnull String procedureName,
        @Nonnull Class<?>... resultClasses);

/**
 * Create an instance of {@link StoredProcedureQuery} for executing
 * a stored procedure in the database, explicitly specifying the
 * {@linkplain SqlResultSetMapping#name name} of a result set
 * {@linkplain SqlResultSetMapping mapping} for every result set
 * returned by the stored procedure.
 * <p>Parameters must be registered before the stored procedure can
 * be executed.
```

```
 * <p>The given result set mappings must be specified in the order
 * in which the corresponding result sets are returned by the stored
 * procedure invocation.
 *
 * @param procedureName The name of the stored procedure in the
 *                      database
 * @param resultSetMappings The names of the result set mappings
 *                          to be used to map result sets returned
 *                          by the stored procedure
 * @return An instance of {@link StoredProcedureQuery} which may be
 *         used to execute the stored procedure
 * @throws IllegalArgumentException if a stored procedure or
 *         result set mapping of the given name does not exist
 *         (or the query execution will fail)
 * @apiNote Use of this overloaded form of the method results
 * in a type cast of each query result in client code. As an
 * alternative, use {@link #createStoredProcedureQuery(String)}
 * and pass each result mapping individually as an argument to
 * {@link StoredProcedureQuery#getResultList(ResultSetMapping)},
 * {@link StoredProcedureQuery#getSingleResult(ResultSetMapping)}, or
 * {@link StoredProcedureQuery#getSingleResultOrNull(ResultSetMapping)}.
 * @see SqlResultSetMapping
 * @since 2.1
 */
@Nonnull
StoredProcedureQuery createStoredProcedureQuery(
        @Nonnull String procedureName,
        @Nonnull String... resultSetMappings);

/**
 * Return an object of the specified type to allow access to
 * a provider-specific API. If the provider implementation
 * of {@code EntityHandler} does not support the given type,
 * the {@link PersistenceException} is thrown.
 * @param type The class of the object to be returned.
 *             This is usually either the underlying class
 *             implementing {@code EntityHandler} or an
 *             interface it implements
 * @param <T> The type of the object to be returned
 * @return An instance of the specified class
 * @throws PersistenceException if the provider does not
 *         support the given type
 * @since 2.0
 */
@Nonnull
<T> T unwrap(@Nonnull Class<T> type);

/**
 * Close an application-managed {@code EntityHandler}.
 * <p>After invocation of {@code close()}, every method of
 * the {@code EntityHandler} instance and of any instance
 * of {@link Query} obtained from it throws the
 * {@link IllegalStateException}, except for
 * {@link #getProperties()}, {@link #getTransaction()},
 * and {@link #isOpen()} (which returns false).
 * <p>If this method is called when the {@code EntityHandler}
 * is joined to an active transaction, any managed entities
 * remain managed until the transaction completes.
 * @throws IllegalStateException if the {@code EntityHandler}
 *         is container-managed
 * @since 1.0
 */
void close();

/**
 * Determine whether the {@code EntityHandler} is open.
 * @return true until the {@code EntityHandler} has been closed
```

```
 * @since 1.0
 */
boolean isOpen();

/**
 * Return the resource-level {@link EntityTransaction} object.
 * The {@code EntityTransaction} instance may be used serially
 * to begin and commit multiple transactions.
 *
 * @return An instance of {@link EntityTransaction}
 * @throws IllegalStateException if invoked on a
 *         {@linkplain PersistenceUnitTransactionType#JTA JTA}
 *         entity manager or entity agent
 * @since 1.0
 */
@Nonnull
EntityTransaction getTransaction();

/**
 * The {@linkplain EntityManagerFactory entity manager factory}
 * which created this {@code EntityHandler}.
 *
 * @return The {@link EntityManagerFactory}
 * @throws IllegalStateException if the {@code EntityHandler}
 *         has been closed
 * @since 2.0
 */
@Nonnull
EntityManagerFactory getEntityManagerFactory();

/**
 * Obtain an instance of {@link CriteriaBuilder} which may be
 * used to construct {@link CriteriaQuery} objects.
 *
 * @return an instance of {@link CriteriaBuilder}
 * @throws IllegalStateException if the {@code EntityHandler}
 *         has been closed
 * @see EntityManagerFactory#getCriteriaBuilder()
 * @since 2.0
 */
@Nonnull
CriteriaBuilder getCriteriaBuilder();

/**
 * Obtain an instance of the {@link Metamodel} interface which
 * provides access to metamodel objects describing the managed
 * types belonging to the persistence unit.
 *
 * @return An instance of {@link Metamodel}
 * @throws IllegalStateException if the {@code EntityHandler}
 *         has been closed
 * @since 2.0
 */
@Nonnull
Metamodel getMetamodel();

/**
 * Create a new mutable {@link EntityGraph}, allowing programmatic
 * definition of the graph.
 *
 * @param rootType the root entity type of the new graph
 * @param <T> The root entity type
 * @return a trivial entity graph with only a root node
 * @see jakarta.persistence.metamodel.EntityType#createEntityGraph()
 * @since 2.1
 */
@Nonnull
```

```
<T> EntityGraph<T> createEntityGraph(@Nonnull Class<T> rootType);

/**
 * Obtain a mutable copy of the named {@link EntityGraph}.
 *
 * @param graphName the name of an existing entity graph
 * @return the entity graph with the given name
 * @throws IllegalArgumentException if there is no entity
 *         graph with the given name
 * @since 2.1
 */
@Nonnull
EntityGraph<?> getEntityGraph(@Nonnull String graphName);

/**
 * Obtain a mutable copy of the named {@link EntityGraph}
 * whose root type is exactly the given entity type.
 *
 * @param rootType the root entity type of the graph
 * @param graphName the name of an existing entity graph
 * @param <T> The root entity type
 * @return the entity graph with the given name
 * @throws IllegalArgumentException if there is no entity
 *         graph with the given name, or if the entity
 *         graph with the given name does not have exactly
 *         the given root entity type
 * @since 4.0
 */
@Nonnull
<T> EntityGraph<T> getEntityGraph(@Nonnull Class<T> rootType,
                                  @Nonnull String graphName);

/**
 * Return all named {@link EntityGraph}s that are defined for
 * the given entity class type.
 *
 * @param entityClass An entity class
 * @param <T> The entity type
 * @return A list of all entity graphs whose root entity type
 *         is a supertype of the given entity class
 * @throws IllegalArgumentException if the class is not an entity
 * @since 2.1
 */
@Nonnull
<T> List<EntityGraph<? super T>> getEntityGraphs(@Nonnull Class<T> entityClass);

/**
 * Execute the given action using the database connection underlying this
 * {@code EntityHandler}. Usually, the connection is a JDBC connection, but a
 * provider might support some other native connection type, and is not required
 * to support {@code java.sql.Connection}. If this {@code EntityHandler} is
 * associated with a transaction, the action is executed in the context of the
 * transaction. The given action should close any resources it creates but should
 * not close the connection itself, nor commit or roll back the transaction. If
 * the given action throws an exception, the persistence provider must mark the
 * transaction for rollback.
 * {@snippet :
 * entityManager.runWithConnection((Connection connection) -> {
 *     try (var statement = connection.createStatement()) {
 *         statement.execute("set constraints all deferred");
 *     }
 * });
 * }
 * @param action The action to execute
 * @param <C> The connection type, usually {@code java.sql.Connection}
 * @throws PersistenceException wrapping the checked {@link Exception} thrown by
 *         {@link ConnectionConsumer#accept}, if any
```

```
     * @since 3.2
     */
    <C> void runWithConnection(@Nonnull ConnectionConsumer<C> action);

    /**
     * Call the given function and return its result using the database connection
     * underlying this {@code EntityHandler}. Usually, the connection is a JDBC
     * connection, but a provider might support some other native connection type,
     * and is not required to support {@code java.sql.Connection}. If this
     * {@code EntityHandler} is associated with a transaction, the function is
     * executed in the context of the transaction. The given function should close
     * any resources it creates but should not close the connection itself, nor
     * commit or roll back the transaction. If the given action throws an exception,
     * the persistence provider must mark the transaction for rollback.
     * {@snippet :
     * LocalTime currentTimeOnServer =
     *         entityManager.callWithConnection((Connection connection) -> {
     *             try (var statement = connection.createStatement()) {
     *                 try (var resultSet = statement.executeQuery("select current_time")) {
     *                     resultSet.next();
     *                     return resultSet.getObject(1, LocalTime.class);
     *                 }
     *             }
     *         });
     * }
     * @param function The function to call
     * @param <C> The connection type, usually {@code java.sql.Connection}
     * @param <T> The type of result returned by the function
     * @return The value returned by {@link ConnectionFunction#apply}.
     * @throws PersistenceException wrapping the checked {@link Exception} thrown by
     *         {@link ConnectionFunction#apply}, if any
     * @since 3.2
     */
    <C,T> T callWithConnection(@Nonnull ConnectionFunction<C, T> function);

}
```

B.2. EntityManager

```
package jakarta.persistence;

import java.util.Map;
import java.util.Set;
import java.util.function.Consumer;
import java.util.function.Function;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.CriteriaStatement;

/**
 * Interface used to interact with the persistence context.
 *
 * <p>An instance of {@code EntityManager} must be obtained from
 * an {@link EntityManagerFactory}, and is only able to manage
 * persistence of entities belonging to the associated persistence
 * unit. In the Jakarta EE environment, an entity manager with a
 * lifecycle managed by the container may be obtained by dependency
 * injection.
 *
 * <ul>
 * <li>An {@code EntityManager} obtained directly from an
 *     {@code EntityManagerFactory} is never thread safe, and it is
 *     always wrong to share a reference to such an entity manager
```

```
*     between multiple concurrently executing threads.
* <li>On the other hand, in the Jakarta EE container environment,
*     a reference to a container-managed, transaction-scoped
*     entity manager obtained by injection is safe to invoke
*     concurrently from distinct threads. The container redirects
*     invocations to distinct instances of {@code EntityManager}
*     based on transaction affinity. Even in this scenario, the
*     underlying persistence context must never be shared between
*     threads.
* </ul>
*
* <p>An application-managed {@code EntityManager} may be created
* via a call to {@link EntityManagerFactory#createEntityManager}.
* The {@code EntityManager} must be explicitly closed via a call
* to {@link #close()}, to allow resources to be cleaned up by the
* persistence provider. This approach places almost complete
* responsibility for cleanup and exception management on the client,
* and is thus considered quite error-prone. It is much safer to use
* the methods {@link EntityManagerFactory#runInTransaction(Consumer)}
* and {@link EntityManagerFactory#callInTransaction(Function)}.
* {@snippet :
* entityManagerFactory.runInTransaction(entityManager -> {
*     // do work in a persistence context
*     ...
* });
* }
*
* <p>In the Jakarta EE environment, a container-managed
* {@link EntityManager} may be obtained by dependency injection,
* using {@link PersistenceContext}.
* {@snippet :
* // inject the container-managed entity manager
* @PersistenceContext(unitName="orderMgt")
* EntityManager entityManager;
* }
*
* <p>If the persistence unit has
* {@linkplain PersistenceUnitTransactionType#RESOURCE_LOCAL
* resource local} transaction management, transactions must
* be managed using the {@link EntityTransaction} obtained by
* calling {@link #getTransaction()}.
*
* <p>A complete idiom for custom application management of
* the {@link EntityManager} and its associated resource-local
* {@link EntityTransaction} is as follows:
* {@snippet :
* EntityManager entityManager = entityManagerFactory.createEntityManager();
* EntityTransaction transaction = entityManager.getTransaction();
* try {
*     transaction.begin();
*     // do work
*     ...
*     transaction.commit();
* }
* catch (Exception e) {
*     if (transaction.isActive()) transaction.rollback();
*     throw e;
* }
* finally {
*     entityManager.close();
* }
* }
*
* <p>Each {@code EntityManager} instance is associated with a
* distinct <em>persistence context</em>. A persistence context
* is a set of entity instances in which for any given persistent
* entity identity (defined by an entity type and primary key)
```

```
* there is at most one entity instance. The entity instances
* associated with a persistence context are considered managed
* objects, with a well-defined lifecycle under the control of
* the persistence provider.
*
* <p>Any entity instance can be characterized as being in one of
* the following lifecycle states:
* <ul>
* <li>A <em>new</em> entity has no persistent identity, and is
*     not yet associated with any persistence context.
* <li>A <em>managed</em> entity is an instance with a persistent
*     identity that is currently associated with a persistence
*     context.
* <li>A <em>detached</em> entity is an instance with a persistent
*     identity that is not (or no longer) associated with any
*     active persistence context.
* <li>A <em>removed</em> entity is an instance with a persistent
*     identity, and associated with a persistence context, that
*     is scheduled for removal from the database upon transaction
*     commit.
* </ul>
*
* <p>The {@code EntityManager} API is used to perform operations
* that affect the state of the persistence context, or that modify
* the lifecycle state of individual entity instances. The client
* may {@linkplain #persist} and {@linkplain #remove} instances,
* {@linkplain #find(Class, Object) find} entities by their primary
* key, and execute {@linkplain #createQuery(String) queries} which
* range over entity types. A given entity may be disassociated
* from the persistence context by calling {@link #detach}, and a
* persistence context may be completely cleared, detaching all
* its entities, by calling {@link #clear()}.
*
* <p>The client may also make changes to the state of an entity
* instance by mutating the entity directly, or it may request
* that the state of a detached instance be {@linkplain #merge
* merged}, replacing the state of a managed instance with the
* same persistent identity. Note that there is no explicit
* "update" operation; since an entity is a managed object,
* modifications to its persistent fields and properties are
* automatically detected, as long as it is associated with an
* active persistence context.
*
* <p>Modifications to the state of entities associated with a
* persistence context are not immediately synchronized with the
* database. Synchronization happens during a process called
* <em>flush</em>. The timing of the flush process depends on the
* {@linkplain FlushModeType flush mode}, which may be set
* explicitly by calling {@link #setFlushMode(FlushModeType)}.
* <ul>
* <li>For {@link FlushModeType#COMMIT}, the persistence context
*     is flushed before the transaction commits.
* <li>For {@link FlushModeType#AUTO}, which is the default, the
*     persistence context must also be flushed before execution
*     of any query whose result set would be affected by
*     unflushed modifications to entities associated with the
*     persistence context.
* </ul>
* The client may force an immediate flush to occur by calling
* {@link #flush()}.
*
* <p>At any given moment, a persistence context might hold an
* optimistic or pessimistic <em>lock</em> on an entity instance.
* The full range of possible lock types is enumerated by
* {@link LockModeType}. Some operations of this interface,
* including the methods {@link #lock(Object, LockModeType)},
* {@link #refresh(Object, RefreshOption...)}, and
```

```
 * {@link #find(Class, Object, FindOption...)}, accept an explicit
 * {@link LockModeType}, allowing the client to request a specific
 * type of lock.
 *
 * <p>Interaction of the persistence context (or first-level cache)
 * with the second-level cache, if any, may be controlled by
 * calling {@link #setCacheRetrieveMode(CacheRetrieveMode)} and
 * {@link #setCacheStoreMode(CacheStoreMode)}.
 *
 * <p>An exception thrown by a method of an {@code EntityManager},
 * or by a method of an instance of {@link Query} obtained from an
 * {@code EntityManager}, causes the active transaction to be marked
 * for rollback unless the exception type is explicitly designated
 * as recoverable.
 *
 * <p>Some operations accept one or more built-in and vendor-specific
 * options:
 * <ul>
 * <li>{@link #find(Class, Object, FindOption...)} and
 *     {@link #find(EntityGraph, Object, FindOption...)} accept
 *     {@link FindOption}s,
 * <li>{@link #refresh(Object, RefreshOption...)} accepts
 *     {@link RefreshOption}s, and
 * <li>{@link #lock(Object, LockModeType, LockOption...)} accepts
 *     {@link LockOption}s.
 * </ul>
 *
 * @see FlushModeType
 * @see SynchronizationType
 * @see EntityManagerFactory
 * @see EntityTransaction
 * @see Statement
 * @see TypedQuery
 * @see StoredProcedureQuery
 * @see CriteriaQuery
 * @see PersistenceContext
 *
 * @since 1.0
 */
public interface EntityManager extends EntityHandler {

    /**
     * Make a new entity instance managed and persistent, resulting in
     * its insertion in the database when the persistence context is
     * synchronized with the database, or make a removed entity managed,
     * undoing the effect of a previous call to {@link #remove(Object)}.
     * This operation cascades to every entity related by an association
     * marked {@link CascadeType#PERSIST cascade=PERSIST}. If the given
     * entity instance is already managed, that is, if it already belongs
     * to this persistence context and has not been marked for removal,
     * it is itself ignored, but the operation still cascades.
     * @param entity  a new, managed, or removed entity instance
     * @throws EntityExistsException if the given entity is detached; if
     *         the entity is detached, either an {@code EntityExistsException}
     *         is thrown immediately, or an {@code EntityExistsException}
     *         or some other subtype of {@code PersistenceException} is
     *         thrown when the persistence context is flushed or when the
     *         transaction commits
     * @throws IllegalArgumentException if the given instance is not an
     *         entity
     * @throws TransactionRequiredException if there is no transaction
     *         when invoked on a container-managed entity manager that
     *         is of type {@link PersistenceContextType#TRANSACTION}
     * @throws PersistenceException if the entity class has a generated
     *         identifier and an identifier could not be generated
     */
    void persist(@Nonnull Object entity);
```

```
/**
 * Merge the state of the given new or detached entity instance
 * into the current persistence context, resulting in, respectively,
 * an insert or possible update when the persistence context is
 * synchronized with the database. Return a managed instance with
 * the same persistent state as the given entity instance, but a
 * distinct Java object identity. If the given entity is detached,
 * the returned entity has the same persistent identity. This
 * operation cascades to every entity related by an association
 * marked {@link CascadeType#MERGE cascade=MERGE}. If the given
 * entity instance is managed, that is, if it belongs to this
 * persistence context and has not been marked for removal, it is
 * itself ignored, but the operation still cascades, and it is
 * returned directly.
 * @param entity  a new, managed, or detached entity instance
 * @param <T> the entity type
 * @return the managed instance that the state was merged to
 * @throws IllegalArgumentException if the instance is not an entity
 *         or is a removed entity
 * @throws TransactionRequiredException if there is no transaction
 *         when invoked on a container-managed entity manager of
 *         type {@link PersistenceContextType#TRANSACTION}
 * @throws OptimisticLockException if an optimistic locking conflict
 *         is detected (note that optimistic version checking might be
 *         deferred until changes are flushed to the database)
 * @throws PersistenceException if a record could not be read from
 *         the database
 *
 */
@Nonnull
<T> T merge(@Nonnull T entity);

/**
 * Mark a managed entity instance as removed, resulting in its deletion
 * from the database when the persistence context is synchronized with
 * the database. This operation cascades to every entity related by an
 * association marked {@link CascadeType#REMOVE cascade=REMOVE}. If the
 * given entity instance is already removed, it is ignored. If the
 * given entity is new, it is itself ignored, but the operation still
 * cascades.
 * @param entity  a managed, new, or removed entity instance
 * @throws IllegalArgumentException if the instance is not an entity
 *         or is a detached entity
 * @throws TransactionRequiredException if invoked on a
 *         container-managed entity manager of type
 *         {@link PersistenceContextType#TRANSACTION} and there is
 *         no transaction
 * @throws OptimisticLockException if an optimistic locking conflict
 *         is detected (note that optimistic version checking might be
 *         deferred until changes are flushed to the database)
 */
void remove(@Nonnull Object entity);

/**
 * Find by primary key, using the specified properties.
 * Search for an entity of the specified class and primary key.
 * If the entity instance is contained in the persistence
 * context, it is returned from there.
 * If a vendor-specific property or hint is not recognized,
 * it is silently ignored.
 * @param entityClass  entity class
 * @param primaryKey   primary key
 * @param properties  standard and vendor-specific properties
 *        and hints
 * @param <T> the entity type
 * @return the found entity instance or null if the entity does
```

```
 *         not exist
 * @throws IllegalArgumentException if the first argument does
 *         not denote an entity type, or if the second argument
 *         is not a valid type for that entity's primary key or
 *         is null
 * @throws PersistenceException if the record could not be
 *         read from the database
 * @since 2.0
 */
@Nullable
<T> T find(@Nonnull Class<T> entityClass,
           @Nonnull Object primaryKey,
           @Nullable Map<String, Object> properties);

/**
 * Find by primary key and lock the entity, using the specified
 * properties. Search for an entity of the specified class and
 * primary key and lock it with respect to the specified lock type.
 * If the entity instance is contained in the persistence context,
 * it is returned from there.
 * <p> If the entity is found within the persistence context and
 * the lock mode type is pessimistic and the entity has a version
 * attribute, the persistence provider must perform optimistic
 * version checks when obtaining the database lock. If these checks
 * fail, the {@link OptimisticLockException} is thrown.
 * <p>If the lock mode type is pessimistic, and the entity instance
 * is found but cannot be locked:
 * <ul>
 * <li>the {@link PessimisticLockException} is thrown if the
 *     database locking failure causes transaction-level rollback
 * <li>the {@link LockTimeoutException} is thrown if the database
 *     locking failure causes only statement-level rollback
 * </ul>
 * <p>If a vendor-specific property or hint is not recognized,
 * it is silently ignored.
 * <p>Portable applications should not rely on the standard
 * timeout hint. Depending on the database in use and the locking
 * mechanisms used by the provider, the hint may or may not be
 * observed.
 * @param entityClass  entity class
 * @param primaryKey  primary key
 * @param lockMode  lock mode
 * @param properties  standard and vendor-specific properties
 *        and hints
 * @param <T> the entity type
 * @return the found entity instance or null if the entity does
 *         not exist
 * @throws IllegalArgumentException if the first argument does
 *         not denote an entity type, or the second argument is
 *         not a valid type for that entity's primary key or
 *         is null
 * @throws TransactionRequiredException if there is no
 *         transaction and a lock mode other than {@code NONE} is
 *         specified or if invoked on an entity manager which has
 *         not been joined to the current transaction and a lock
 *         mode other than {@code NONE} is specified
 * @throws OptimisticLockException if the optimistic version check
 *         fails
 * @throws PessimisticLockException if pessimistic locking fails
 *         and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking fails and
 *        only the statement is rolled back
 * @throws PersistenceException if the given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if the
 *         record could not be read from the database
 * @since 2.0
```

```
 */
@Nullable
<T> T find(@Nonnull Class<T> entityClass,
           @Nonnull Object primaryKey,
           @Nonnull LockModeType lockMode,
           @Nullable Map<String, Object> properties);

/**
 * Obtain a reference to an instance of the given entity class
 * with the given primary key, whose state may be lazily fetched.
 * <p>If the requested instance does not exist in the database,
 * the {@link EntityNotFoundException} is thrown when the
 * instance state is first accessed.
 * (The persistence provider runtime is permitted but not
 * required to throw the {@code EntityNotFoundException} when
 * {@code getReference()} is called.)
 * <p>This operation allows the application to create an
 * association to an entity without loading its state from the
 * database.
 * <p>The application should not expect the instance state to
 * be available upon detachment unless it was accessed by the
 * application while the entity manager was open.
 * @param entityClass  entity class
 * @param primaryKey  primary key
 * @param <T> the entity type
 * @return a reference to the entity instance
 * @throws IllegalArgumentException if the first argument does
 *         not denote an entity type, or the second argument is
 *         not a valid type for that entity's primary key or
 *         is null
 * @throws EntityNotFoundException if the entity state cannot
 *         be accessed
 */
@Nonnull
<T> T getReference(@Nonnull Class<T> entityClass,
                   @Nonnull Object primaryKey);

/**
 * Obtain a reference to an instance of the entity class of the
 * given object, with the same primary key as the given object,
 * whose state may be lazily fetched. The given object may be
 * persistent or detached, but may be neither new nor removed.
 * <p>If the requested instance does not exist in the database,
 * the {@link EntityNotFoundException} is thrown when the
 * instance state is first accessed.
 * (The persistence provider runtime is permitted but not
 * required to throw the {@code EntityNotFoundException} when
 * {@code getReference()} is called.)
 * <p>This operation allows the application to create an
 * association to an entity without loading its state from the
 * database.
 * <p>The application should not expect the instance state to
 * be available upon detachment unless it was accessed by the
 * application while the entity manager was open.
 * @param entity  a persistent or detached entity instance
 * @param <T> the entity type
 * @return a reference to the entity instance
 * @throws IllegalArgumentException if the given object is not
 *         an entity, or if it is neither persistent nor detached
 * @throws EntityNotFoundException if the entity state cannot be
 *         accessed
 * @since 3.2
 */
@Nonnull
<T> T getReference(@Nonnull T entity);

/**
```

```
 * Synchronize changes held in the persistence context to the
 * underlying database.
 * @throws TransactionRequiredException if there is
 *        no transaction or if the entity manager has not been
 *        joined to the current transaction
 * @throws EntityExistsException if any kind of uniqueness
 *         constraint is violated by an operation which inserts
 *         an entity in the database
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected
 */
void flush();

/**
 * Set the {@linkplain FlushModeType flush mode} that applies to
 * all objects contained in the persistence context.
 * @param flushMode  flush mode
 */
void setFlushMode(@Nonnull FlushModeType flushMode);

/**
 * Get the {@linkplain FlushModeType flush mode} that applies to
 * all objects contained in the persistence context.
 * @return the current {@link FlushModeType}
 */
@Nonnull
FlushModeType getFlushMode();

/**
 * Lock an entity instance belonging to the persistence context,
 * obtaining the specified {@linkplain LockModeType lock mode}.
 * <p>If a pessimistic lock mode type is specified and the entity
 * contains a version attribute, the persistence provider must
 * also perform optimistic version checks when obtaining the
 * database lock. If these checks fail, the
 * {@link OptimisticLockException} is thrown.
 * <p>If the lock mode type is pessimistic, and the entity instance
 * is found but cannot be locked:
 * <ul>
 * <li>the {@link PessimisticLockException} is thrown if the
 *     database locking failure causes transaction-level rollback
 * <li>the {@link LockTimeoutException} is thrown if the database
 *     locking failure causes only statement-level rollback
 * </ul>
 * @param entity  a managed entity instance
 * @param lockMode  lock mode
 * @throws IllegalArgumentException if the instance is not an
 *         entity or is a detached entity
 * @throws TransactionRequiredException if there is no
 *         transaction or if invoked on an entity manager which
 *         has not been joined to the current transaction
 * @throws EntityNotFoundException if the entity does not exist
 *         in the database when pessimistic locking is
 *         performed
 * @throws OptimisticLockException if the optimistic version check
 *         fails
 * @throws PessimisticLockException if pessimistic locking fails
 *         and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking fails and
 *         only the statement is rolled back
 * @throws PersistenceException if the given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class
 */
void lock(@Nonnull Object entity, @Nonnull LockModeType lockMode);
```

```
/**
 * Lock an entity instance belonging to the persistence context,
 * obtaining the specified {@linkplain LockModeType lock mode},
 * using the specified properties.
 * <p>If a pessimistic lock mode type is specified and the entity
 * contains a version attribute, the persistence provider must
 * also perform optimistic version checks when obtaining the
 * database lock. If these checks fail, the
 * {@link OptimisticLockException} is thrown.
 * <p>If the lock mode type is pessimistic, and the entity instance
 * is found but cannot be locked:
 * <ul>
 * <li>the {@link PessimisticLockException} is thrown if the
 *     database locking failure causes transaction-level rollback
 * <li>the {@link LockTimeoutException} ia thrown if the database
 *     locking failure causes only statement-level rollback
 * </ul>
 * <p>If a vendor-specific property or hint is not recognized,
 * it is silently ignored.
 * <p>Portable applications should not rely on the standard
 * timeout hint. Depending on the database in use and the locking
 * mechanisms used by the provider, the hint may or may not be
 * observed.
 * @param entity  a managed entity instance
 * @param lockMode  lock mode
 * @param properties  standard and vendor-specific properties
 *        and hints
 * @throws IllegalArgumentException if the instance is not an
 *         entity or is a detached entity
 * @throws TransactionRequiredException if there is no
 *         transaction or if invoked on an entity manager which
 *         has not been joined to the current transaction
 * @throws EntityNotFoundException if the entity does not exist
 *         in the database when pessimistic locking is performed
 * @throws OptimisticLockException if the optimistic version check
 *         fails
 * @throws PessimisticLockException if pessimistic locking fails
 *         and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking fails and
 *         only the statement is rolled back
 * @throws PersistenceException if the given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class
 * @since 2.0
 */
void lock(@Nonnull Object entity, @Nonnull LockModeType lockMode,
          @Nullable Map<String, Object> properties);

/**
 * Lock an entity instance belonging to the persistence context,
 * obtaining the specified {@linkplain LockModeType lock mode},
 * using the specified {@linkplain LockOption options}. If the
 * given options include a {@link PessimisticLockScope}, the
 * specified scope determines whether a pessimistic lock is
 * applied to associated collections and relationships.
 * <p>If a pessimistic lock mode type is specified and the entity
 * contains a version attribute, the persistence provider must
 * also perform optimistic version checks when obtaining the
 * database lock. If these checks fail, the
 * {@link OptimisticLockException} is thrown.
 * <p>If the lock mode type is pessimistic, and the entity instance
 * is found but cannot be locked:
 * <ul>
 * <li>the {@link PessimisticLockException} is thrown if the
 *     database locking failure causes transaction-level rollback
 * <li>the {@link LockTimeoutException} is thrown if the database
 *     locking failure causes only statement-level rollback
```

```
 * </ul>
 * <p>If a vendor-specific {@link LockOption} is not recognized,
 * it is silently ignored.
 * <p>Portable applications should not rely on the standard
 * {@linkplain Timeout timeout option}. Depending on the database
 * in use and the locking mechanisms used by the provider, the
 * option may or may not be observed.
 * @param entity  a managed entity instance
 * @param lockMode  lock mode
 * @param options  standard and vendor-specific options
 * @throws IllegalArgumentException if the instance is not an
 *         entity or is a detached entity
 * @throws TransactionRequiredException if there is no
 *         transaction or if invoked on an entity manager which
 *         has not been joined to the current transaction
 * @throws EntityNotFoundException if the entity does not exist
 *         in the database when pessimistic locking is
 *         performed
 * @throws OptimisticLockException if the optimistic version
 *         check fails
 * @throws PessimisticLockException if pessimistic locking fails
 *         and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking fails and
 *         only the statement is rolled back
 * @throws PersistenceException if the given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class
 * @since 3.2
 */
void lock(@Nonnull Object entity, @Nonnull LockModeType lockMode,
          @Nullable LockOption... options);

/**
 * Refresh the state of the given managed entity instance from
 * the database, overwriting unflushed changes made to the entity,
 * if any. This operation cascades to every entity related by an
 * association marked {@link CascadeType#REFRESH cascade=REFRESH}.
 * @param entity  a managed entity instance
 * @throws IllegalArgumentException if the instance is not
 *         an entity or the entity is not managed
 * @throws TransactionRequiredException if there is no
 *         transaction when invoked on a container-managed
 *         entity manager of type
 *         {@link PersistenceContextType#TRANSACTION}
 * @throws EntityNotFoundException if the entity no longer
 *         exists in the database
 * @throws PersistenceException if the record could not be read
 *         from the database
 */
void refresh(@Nonnull Object entity);

/**
 * Refresh the state of the given managed entity instance from
 * the database, using the specified properties, and overwriting
 * unflushed changes made to the entity, if any. This operation
 * cascades to every entity related by an association marked
 * {@link CascadeType#REFRESH cascade=REFRESH}.
 * <p>If a vendor-specific property or hint is not recognized,
 * it is silently ignored.
 * @param entity  a managed entity instance
 * @param properties  standard and vendor-specific properties
 *        and hints
 * @throws IllegalArgumentException if the instance is not
 *         an entity or the entity is not managed
 * @throws TransactionRequiredException if there is no
 *         transaction when invoked on a container-managed
 *         entity manager of type
```

```
 *         {@link PersistenceContextType#TRANSACTION}
 * @throws EntityNotFoundException if the entity no longer
 *         exists in the database
 * @throws PersistenceException if the record could not be read
 *         from the database
 * @since 2.0
 */
void refresh(@Nonnull Object entity,
             @Nullable Map<String, Object> properties);

/**
 * Refresh the state of the given managed entity instance from
 * the database, overwriting unflushed changes made to the entity,
 * if any, and obtain the given {@linkplain LockModeType lock mode},
 * using the specified properties. This operation cascades to every
 * entity related by an association marked {@link CascadeType#REFRESH
 * cascade=REFRESH}.
 * <p>If the lock mode type is pessimistic, and the entity instance
 * is found but cannot be locked:
 * <ul>
 * <li>the {@link PessimisticLockException} is thrown if the
 *     database locking failure causes transaction-level rollback
 * <li>the {@link LockTimeoutException} is thrown if the database
 *     locking failure causes only statement-level rollback.
 * </ul>
 * <p>If a vendor-specific property or hint is not recognized,
 * it is silently ignored.
 * <p>Portable applications should not rely on the standard
 * timeout hint. Depending on the database in use and the locking
 * mechanisms used by the provider, the hint may or may not be
 * observed.
 * @param entity  a managed entity instance
 * @param lockMode  lock mode
 * @param properties  standard and vendor-specific properties
 *        and hints
 * @throws IllegalArgumentException if the instance is not an
 *         entity or if the entity is not managed
 * @throws TransactionRequiredException if invoked on a
 *         container-managed entity manager of type
 *         {@link PersistenceContextType#TRANSACTION} when there is
 *         no transaction; if invoked on an extended entity manager
 *         when there is no transaction and a lock mode other than
 *         {@link LockModeType#NONE} was specified; or if invoked
 *         on an extended entity manager that has not been joined
 *         to the current transaction and any lock mode other than
 *         {@code NONE} was specified
 * @throws EntityNotFoundException if the entity no longer exists
 *        in the database
 * @throws PessimisticLockException if pessimistic locking fails
 *        and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking fails and
 *        only the statement is rolled back
 * @throws PersistenceException if the given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if the record
 *         could not be read from the database
 * @since 2.0
 */
void refresh(@Nonnull Object entity, @Nonnull LockModeType lockMode,
             @Nullable Map<String, Object> properties);

/**
 * Refresh the state of the given managed entity instance from the
 * database, using the specified {@linkplain RefreshOption options},
 * overwriting changes made to the entity, if any. If the supplied
 * options include a {@link LockModeType}, lock the given entity,
 * obtaining the given lock mode. This operation cascades to every
```

```
 * entity related by an association marked {@link CascadeType#REFRESH
 * cascade=REFRESH}.
 * <p>If the lock mode type is pessimistic, and the entity instance is
 * found but cannot be locked:
 * <ul>
 * <li>the {@link PessimisticLockException} is thrown if the
 *     database locking failure causes transaction-level rollback
 * <li>the {@link LockTimeoutException} is thrown if the database
 *     locking failure causes only statement-level rollback.
 * </ul>
 * <p>If a vendor-specific {@link RefreshOption} is not recognized,
 * it is silently ignored.
 * <p>Portable applications should not rely on the standard
 * {@linkplain Timeout timeout option}. Depending on the database in
 * use and the locking mechanisms used by the provider, the hint may
 * or may not be observed.
 * @param entity  a managed entity instance
 * @param options  standard and vendor-specific options
 * @throws IllegalArgumentException if the instance is not an entity
 *         or if the entity is not managed
 * @throws TransactionRequiredException if invoked on a
 *         container-managed entity manager of type
 *         {@link PersistenceContextType#TRANSACTION} when there is
 *         no transaction; if invoked on an extended entity manager
 *         when there is no transaction and a lock mode other than
 *         {@link LockModeType#NONE} was specified; or if invoked
 *         on an extended entity manager that has not been joined
 *         to the current transaction and any lock mode other than
 *         {@code NONE} was specified
 * @throws EntityNotFoundException if the entity no longer exists in
 *         the database
 * @throws PessimisticLockException if pessimistic locking fails and
 *         the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking fails and only
 *         the statement is rolled back
 * @throws PersistenceException if a given
 *         {@linkplain LockModeType lock mode type} is not
 *         supported for the given entity class or if the record
 *         could not be read from the database
 * @since 3.2
 */
void refresh(@Nonnull Object entity,
             @Nullable RefreshOption... options);

/**
 * Clear the persistence context, causing all managed entities to
 * become detached. Changes made to entities that have not already
 * been flushed to the database will never be made persistent.
 */
void clear();

/**
 * Evict the given managed or removed entity from the persistence
 * context, causing the entity to become immediately detached.
 * Unflushed changes made to the entity, if any, including deletion
 * of the entity, will never be synchronized to the database.
 * Managed entities which reference the given entity continue to
 * reference it. This operation cascades to every entity related by
 * an association marked {@link CascadeType#DETACH cascade=DETACH}.
 * If the given entity instance is new or detached, that is, if it
 * is not associated with this persistence context, it is ignored.
 * @param entity  a managed or removed entity instance
 * @throws IllegalArgumentException if the instance is not an
 *         entity
 * @since 2.0
 */
void detach(@Nonnull Object entity);
```

```
/**
 * Determine if the given object is a managed entity instance
 * belonging to the current persistence context.
 * @param entity  entity instance
 * @return boolean value indicating if entity belongs to the
 *         persistence context
 * @throws IllegalArgumentException if not an entity
 */
boolean contains(@Nonnull Object entity);

/**
 * Get the current {@linkplain LockModeType lock mode} held by
 * this persistence context on the given managed entity instance.
 * @param entity  a managed entity instance
 * @return the lock mode currently held
 * @throws TransactionRequiredException if there is no active
 *         transaction, or if the entity manager has not been
 *         joined to the current transaction
 * @throws IllegalArgumentException if a transaction is active
 *         but the given instance is not a managed entity
 * @since 2.0
 */
@Nonnull
LockModeType getLockMode(@Nonnull Object entity);

/**
 * Join the current active JTA transaction.
 * <p>This method should be called on a JTA application-managed
 * {@code EntityManager} that was created outside the scope of
 * the active transaction or on an {@code EntityManager} of
 * type {@link SynchronizationType#UNSYNCHRONIZED} to associate
 * it with the current JTA transaction.
 * @throws TransactionRequiredException if there is no active
 *         transaction
 * @since 1.0
 */
void joinTransaction();

/**
 * Determine whether the {@code EntityManager} is joined to the
 * current transaction. Returns false if the {@code EntityManager}
 * is not joined to the current transaction or if no
 * transaction is active.
 * @return True if the {@code EntityManager} is joined to the
 *         current transaction, or false otherwise
 * @since 2.1
 */
boolean isJoinedToTransaction();

/**
 * Create an instance of {@link Statement} for executing a
 * {@linkplain CriteriaStatement criteria statement}.
 * @param statement A criteria statement object
 * @return An instance of {@link Statement} which may be
 *         used to execute the given statement
 * @throws IllegalArgumentException if the Statement is
 *         found to be invalid
 * @since 2.1
 * @deprecated Use {@link #createStatement(CriteriaStatement)}.
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
Statement createQuery(@Nonnull CriteriaStatement<?> statement);

/**
 * Obtain a mutable copy of a named {@link EntityGraph} or
```

```
 * return null if there is no entity graph with the given
 * name.
 * @param graphName The name of an entity graph
 * @return A mutable copy of the entity graph, or null
 * @since 2.1
 * @deprecated Use {@link #getEntityGraph(String)} instead.
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nullable
EntityGraph<?> createEntityGraph(@Nonnull String graphName);

/**
 * Return the underlying provider object for the
 * {@link EntityManager}, if available. The result of this
 * method is implementation-specific.
 * <p>The {@code unwrap} method is to be preferred for new
 * applications.
 * @return the underlying provider object
 *
 * @deprecated Use {@link #unwrap(Class)} instead.
 */
@Deprecated(since = "4.0")
@Nonnull
Object getDelegate();

/**
 * Specify an {@linkplain Option option} controlling the
 * behavior of this entity manager, overwriting any existing
 * option of the same type.
 *
 * @param option the option
 * @since 4.0
 */
void addOption(@Nonnull Option option);

/**
 * Get the {@linkplain Option options} controlling the behavior
 * of this entity manager. The returned set includes options set
 * via {@link #addOption}, along with options specified via
 * {@link #setCacheRetrieveMode}, {@link #setCacheStoreMode}, or
 * {@link #setFlushMode}. Mutation of the returned set does not
 * affect the options of the entity manager.
 *
 * @return the options for this entity manager
 * @since 4.0
 */
@Nonnull
Set<Option> getOptions();

/**
 * An option controlling the creation of an entity manager.
 *
 * <p>This interface may be implemented by custom provider-specific
 * options which extend the options defined by the specification.
 *
 * @see SynchronizationType
 *
 * @since 4.0
 */
interface CreationOption {
}

/**
 * An option controlling the behavior of an entity manager.
 * This provides a more type safe alternative to the use of
 * {@linkplain #setProperty properties and hints}.
 *
```

```
     * <p>This interface may be implemented by custom provider-specific
     * options which extend the options defined by the specification.
     *
     * @see FlushModeType
     * @see CacheRetrieveMode
     * @see CacheStoreMode
     *
     * @since 4.0
     */
    interface Option extends CreationOption {
    }
}
```

B.3. EntityAgent

```
import jakarta.annotation.Nonnull;

import java.util.List;
import java.util.Set;
import java.util.function.Consumer;
import java.util.function.Function;

/**
 * Provides entity operations that are performed independently
 * of a persistence context. This interface enables more direct
 * control over interaction with the database than what is
 * possible with {@link EntityManager}.
 *
 * <p>An instance of {@code EntityAgent} must be obtained from
 * an {@link EntityManagerFactory} and is only able to manage
 * persistence of entities belonging to the associated persistence
 * unit. In the Jakarta EE environment, an entity agent with a
 * lifecycle managed by the container may be obtained by
 * dependency injection.
 *
 * <ul>
 * <li>An {@code EntityAgent} obtained directly from an
 *     {@code EntityManagerFactory} is never thread safe, and
 *     it is always wrong to share a reference to such an entity
 *     agent between multiple concurrently executing threads.
 * <li>On the other hand, in the Jakarta EE container environment,
 *     a reference to a container-managed, transaction-scoped
 *     entity agent obtained by injection is safe to invoke
 *     concurrently from distinct threads. The container
 *     redirects invocations to distinct instances of
 *     {@code EntityAgent} based on transaction affinity.
 * </ul>
 *
 * <p>An application-managed {@code EntityAgent} may be created
 * via a call to {@link EntityManagerFactory#createEntityAgent}.
 * The {@code EntityAgent} must be explicitly closed via a call
 * to {@link #close()} to allow resources to be cleaned up by the
 * persistence provider. This approach places almost complete
 * responsibility for cleanup and exception management on the client,
 * and is thus considered quite error-prone. It is much safer to use
 * the methods {@link EntityManagerFactory#runInTransaction(Class, Consumer)}
 * and {@link EntityManagerFactory#callInTransaction(Class, Function)}.
 * {@snippet :
 * entityManagerFactory.runInTransaction(EntityAgent.class, entityAgent -> {
 *     // do work
 *     ...
 * });
 * }
 *
 * <p>In the Jakarta EE environment, a container-managed
```

```
 * {@link EntityAgent} may be obtained by dependency injection,
 * using {@link PersistenceAgent}.
 * {@snippet :
 * // inject the container-managed entity agent
 * @PersistenceAgent(unitName="orderMgt")
 * EntityAgent entityAgent;
 * }
 *
 * <p>If the persistence unit has
 * {@linkplain PersistenceUnitTransactionType#RESOURCE_LOCAL
 * resource local} transaction management, transactions must
 * be managed using the {@link EntityTransaction} obtained by
 * calling {@link #getTransaction()}.
 *
 * <p>A complete idiom for custom application management of
 * the {@link EntityAgent} and its associated resource-local
 * {@link EntityTransaction} is as follows:
 * {@snippet :
 * EntityAgent entityAgent = entityManagerFactory.createEntityAgent();
 * EntityTransaction transaction = entityAgent.getTransaction();
 * try {
 *     transaction.begin();
 *     // do work
 *     ...
 *     transaction.commit();
 * }
 * catch (Exception e) {
 *     if (transaction.isActive()) transaction.rollback();
 *     throw e;
 * }
 * finally {
 *     entityAgent.close();
 * }
 * }
 *
 * <p>An {@code EntityAgent} has no associated persistence context,
 * and works only with detached entity instances. When a method of
 * this interface is called, any necessary interaction with the
 * database happens immediately and synchronously. In particular,
 * {@linkplain #update update} is an explicit operation. Since
 * there is no {@linkplain EntityManager#flush flush operation},
 * and since the entities themselves are detached, modifications to
 * the entities are never automatically detected and made persistent.
 *
 * <p>An exception thrown by a method of an {@code EntityAgent},
 * or by a method of an instance of {@link Query} obtained from
 * an {@code EntityAgent}, is considered recoverable and does not
 * usually cause the active transaction to be marked for rollback.
 *
 * @since 4.0
 *
 * @see EntityManager
 */
public interface EntityAgent extends EntityHandler {

    /**
     * Insert a record.
     * <p>
     * If the entity {@link Id} field is declared to be generated,
     * for example, if it is annotated {@link GeneratedValue}, the
     * id is generated and assigned to the given instance.
     * <p>
     * <ul>
     * <li>The {@link jakarta.persistence.PreInsert} callback is
     *     triggered.
     * <li>The {@link jakarta.persistence.PostInsert} callback is
     *     triggered if the operation is successful.
```

```
 * </ul>
 *
 * @param entity a new or removed entity instance
 *
 * @throws IllegalArgumentException if the given instance is
 *         determined to not be new or removed
 * @throws EntityExistsException if any kind of uniqueness
 *         constraint is violated
 * @throws PersistenceException if a record could not be
 *         inserted in the database
 */
void insert(@Nonnull Object entity);

/**
 * Insert every record in the given list. The records are
 * inserted in the order in which they occur in the given list
 * and according to the requirements of the {@link #insert(Object)}
 * method.
 *
 * @param entities The entities to be inserted.
 *
 * @throws IllegalArgumentException if one of the given
 *         instances is determined to not be new or removed
 * @throws EntityExistsException if any kind of uniqueness
 *         constraint is violated
 * @throws PersistenceException if a record could not be
 *         inserted in the database
 *
 * @see #insert(Object)
 */
void insertMultiple(@Nonnull List<?> entities);

/**
 * Update a record.
 * <p>
 * <ul>
 * <li>The {@link jakarta.persistence.PreUpdate} callback is
 *     triggered.
 * <li>The {@link jakarta.persistence.PostUpdate} callback is
 *     triggered if the operation is successful.
 * </ul>
 *
 * @param entity a detached entity instance
 *
 * @throws IllegalArgumentException if the given instance is
 *         determined to not be detached
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected, that is, if no row matching
 *         the identifier of the given entity exists in the
 *         database or if an optimistic version check fails
 * @throws PersistenceException if a record could not be
 *         updated in the database
 */
void update(@Nonnull Object entity);

/**
 * Update every record in the given list. The records are
 * updated in the order in which they occur in the given list
 * and according to the requirements of the {@link #update(Object)}
 * method.
 *
 * @param entities The entities to be updated.
 *
 * @throws IllegalArgumentException if the one of the given
 *         instances is determined to not be detached
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected, that is, if no row matching
```

```
 *         the identifier of one of the given entities exists
 *         in the database or if an optimistic version check
 *         fails
 * @throws PersistenceException if a record could not be
 *         updated in the database
 *
 * @see #update(Object)
 */
void updateMultiple(@Nonnull List<?> entities);

/**
 * Delete a record.
 * <p>
 * <ul>
 * <li>The {@link jakarta.persistence.PreDelete} callback is
 *     triggered.
 * <li>The {@link jakarta.persistence.PostDelete} callback is
 *     triggered if the operation is successful.
 * </ul>
 *
 * @param entity a detached entity instance
 *
 * @throws IllegalArgumentException if the given instance is
 *         determined to not be detached
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected, that is, if no row matching
 *         the identifier of the given entity exists in the
 *         database or if an optimistic version check fails
 * @throws PersistenceException if a record could not be
 *         deleted from the database
 */
void delete(@Nonnull Object entity);

/**
 * Delete every record in the given list. The records are
 * deleted in the order in which they occur in the given list
 * and according to the requirements of the {@link #delete(Object)}
 * method.
 *
 * @param entities The entities to be deleted.
 *
 * @throws IllegalArgumentException if the one of the given
 *         instances is determined to not be detached
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected, that is, if no row matching
 *         the identifier of one of the given entities exists
 *         in the database or if an optimistic version check
 *         fails
 * @throws PersistenceException if a record could not be
 *         deleted from the database
 *
 * @see #delete(Object)
 */
void deleteMultiple(@Nonnull List<?> entities);

/**
 * Perform an upsert, that is, insert the record if it does not
 * exist, or update it if it already exists.
 * <p>
 * This method never performs id generation and does not accept
 * an entity instance with a null identifier. When id generation
 * is required, use {@link #insert(Object)}.
 * <p>
 * On the other hand, {@code upsert()} does accept an entity
 * instance with an assigned identifier value, even if the entity
 * {@link Id} field is declared to be generated, for example, if
 * it is annotated {@link GeneratedValue}. Thus, this method may
```

```
 * be used to import data from an external source.
 * <p>
 * <ul>
 * <li>The {@link jakarta.persistence.PreUpsert} callback is
 *     triggered.
 * <li>The {@link jakarta.persistence.PostUpsert} callback is
 *     triggered if the operation is successful.
 * </ul>
 *
 * @param entity a detached entity instance, or a new instance
 *               with an assigned identifier
 *
 * @throws IllegalArgumentException if the given entity has a
 *         null identifier value
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected, that is, if an optimistic
 *         version check fails
 * @throws PersistenceException if a record could not be
 *         upserted in the database
 */
void upsert(@Nonnull Object entity);

/**
 * Upsert every record in the given list. The records are
 * upserted in the order in which they occur in the given list
 * and according to the requirements of the {@link #upsert(Object)}
 * method.
 *
 * @param entities The entities to be inserted or updated.
 *
 * @throws IllegalArgumentException if one of the given
 *         entities has a null identifier value
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected, that is, if an optimistic
 *         version check fails
 * @throws PersistenceException if a record could not be
 *         upserted in the database
 *
 * @see #upsert(Object)
 */
void upsertMultiple(@Nonnull List<?> entities);

/**
 * Refresh the entity instance state from the database.
 *
 * @param entity The entity to be refreshed.
 *
 * @throws EntityNotFoundException if the given entity no
 *         longer exists in the database
 * @throws PersistenceException if a record could not be
 *         read from the database
 */
void refresh(@Nonnull Object entity);

/**
 * Refresh multiple entities.
 *
 * @param entities The entities to be refreshed.
 *
 * @see #refresh(Object)
 *
 * @throws EntityNotFoundException if one of the given
 *         entities no longer exists in the database
 * @throws PersistenceException if a record could not be
 *         read from the database
 */
void refreshMultiple(@Nonnull List<?> entities);
```

```
/**
 * Refresh the entity instance state from the database.
 *
 * @param entity The entity to be refreshed.
 * @param lockMode The LockMode to be applied.
 *
 * @throws EntityNotFoundException if the given entity no
 *         longer exists in the database
 * @throws PersistenceException if a record could not be
 *         read from the database
 */
void refresh(@Nonnull Object entity, @Nonnull LockModeType lockMode);

/**
 * Fetch an association or collection that was configured
 * for lazy loading if it has not yet been fetched from
 * the database. If the association or collection is
 * already fetched, this operation has no effect. For
 * convenience, this method always returns its argument.
 * {@snippet :
 * Book book = agent.get(Book.class, isbn);  // book is immediately detached
 * agent.fetch(book.getAuthors())            // fetch the associated authors
 *      .forEach(author -> ... );            // iterate the collection
 * }
 *
 * @param association A {@linkplain FetchType#LAZY lazy}
 *                    association
 * @param <T> The type of the entity attribute type,
 *            either an entity type or a collection type
 *
 * @return The association with its state fetched.
 * @throws PersistenceException if a record could not be
 *         read from the database
 */
<T> T fetch(T association);

/**
 * Specify an {@linkplain Option option} controlling the
 * behavior of this entity agent, overwriting any existing
 * option of the same type.
 *
 * @param option the option
 */
void addOption(@Nonnull Option option);

/**
 * Get the {@linkplain Option options} controlling the behavior
 * of this entity agent. The returned set includes options set
 * via {@link #addOption}, along with options specified via
 * {@link #setCacheRetrieveMode} or {@link #setCacheStoreMode}.
 * Mutation of the returned set does not affect the options of
 * the entity agent.
 *
 * @return the options for this entity agent
 */
@Nonnull
Set<Option> getOptions();

/**
 * An option controlling the creation of an entity agent.
 *
 * <p>This interface may be implemented by custom provider-specific
 * options which extend the options defined by the specification.
 *
 * @since 4.0
 */
```

```
    interface CreationOption {
    }

    /**
     * An option controlling the behavior of an entity agent.
     * This provides a more type safe alternative to the use of
     * {@linkplain #setProperty properties and hints}.
     *
     * <p>This interface may be implemented by custom provider-specific
     * options which extend the options defined by the specification.
     *
     * @see CacheRetrieveMode
     * @see CacheStoreMode
     */
    interface Option extends CreationOption {
    }
}
```

B.4. EntityTransaction

```
package jakarta.persistence;

import jakarta.annotation.Nullable;

/**
 * Interface used to control transactions involving resource-local entity
 * managers and entity agents. An instance of {@code EntityTransaction}
 * may be obtained by calling {@link EntityHandler#getTransaction()}.
 *
 * @since 1.0
 */
public interface EntityTransaction {

    /**
     * Start a resource transaction.
     * @throws IllegalStateException if {@link #isActive()} is true
     */
    void begin();

    /**
     * Commit the current resource transaction, writing any unflushed
     * changes to the database.
     * @throws IllegalStateException if {@link #isActive()} is false
     * @throws RollbackException if the commit fails
     */
    void commit();

    /**
     * Roll back the current resource transaction.
     * @throws IllegalStateException if {@link #isActive()} is false
     * @throws PersistenceException if an unexpected error
     *         condition is encountered
     */
    void rollback();

    /**
     * Mark the current resource transaction so that the only possible
     * outcome of the transaction is for the transaction
     * to be rolled back.
     * @throws IllegalStateException if {@link #isActive()} is false
     */
    void setRollbackOnly();

    /**
     * Determine whether the current resource transaction has been
```

```
     * marked for rollback.
     * @return boolean indicating whether the transaction has been
     *         marked for rollback
     * @throws IllegalStateException if {@link #isActive()} is false
     */
    boolean getRollbackOnly();

    /**
     * Indicate whether a resource transaction is in progress.
     * @return boolean indicating whether transaction is in progress
     * @throws PersistenceException if an unexpected error
     *         condition is encountered
     */
    boolean isActive();

    /**
     * Set the transaction timeout, in seconds. This is a hint.
     * @param timeout the timeout, in seconds, or null to indicate
     *                that the database server should set the timeout
     * @since 3.2
     */
    void setTimeout(@Nullable Integer timeout);

    /**
     * The transaction timeout.
     * @since 3.2
     */
    @Nullable Integer getTimeout();
}
```

B.5. EntityManagerFactory

```
//     Linda DeMichiel - 2.0

package jakarta.persistence;

import java.lang.annotation.Annotation;
import java.util.Map;
import java.util.function.Consumer;
import java.util.function.Function;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.Metamodel;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.sql.ResultSetMapping;

/**
 * Interface used to interact with the persistence unit, and to
 * create new instances of {@link EntityManager} and
 * {@link EntityAgent}.
 *
 * <p>A persistence unit defines a set of managed classes that are
 * interrelated or grouped by the application and which map to a
 * single database. If two entity types participate in an association,
 * then they must belong to the same persistence unit. If two entity
 * types are stored in different databases, they belong to distinct
 * persistence units.
 *
 * <p>A persistence unit may be:
 * <ul>
 * <li>declared by a {@code <persistence-unit>} element of a
 *     {@code META-INF/persistence.xml} file, or
 * <li>be defined at runtime via the {@link PersistenceConfiguration}
 *     API.
```

```
* </ul>
* <p>Every persistence unit has a {@linkplain #getName name}. The
* name of the persistence unit must be unique within the module to
* which it belongs. In the Jakarta EE environment the name must be
* unique within a given EJB-JAR file, within a given WAR file,
* within a given application client JAR, or within an EAR.
*
* <p>Every persistence unit has a
* {@linkplain #getTransactionType transaction type}, either:
* <ul>
* <li>{@linkplain PersistenceUnitTransactionType#JTA JTA}, meaning
*     that transactions are managed by Jakarta Transactions, and
*     demarcated either declaratively or programmatically via the
*     {@code UserTransaction} interface, or
* <li>{@linkplain PersistenceUnitTransactionType#RESOURCE_LOCAL
*     resource-local}, meaning that transactions are managed by
*     the Persistence provider, and programmatically demarcated by
*     the application via the {@link EntityTransaction} interface.
* </ul>
*
* <p>An {@link EntityManagerFactory} with a lifecycle managed by
* the application may be created using the static operations of
* the {@link Persistence} class:
* <ul>
* <li>if the persistence unit is defined in {@code persistence.xml},
*     an entity manager factory may be created by calling
*     {@link Persistence#createEntityManagerFactory(String)} or
*     {@link Persistence#createEntityManagerFactory(String,Map)},
*     or
* <li>if the persistence unit was defined using
*     {@link PersistenceConfiguration}, an entity manager factory
*     may be created by calling
*     {@link PersistenceConfiguration#createEntityManagerFactory()}.
* </ul>
*
* <p>Usually, there is exactly one {@code EntityManagerFactory} for
* each persistence unit, created at system initialization time:
* {@snippet :
* // Create a factory for the persistence unit named 'Library',
* // reading its configuration from META-INF/persistence.xml
* static final EntityManagerFactory entityManagerFactory =
*         Persistence.createEntityManagerFactory("Library");
* }
* {@snippet :
* // Create a factory for a persistence unit named 'Library',
* // specifying its configuration programmatically
* static final EntityManagerFactory entityManagerFactory =
       new PersistenceConfiguration("Library")
*             .nonJtaDataSource("java:global/jdbc/LibraryDatabase")
*             .managedClass(Book.class)
*             .managedClass(Author.class)
*             .createEntityManagerFactory();
* }
* <p>Alternatively, in the Jakarta EE environment, a
* container-managed {@code EntityManagerFactory} may be obtained
* by dependency injection, using {@link PersistenceUnit}.
* {@snippet :
* // Inject the container-managed factory
* @PersistenceUnit(unitName="Library")
* EntityManagerFactory entityManagerFactory;
* }
*
* <p>An {@code EntityManager} or {@code EntityAgent} may be created
* via a call to, respectively, {@link #createEntityManager} or
* {@link #createEntityAgent}. However, this approach places complete
* responsibility for cleanup and exception management on the client and
* is thus considered error-prone. It is much safer to use the methods
```

```
 * {@link #runInTransaction} and {@link #callInTransaction} to obtain
 * entity managers and entity agents.
 * {@snippet :
 * entityManagerFactory.runInTransaction(entityManager -> entityManager.merge(book));
 * }
 *
 * <p>Alternatively, in the Jakarta EE environment, a container-managed
 * {@link EntityManager} may be obtained by dependency injection,
 * using {@link PersistenceContext}, and the application need not
 * interact with the {@code EntityManagerFactory} directly.
 * {@snippet :
 * // Inject the container-managed entity manager
 * @PersistenceContext(unitName="Library")
 * EntityManagerFactory entityManagerFactory;
 * }
 *
 * <p>The {@code EntityManagerFactory} provides access to certain
 * other useful APIs:
 * <ul>
 * <li>an instance of {@link Metamodel} exposing a model of the
 *     managed types associated with the persistence unit may be
 *     obtained by calling {@link #getMetamodel()},
 * <li>an instance of {@link SchemaManager}, allowing programmatic
 *     control over schema generation and validation, may be
 *     obtained by calling {@link #getSchemaManager()},
 * <li>an instance of {@link Cache}, allowing direct programmatic
 *     control over the second-level cache, may be obtained by
 *     calling {@link #getCache()},
 * <li>the {@link CriteriaBuilder}, used to define criteria queries,
 *     may be obtained by calling {@link #getCriteriaBuilder()},
 *     and
 * <li>the {@link PersistenceUnitUtil} may be obtained by calling
 *     {@link #getPersistenceUnitUtil()}.
 * </ul>
 *
 * <p>When the application has finished using the entity manager
 * factory, or when the application terminates, the application
 * should {@linkplain #close} the entity manager factory. If
 * necessary, a {@link java.lang.ref.Cleaner} may be used:
 * {@snippet :
 * // Destroy factory before program terminates
 * Cleaner.create()
 *         .register(entityManagerFactory, entityManagerFactory::close);
 * }
 * Once an {@code EntityManagerFactory} has been closed, all its
 * entity managers are considered to be in the closed state.
 *
 * @see EntityManager
 * @see EntityAgent
 *
 * @since 1.0
 */
public interface EntityManagerFactory extends AutoCloseable {

    /**
     * Create a new application-managed {@link EntityManager}. This
     * method returns a new {@code EntityManager} instance each time
     * it is invoked. The returned object must not be reused across
     * distinct units of work or shared between concurrent threads.
     * @param options options controlling the behavior of the entity
     *                manager
     * @return a new {@linkplain EntityManager#isOpen open} entity
     *         manager
     * @throws IllegalStateException if the entity manager factory
     *         has been closed
     */
    @Nonnull
```

```
EntityManager createEntityManager(@Nullable EntityManager.CreationOption... options);

/**
 * Create a new application-managed {@link EntityManager} with
 * the given {@link Map} specifying property settings. This
 * method returns a new {@code EntityManager} instance each time
 * it is invoked. The returned object must not be reused across
 * distinct units of work or shared between concurrent threads.
 * @param properties properties and hints affecting the behavior
 *                   of the returned entity manager
 * @return a new {@linkplain EntityManager#isOpen open} entity
 *         manager
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 * @apiNote The use of named properties lacks type safety compared
 * to the use of {@linkplain EntityManager.CreationOption options}.
 */
@Nonnull
EntityManager createEntityManager(@Nullable Map<?, ?> properties);

/**
 * Create a new application-managed JTA {@link EntityManager} with
 * the specified {@linkplain SynchronizationType synchronization
 * type} and given property settings. This method returns a new
 * {@code EntityManager} instance each time it is invoked. The
 * returned object must not be reused across distinct units of work
 * or shared between concurrent threads.
 * @param synchronizationType how and when the entity manager should
 *                            be synchronized with the current JTA
 *                            transaction
 * @param properties properties and hints affecting the behavior
 *                   of the returned entity manager
 * @return a new {@linkplain EntityManager#isOpen open} entity
 *         manager
 * @throws IllegalStateException if the entity manager factory has
 *         {@linkplain PersistenceUnitTransactionType#RESOURCE_LOCAL
 *         resource local} transaction type or has been closed
 * @apiNote The use of named properties lacks type safety compared
 * to the use of {@linkplain EntityManager.CreationOption options}.
 *
 * @since 2.1
 */
@Nonnull
EntityManager createEntityManager(@Nonnull SynchronizationType synchronizationType,
                                  @Nullable Map<?, ?> properties);

/**
 * Create a new application-managed {@link EntityAgent}. This
 * method returns a new {@code EntityAgent} instance each time
 * it is invoked. The returned object most not be shared
 * between concurrent threads.
 * @param options options controlling the behavior of the entity
 *                agent
 * @return a new {@linkplain EntityAgent#isOpen open} entity
 *         agent
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 *
 * @since 4.0
 */
@Nonnull
EntityAgent createEntityAgent(@Nullable EntityAgent.CreationOption... options);

/**
 * Create a new application-managed {@link EntityAgent} with
 * the given {@link Map} specifying property settings. This
 * method returns a new {@code EntityAgent} instance each time
```

```
 * it is invoked. The returned object must not be shared
 * between concurrent threads.
 * @param properties properties and hints affecting the behavior
 *                   of the returned entity agent
 * @return a new {@linkplain EntityAgent#isOpen open} entity
 *         agent
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 * @apiNote The use of named properties lacks type safety compared
 * to the use of {@linkplain EntityAgent.CreationOption options}.
 *
 * @since 4.0
 */
@Nonnull
EntityAgent createEntityAgent(@Nullable Map<?, ?> properties);

/**
 * Return an instance of {@link CriteriaBuilder} which may be used
 * to construct {@link jakarta.persistence.criteria.CriteriaQuery}
 * objects.
 * @return an instance of {@link CriteriaBuilder}
 * @throws IllegalStateException if the entity manager factory has
 *         been closed
 *
 * @see EntityHandler#getCriteriaBuilder()
 *
 * @since 2.0
 */
@Nonnull
CriteriaBuilder getCriteriaBuilder();

/**
 * Return an instance of the {@link Metamodel} interface for access
 * to the metamodel of the persistence unit.
 * @return an instance of {@link Metamodel}
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 *
 * @since 2.0
 */
@Nonnull
Metamodel getMetamodel();

/**
 * Indicates whether the factory is open. Returns true until the
 * factory has been closed.
 * @return boolean indicating whether the factory is open
 */
boolean isOpen();

/**
 * Close the factory, releasing any resources that it holds.
 * After a factory instance has been closed, all methods invoked
 * on it will throw the {@link IllegalStateException}, except
 * for {@link #isOpen}, which will return false. Once an
 * {@code EntityManagerFactory} has been closed, all its
 * entity managers are considered to be in the closed state.
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 */
void close();

/**
 * The name of the persistence unit.
 *
 * @since 3.2
 */
```

```
@Nonnull
String getName();

/**
 * Get the properties and associated values that are in effect
 * for the entity manager factory. Changing the contents of the
 * map does not change the configuration in effect.
 * @return properties
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 *
 * @since 2.0
 */
@Nonnull
Map<String, Object> getProperties();

/**
 * Access the cache that is associated with the entity manager
 * factory (the "second level cache").
 * @return an instance of {@link Cache}, or null if there is no
 * second-level cache in use
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 *
 * @since 2.0
 */
@Nullable
Cache getCache();

/**
 * Return interface providing access to utility methods for the
 * persistence unit.
 * @return an instance of {@link PersistenceUnitUtil}
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 *
 * @since 2.0
 */
@Nonnull
PersistenceUnitUtil getPersistenceUnitUtil();

/**
 * The type of transaction management used by this persistence
 * unit, either resource-local transaction management, or JTA.
 *
 * @since 3.2
 */
@Nonnull
PersistenceUnitTransactionType getTransactionType();

/**
 * Return interface providing access to schema management
 * operations for the persistence unit.
 * @return an instance of {@link SchemaManager}
 * @throws IllegalStateException if the entity manager factory
 *         has been closed
 *
 * @since 3.2
 */
@Nonnull
SchemaManager getSchemaManager();

/**
 * Define the query, typed query, or stored procedure query as
 * a named query such that future query objects can be created
 * from it using the {@link EntityHandler#createNamedQuery} or
 * {@link EntityHandler#createNamedStoredProcedureQuery} methods.
```

```
 * <p>Any configuration of the query object (except for actual
 * parameter binding) in effect when the named query is added
 * is retained as part of the named query definition. This
 * includes configuration information such as max results, hints,
 * flush mode, lock mode, result set mapping information, and
 * information about stored procedure parameters.
 * <p>When the query is executed, information that can be set by
 * means of the query APIs can be overridden. Information that is
 * overridden does not affect the named query as registered with
 * the entity manager factory, and thus does not affect subsequent
 * query objects created from it by calling {@code createNamedQuery}
 * or {@code createNamedStoredProcedureQuery}.
 * <p>If a named query of the same name has been previously defined,
 * either statically via metadata or via this method, that query
 * definition is replaced.
 *
 * @param name name for the query
 * @param query a {@link Query}, {@link TypedQuery},
 *             or {@link StoredProcedureQuery}
 *
 * @since 2.1
 */
void addNamedQuery(@Nonnull String name, @Nonnull Query query);

/**
 * Define the given {@link TypedQuery} as a named query such that
 * future query objects can be created from it by calling either
 * {@link EntityHandler#createQuery(TypedQueryReference)} or
 * {@link EntityHandler#createNamedQuery(String,Class)}. Return a
 * {@linkplain TypedQueryReference reference} to the query.
 * <p>Any configuration of the query object (except for actual
 * parameter binding) in effect when the named query is added
 * is retained as part of the named query definition. This
 * includes configuration information such as max results, hints,
 * flush mode, lock mode, result set mapping information, and
 * information about stored procedure parameters.
 * <p>When the query is executed, information that can be set by
 * means of the query APIs can be overridden. Information that is
 * overridden does not affect the named query as registered with
 * the entity manager factory, and thus does not affect subsequent
 * query objects created from it by calling {@code createNamedQuery}.
 * <p>If a named query of the same name has been previously defined,
 * either statically via metadata or via this method, that query
 * definition is replaced.
 *
 * @param name name for the query
 * @param query a {@link TypedQuery}
 * @param <R> the query result type
 * @return a reference to the query
 *
 * @since 4.0
 *
 * @see EntityHandler#createQuery(TypedQueryReference)
 * @see EntityHandler#createNamedQuery(String,Class)
 */
@Nonnull
<R> TypedQueryReference<R> addNamedQuery(@Nonnull String name,
                                         @Nonnull TypedQuery<R> query);

/**
 * Define the given {@link Statement} as a named statement such that
 * future statement objects can be created from it by calling either
 * {@link EntityHandler#createStatement(StatementReference)} or
 * {@link EntityHandler#createNamedStatement(String)}. Returns a
 * {@linkplain StatementReference reference} to the statement.
 * <p>Any configuration of the statement object (except for actual
 * parameter binding) in effect when the named query is added
```

```
 * is retained as part of the named query definition. This
 * includes configuration information such as hints,
 * and flush mode.
 * <p>When the statement is executed, information that can be set by
 * means of the statement APIs can be overridden. Information that is
 * overridden does not affect the named statement as registered with
 * the entity manager factory, and thus does not affect subsequent
 * statement objects created from it by calling {@code createNamedStatement}.
 * <p>If a named statement of the same name has been previously defined,
 * either statically via metadata or via this method, that statement
 * definition is replaced.
 *
 * @param name Name for the query.
 * @param statement The statement to convert into a {@code StatementReference} and store by name.
 * @return Reference to the stored statement.
 *
 * @see EntityHandler#createStatement(StatementReference)
 * @see EntityHandler#createNamedStatement(String)
 *
 * @since 4.0
 */
@Nonnull
StatementReference addNamedStatement(@Nonnull String name,
                                     @Nonnull Statement statement);

/**
 * Return an object of the specified type to allow access to
 * a provider-specific API. If the provider implementation of
 * {@code EntityManagerFactory} does not support the given
 * type, the {@link PersistenceException} is thrown.
 * @param cls the class of the object to be returned.
 *            This is usually either the underlying class
 *            implementing {@code EntityManagerFactory} or an
 *            interface it implements.
 * @param <T> the type of the object to be returned
 * @return an instance of the specified class
 * @throws PersistenceException if the provider does not support
 *         the given type
 * @since 2.1
 */
@Nonnull
<T> T unwrap(@Nonnull Class<T> cls);

/**
 * Add a named copy of the given {@link EntityGraph} to this
 * {@code EntityManagerFactory}. If an entity graph with the
 * given name already exists, it is replaced.
 * @param graphName  name for the entity graph
 * @param entityGraph  entity graph
 * @param <T> the entity graph type
 * @since 2.1
 */
<T> void addNamedEntityGraph(@Nonnull String graphName,
                             @Nonnull EntityGraph<T> entityGraph);

/**
 * A map keyed by {@linkplain NamedQuery#name query name}, containing
 * {@linkplain TypedQueryReference references} to every named query whose
 * result type is assignable to the given Java type.
 * @param resultType any Java type, including {@code Object.class}
 *                   meaning all queries
 * @return a map keyed by query name
 * @param <R> the specified upper bound on the query result types
 *
 * @see NamedQuery
 * @see NamedNativeQuery
 * @see #addNamedQuery(String, TypedQuery)
```

```
 *
 * @since 3.2
 */
@Nonnull
<R> Map<String, TypedQueryReference<R>> getNamedQueries(@Nonnull Class<R> resultType);

/**
 * A map keyed by {@linkplain NamedStatement#name query name}, containing
 * {@linkplain StatementReference references} to every named statement.
 * @return a map keyed by query name
 *
 * @see NamedStatement
 * @see NamedNativeStatement
 * @see #addNamedQuery(String, Query)
 *
 * @since 4.0
 */
@Nonnull
Map<String, StatementReference> getNamedStatements();

/**
 * A map keyed by {@linkplain NamedEntityGraph#name graph name}, containing
 * every named {@linkplain EntityGraph entity graph} whose entity type is
 * assignable to the given Java type.
 * @param entityType any Java type, including {@code Object.class}
 *                   meaning all entity graphs
 * @return a map keyed by graph name
 * @param <E> the specified upper bound on the entity graph types
 *
 * @see jakarta.persistence.metamodel.EntityType#getNamedEntityGraphs()
 *
 * @since 3.2
 */
@Nonnull
<E> Map<String, EntityGraph<? extends E>> getNamedEntityGraphs(@Nonnull Class<E> entityType);

/**
 * A map keyed by {@linkplain SqlResultSetMapping#name name}, containing
 * a typed {@link ResultSetMapping} for every result set mapping defined
 * {@linkplain SqlResultSetMapping in annotations} whose inferred result
 * type is exactly the given Java type.
 * @param resultType any Java type, including {@code Object.class}
 *                   meaning all result set mappings
 * @return a map keyed by query name
 * @param <R> the specified query result type
 *
 * @since 4.0
 */
@Nonnull
<R> Map<String, ResultSetMapping<R>> getResultSetMappings(@Nonnull Class<R> resultType);

/**
 * Register a listener for the given kind of entity lifecycle event
 * and given entity type, returning an {@link EntityListenerRegistration}
 * which may be used to {@linkplain EntityListenerRegistration#cancel cancel}
 * the registration.
 * <p>
 * The given entity type may be an entity class or any supertype of
 * an entity class, including {@code java.lang.Object}. The lifecycle
 * event type is identified by the class of a callback annotation type,
 * for example, {@link PostPersist PostPersist.class}. When a lifecycle
 * event of the given type occurs for any entity assignable to the given
 * entity type, the listener is invoked with the entity instance.
 *
 * @param entityType An entity class, or any supertype of an entity
 *                   class
 * @param callbackType The kind of entity callback, represented by
```

```
 *                     the class of the callback annotation type,
 *                     usually one of the following classes:
 *                     {@link PrePersist}, {@link PostPersist},
 *                     {@link PreRemove}, {@link PostRemove},
 *                     {@link PreMerge}, {@link PostLoad}; or
 *                     {@link PreInsert}, {@link PostInsert},
 *                     {@link PreUpdate}, {@link PostUpdate},
 *                     {@link PreUpsert}, {@link PostUpsert},
 *                     {@link PreDelete}, {@link PostDelete}.
 * @param listener The callback listener
 * @param <E> the entity type
 * @return An {@link EntityListenerRegistration} representing the
 *         registration of the given listener with this factory
 * @throws IllegalArgumentException if the given annotation type
 *         does not represent a recognized lifecycle callback
 *
 * @since 4.0
 */
@Nonnull
<E> EntityListenerRegistration addListener(
        @Nonnull Class<E> entityType,
        @Nonnull Class<? extends Annotation> callbackType,
        @Nonnull Consumer<? super E> listener);

/**
 * Create a new application-managed {@link EntityManager} with an active
 * transaction, and execute the given function, passing the {@code EntityManager}
 * to the function.
 * <p>
 * If the transaction type of the persistence unit is JTA, and there is a JTA
 * transaction already associated with the caller, then the {@code EntityManager}
 * is associated with this current transaction. If the given function throws an
 * exception, the JTA transaction is marked for rollback, and the exception is
 * rethrown.
 * <p>
 * Otherwise, if the transaction type of the persistence unit is resource-local,
 * or if there is no JTA transaction already associated with the caller, then
 * the {@code EntityManager} is associated with a new transaction. If the given
 * function returns without throwing an exception, this transaction is committed.
 * If the function does throw an exception, the transaction is rolled back, and
 * the exception is rethrown.
 * <p>
 * Finally, the {@code EntityManager} is closed before this method returns
 * control to the client.
 *
 * @param work a function to be executed in the scope of the transaction
 *
 * @since 3.2
 */
void runInTransaction(@Nonnull Consumer<EntityManager> work);
/**
 * Create a new application-managed {@link EntityManager} with an active
 * transaction, and call the given function, passing the {@code EntityManager}
 * to the function.
 * <p>
 * If the transaction type of the persistence unit is JTA, and there is a JTA
 * transaction already associated with the caller, then the {@code EntityManager}
 * is associated with this current transaction. If the given function returns
 * without throwing an exception, the result of the function is returned. If the
 * given function throws an exception, the JTA transaction is marked for rollback,
 * and the exception is rethrown.
 * <p>
 * Otherwise, if the transaction type of the persistence unit is resource-local,
 * or if there is no JTA transaction already associated with the caller, then
 * the {@code EntityManager} is associated with a new transaction. If the given
 * function returns without throwing an exception, this transaction is committed
 * and the result of the function is returned. If the function does throw an
```

```
 * exception, the transaction is rolled back, and the exception is rethrown.
* <p>
 * Finally, the {@code EntityManager} is closed before this method returns
 * control to the client.
 *
 * @param work a function to be called in the scope of the transaction
 * @param <R> the function result type
 * @return the value returned by the given function
 *
 * @since 3.2
 */
<R> R callInTransaction(@Nonnull Function<EntityManager, R> work);

/**
 * Create a new application-managed {@link EntityHandler} of the given type,
 * with an active transaction, and execute the given function, passing the
 * {@code EntityHandler} to the function.
 * <p>
 * If the transaction type of the persistence unit is JTA, and there is a JTA
 * transaction already associated with the caller, then the {@code EntityHandler}
 * is associated with this current transaction. If the given function throws an
 * exception, the JTA transaction is marked for rollback, and the exception is
 * rethrown.
 * <p>
 * Otherwise, if the transaction type of the persistence unit is resource-local,
 * or if there is no JTA transaction already associated with the caller, then
 * the {@code EntityHandler} is associated with a new transaction. If the given
 * function returns without throwing an exception, this transaction is committed.
 * If the function does throw an exception, the transaction is rolled back, and
 * the exception is rethrown.
 * <p>
 * Finally, the {@code EntityHandler} is closed before this method returns
 * control to the client.
 *
 * @param handlerClass the type of {@link EntityHandler} to create, for
 *                     example, {@link EntityAgent EntityAgent.class}.
 * @param work a function to be executed in the scope of the transaction
 * @param <H> the entity handler type
 *
 * @since 4.0
 */
<H extends EntityHandler> void runInTransaction(@Nonnull Class<H> handlerClass,
                                                @Nonnull Consumer<H> work);
/**
 * Create a new application-managed {@link EntityHandler} of the given type,
 * with an active transaction, and call the given function, passing the
 * {@code EntityHandler} to the function.
 * <p>
 * If the transaction type of the persistence unit is JTA, and there is a JTA
 * transaction already associated with the caller, then the {@code EntityHandler}
 * is associated with this current transaction. If the given function returns
 * without throwing an exception, the result of the function is returned. If the
 * given function throws an exception, the JTA transaction is marked for rollback,
 * and the exception is rethrown.
 * <p>
 * Otherwise, if the transaction type of the persistence unit is resource-local,
 * or if there is no JTA transaction already associated with the caller, then
 * the {@code EntityHandler} is associated with a new transaction. If the given
 * function returns without throwing an exception, this transaction is committed
 * and the result of the function is returned. If the function does throw an
 * exception, the transaction is rolled back, and the exception is rethrown.
 * <p>
 * Finally, the {@code EntityHandler} is closed before this method returns
 * control to the client.
 *
 * @param handlerClass the type of {@link EntityHandler} to create, for
 *                     example, {@link EntityAgent EntityAgent.class}.
```

```
     * @param work a function to be called in the scope of the transaction
     * @param <R> the function result type
     * @param <H> the entity handler type
     * @return the value returned by the given function
     *
     * @since 4.0
     */
    <R, H extends EntityHandler> R callInTransaction(@Nonnull Class<H> handlerClass,
                                                     @Nonnull Function<H,R> work);
}
```

B.6. LockModeType

```
/**
 * Enumerates the kinds of optimistic or pessimistic lock which may
 * be obtained on an entity instance.
 *
 * <p>A specific lock mode may be requested by passing an explicit
 * {@code LockModeType} as an argument to:
 * <ul>
 * <li>one of the methods of {@link EntityManager} which obtains
 *     locks ({@link EntityManager#lock lock()},
 *     {@link EntityManager#find find()}, or
 *     {@link EntityManager#refresh refresh()}), or
 * <li>to {@link TypedQuery#setLockMode(LockModeType)}.
 * </ul>
 * <p>Lock modes other than {@link #NONE} prevent dirty reads and
 * non-repeatable reads of locked entity data within the current
 * transaction.
 *
 * <p>An optimistic lock is requested by specifying the lock mode
 * type {@link #OPTIMISTIC} or {@link #OPTIMISTIC_FORCE_INCREMENT}.
 * The lock mode types {@link #READ} and {@link #WRITE} are synonyms
 * for {@code OPTIMISTIC} and {@code OPTIMISTIC_FORCE_INCREMENT}
 * respectively. The latter are preferred for new applications.
 *
 * <p>Optimistic locking usually requires a {@linkplain Version
 * version}. The persistence implementation is not required to
 * accept requests for optimistic locks on non-versioned entities.
 * When a provider rejects such a lock request, it must throw
 * {@link PersistenceException}.
 *
 * <p>An immediate pessimistic database-level lock is requested by
 * specifying one of the lock mode types {@link #PESSIMISTIC_READ},
 * {@link #PESSIMISTIC_WRITE}, or {@link #PESSIMISTIC_FORCE_INCREMENT}.
 * <ul>
 * <li>A lock with type {@code PESSIMISTIC_WRITE} can be obtained on
 *     an entity instance to force serialization among transactions
 *     attempting to update the entity data.
 * <li>A lock with type {@code PESSIMISTIC_READ} can be used to query
 *     data using repeatable-read semantics without the need to reread
 *     the data at the end of the transaction to obtain a lock, and
 *     without blocking other transactions reading the data.
 * <li>A lock with type {@code PESSIMISTIC_WRITE} can be used when
 *     querying data and there is a high likelihood of deadlock or
 *     update failure among concurrent updating transactions.
 * </ul>
 *
 * <p>The persistence implementation must accept requests for locks
 * of type {@code PESSIMISTIC_READ} and {@code PESSIMISTIC_WRITE} on
 * both versioned and non-versioned entities.
 * <ul>
 * <li>When the lock cannot be obtained, and the database locking
 *     failure results in transaction-level rollback, the provider
 *     must throw {@link PessimisticLockException} and ensure that
```

```
 *     the JTA transaction or {@code EntityTransaction} has been
 *     marked for rollback.
 * <li>When the lock cannot be obtained, and the database locking
 *     failure results in only statement-level rollback, the provider
 *     must throw the {@link LockTimeoutException} (and must not mark
 *     the transaction for rollback).</li>
 * </ul>
 * <p>The persistence implementation is not required to accept
 * requests for locks of type {@code PESSIMISTIC_FORCE_INCREMENT} on
 * non-versioned entities. When a provider does not support such a
 * lock request, it must throw {@link PersistenceException}.
 *
 * @since 1.0
 *
 */
public enum LockModeType
        implements FindOption, RefreshOption, TypedQuery.Option {
    /**
     * Synonymous with {@link #OPTIMISTIC}.
     * <p>
     * {@code OPTIMISTIC} is preferred for new applications.
     *
     */
    READ,

    /**
     * Synonymous with {@link #OPTIMISTIC_FORCE_INCREMENT}.
     * <p>
     * {@code OPTIMISTIC_FORCE_INCREMENT} is preferred for
     * new applications.
     *
     */
    WRITE,

    /**
     * An optimistic lock.
     *
     * <p>If transaction T1 calls for a lock of type
     * {@code OPTIMISTIC} on a versioned object, the entity manager
     * must ensure that neither of the following phenomena can occur:
     * <ul>
     * <li>P1 <b>Dirty read</b>: Transaction T1 modifies a row.
     *     Another transaction T2 then reads that row and obtains the
     *     modified value, before T1 has committed or rolled back.
     *     Transaction T2 eventually commits successfully; it does not
     *     matter whether T1 commits or rolls back and whether it does
     *     so before or after T2 commits.
     * </li>
     * <li>P2 <b>Non-repeatable read</b>: Transaction T1 reads a row.
     *     Another transaction T2 then modifies or deletes that row,
     *     before T1 has committed. Both transactions eventually commit
     *     successfully.
     * </li>
     * </ul>
     *
     * <p>The persistence implementation is not required to accept
     * requests for {@code OPTIMISTIC} locks on non-versioned entities.
     *
     * @since 2.0
     */
    OPTIMISTIC,

    /**
     * Optimistic lock, with version update, which prevents the
     * following phenomena:
     *
     * <p>If transaction T1 calls for a lock of type
```

```
 * {@code OPTIMISTIC_FORCE_INCREMENT} on a versioned object,
 * the entity manager must ensure that neither of the following
 * phenomena can occur:
 * <ul>
 * <li>P1 <b>Dirty read</b>: Transaction T1 modifies a row.
 *     Another transaction T2 then reads that row and obtains the
 *     modified value, before T1 has committed or rolled back.
 *     Transaction T2 eventually commits successfully; it does not
 *     matter whether T1 commits or rolls back and whether it does
 *     so before or after T2 commits.
 * </li>
 * <li>P2 <b>Non-repeatable read</b>: Transaction T1 reads a row.
 *     Another transaction T2 then modifies or deletes that row,
 *     before T1 has committed. Both transactions eventually commit
 *     successfully.
 * </li>
 * </ul>
 *
 * <p>In addition, obtaining a lock of type
 * {@code OPTIMISTIC_FORCE_INCREMENT} on a versioned entity will
 * also force an update (increment) to the {@linkplain Version
 * version column}.
 *
 * <p>The persistence implementation is not required to accept
 * requests for {@code OPTIMISTIC_FORCE_INCREMENT} locks on
 * non-versioned entities.
 *
 * @since 2.0
 */
OPTIMISTIC_FORCE_INCREMENT,

/**
 * Pessimistic read lock.
 *
 * <p>If transaction T1 calls for a lock of type
 * {@code PESSIMISTIC_READ} on an object, the entity manager must
 * ensure that neither of the following phenomena can occur:
 * <ul>
 * <li>P1 <b>Dirty read</b>: Transaction T1 modifies a row.
 *     Another transaction T2 then reads that row and obtains the
 *     modified value, before T1 has committed or rolled back.
 * </li>
 * <li>P2 <b>Non-repeatable read</b>: Transaction T1 reads a row.
 *     Another transaction T2 then modifies or deletes that row,
 *     before T1 has committed or rolled back.
 * </ul>
 *
 * @since 2.0
 */
PESSIMISTIC_READ,

/**
 * Pessimistic write lock.
 *
 * <p>If transaction T1 calls for a lock of type
 * {@code PESSIMISTIC_WRITE} on an object, the entity manager must
 * ensure that neither of the following phenomena can occur:
 * <ul>
 * <li>P1 <b>Dirty read</b>: Transaction T1 modifies a row.
 *     Another transaction T2 then reads that row and obtains the
 *     modified value, before T1 has committed or rolled back.
 * </li>
 * <li>P2 <b>Non-repeatable read</b>: Transaction T1 reads a row.
 *     Another transaction T2 then modifies or deletes that row,
 *     before T1 has committed or rolled back.
 * </ul>
 *
```

```
     * @since 2.0
     */
    PESSIMISTIC_WRITE,

    /**
     * Pessimistic write lock, with version update.
     *
     * <p>If transaction T1 calls for a lock of type
     * {@code PESSIMISTIC_FORCE_INCREMENT} on an object, the entity
     * manager must ensure that neither of the following phenomena
     * can occur:
     * <ul>
     * <li>P1 <b>Dirty read</b>: Transaction T1 modifies a row.
     *     Another transaction T2 then reads that row and obtains the
     *     modified value, before T1 has committed or rolled back.
     * </li>
     * <li>P2 <b>Non-repeatable read</b>: Transaction T1 reads a row.
     *     Another transaction T2 then modifies or deletes that row,
     *     before T1 has committed or rolled back.
     * </ul>
     *
     * <p>In addition, obtaining a lock of type
     * {@code PESSIMISTIC_FORCE_INCREMENT} on a versioned entity will
     * also force an update (increment) to the {@linkplain Version
     * version column}.
     *
     * <p>The persistence implementation is not required to accept
     * requests for {@code PESSIMISTIC_FORCE_INCREMENT} locks on
     * non-versioned entities.
     *
     * @since 2.0
     */
    PESSIMISTIC_FORCE_INCREMENT,

    /**
     * No lock.
     *
     * @since 2.0
     */
    NONE
}
```

B.7. PessimisticLockScope

```
package jakarta.persistence;

/**
 * Controls how a {@linkplain LockModeType pessimistic lock}
 * applied to an entity affects associated collections and
 * relationships.
 * <p>
 * A {@code PessimisticLockScope} may be passed:
 * <ul>
 * <li>as a {@link FindOption}, {@link RefreshOption},
 *     or {@link LockOption},
 * <li>as an argument to {@link TypedQuery#setLockScope}, or
 * <li>as the value of {@link NamedQuery#lockScope}.
 * </ul>
 * <p>Alternatively, it may be specified via the configuration
 * property {@code jakarta.persistence.lock.scope}. This property
 * may be passed as an argument to methods of {@link EntityManager}
 * which accept a {@link LockModeType}, or to {@link Query#setHint},
 * or it may be specified via {@link NamedQuery#hints}.
 *
 * @since 2.0
```

```
 */
public enum PessimisticLockScope
        implements FindOption, RefreshOption, LockOption, TypedQuery.Option {

    /**
     * <p>The persistence provider must lock the row or rows of the
     * {@linkplain Table primary} and {@linkplain SecondaryTable secondary}
     * tables mapped by the entity. In the case of a
     * {@linkplain InheritanceType#JOINED joined} inheritance hierarchy,
     * this includes all rows in tables mapped by the entity superclasses
     * of the entity.
     *
     * <p>The pessimistic lock does not extend to rows of:
     * <ul>
     * <li>tables mapped by associated entities,
     * <li>{@linkplain JoinTable join tables} mapped by associations,
     *     or
     * <li>{@linkplain CollectionTable collection tables}.
     * </ul>
     *
     * <p>This is the default behavior of pessimistic locking.
     */
    NORMAL,

    /**
     * <p>The persistence provider must lock the row or rows of:
     * <ul>
     * <li>the {@linkplain Table primary} and
     *     {@linkplain SecondaryTable secondary} tables mapped by the
     *     entity, including all rows in tables mapped by the entity
     *     superclasses of the entity in the case of a
     *     {@linkplain InheritanceType#JOINED joined} inheritance
     *     hierarchy,
     * <li>{@linkplain JoinTable join tables} mapped by associations
     *     owned by the entity, and
     * <li>{@linkplain CollectionTable collection tables} mapped by
     *     collections belonging to the entity.
     * </ul>
     *
     * <p>The pessimistic lock does not extend to rows of:
     * <ul>
     * <li>tables mapped by associated entities,
     * <li>{@linkplain JoinTable join tables} mapped by associations
     *     not owned by the entity.
     * </ul>
     */
    EXTENDED,

    /**
     * <p>The persistence provider must lock the row or rows of:
     * <ul>
     * <li>the {@linkplain Table primary} and
     *     {@linkplain SecondaryTable secondary} tables mapped by the
     *     entity, including all rows in tables mapped by the entity
     *     superclasses of the entity in the case of a
     *     {@linkplain InheritanceType#JOINED joined} inheritance
     *     hierarchy,
     * <li>{@linkplain JoinTable join tables} mapped by associations
     *     which were fetched as part of the operation which obtained
     *     the pessimistic lock, along with the primary and secondary
     *     tables mapped by the associated entities,
     * <li>{@linkplain CollectionTable collection tables} mapped by
     *     collections belonging to the entity which were fetched as
     *     part of the operation which obtained the pessimistic lock.
     * </ul>
     *
     * <p>The pessimistic lock does not extend to data which was not
```

```
     * fetched as part of the operation which obtained the pessimistic
     * lock.
     *
     * @since 4.0
     */
    FETCHED
}
```

B.8. Cache

```
import jakarta.annotation.Nonnull;

/**
 * Interface used to interact with the second-level cache.
 * If no second-level cache is in use, the methods of this
 * interface have no effect, except for {@link #contains},
 * which returns false.
 *
 * @since 2.0
 */
public interface Cache {

    /**
     * Whether the cache contains data for the given entity.
     * @param cls  entity class
     * @param primaryKey  primary key
     * @return boolean indicating whether the entity is in the cache
     */
    boolean contains(@Nonnull Class<?> cls, @Nonnull Object primaryKey);

    /**
     * Remove the data for the given entity from the cache.
     * @param cls  entity class
     * @param primaryKey  primary key
     */
    void evict(@Nonnull Class<?> cls, @Nonnull Object primaryKey);

    /**
     * Remove the data for entities of the specified class
     * (and its subclasses) from the cache.
     * @param cls  entity class
     */
    void evict(@Nonnull Class<?> cls);

    /**
     * Clear the cache.
     */
    void evictAll();

    /**
     * Return an object of the specified type to allow access to
     * the provider-specific API. If the provider's implementation
     * of the {@code Cache} interface does not support the specified
     * class, the {@link PersistenceException} is thrown.
     * @param cls  the class of the object to be returned.
     *             This is usually either the underlying class
     *             implementing {@code Cache}, or an interface it
     *             implements.
     * @param <T> the type of the object to be returned
     * @return an instance of the specified type
     * @throws PersistenceException if the provider does not support
     *         the given type
     * @since 2.1
     */
    @Nonnull
```

```
    <T> T unwrap(@Nonnull Class<T> cls);
}
```

B.9. Query

```
//     Linda DeMichiel - 2.0

package jakarta.persistence;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.Type;

import java.util.Calendar;
import java.util.Date;
import java.util.List;
import java.util.Set;
import java.util.Map;
import java.util.stream.Stream;

/**
 * Declares common operations for controlling the execution of
 * statements and queries written in the Jakarta Persistence
 * query language or in native SQL.
 * <ul>
 * <li>For a Jakarta Persistence {@code UPDATE} or {@code DELETE}
 *     statement, or for a native SQL statement that returns a row
 *     count, an instance of {@link Statement} should be used to
 *     execute the statement.
 * <li>For a Jakarta Persistence {@code SELECT} query or for any
 *     native SQL query that returns a result set, an instance of
 *     {@link TypedQuery} should be used.
 * <li>For a stored procedure call, a {@link StoredProcedureQuery}
 *     should be used.
 * </ul>
 *
 * @apiNote Every operation only relevant to {@code SELECT} queries,
 * for example, {@link #getResultList} and {@link #setMaxResults},
 * is now declared deprecated by this interface. Such operations
 * should be invoked via the {@link TypedQuery} interface. Similarly,
 * the operation {@link #executeUpdate}, which was only used to
 * execute statements, is declared as deprecated; the new operation
 * {@link Statement#execute} should be used instead.
 *
 * @see Statement
 * @see TypedQuery
 * @see StoredProcedureQuery
 * @see StatementOrTypedQuery
 * @see Parameter
 *
 * @since 1.0
 */
public interface Query {

    /**
     * Execute a {@code SELECT} query or a native query that returns
     * a result set and return the query results as an untyped
     * {@link List}. If necessary, first synchronize changes with the
     * database by flushing the persistence context.
     *
     * @return a list of the results, or an empty list if there are
     *         no results
     * @throws IllegalStateException if called for a Jakarta
     *         Persistence query language UPDATE or DELETE statement
     * @throws QueryTimeoutException if the query execution exceeds
```

```
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws TransactionRequiredException if a lock mode other than
 *         {@code NONE} has been set and there is no transaction
 *         or the persistence context has not been joined to the
 *         transaction
 * @throws PessimisticLockException if pessimistic locking
 *         fails and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking
 *         fails and only the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 * @deprecated This method returns a raw type.
 *             Use {@link TypedQuery#getResultList} to execute
 *             queries.
 */
@SuppressWarnings("rawtypes")
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
List getResultList();

/**
 * Execute a {@code SELECT} query or a native query that returns
 * a result set and return the query results as an untyped
 * {@link Stream}. If necessary, first synchronize changes with
 * the database by flushing the persistence context.
 * <p>By default, this method delegates to {@code getResultList().stream()},
 * The persistence provider may choose to override this method to
 * provide additional capabilities.
 *
 * @return a stream of the results, or an empty stream if there
 *         are no results
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language UPDATE or DELETE statement
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws TransactionRequiredException if a lock mode other than
 *         {@code NONE} has been set and there is no transaction
 *         or the persistence context has not been joined to the
 *         transaction
 * @throws PessimisticLockException if pessimistic locking
 *         fails and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking
 *         fails and only the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 * @see Stream
 * @see #getResultList()
 * @since 2.2
 * @deprecated This method returns a raw type.
 *             Use {@link TypedQuery#getResultStream} to execute
 *             queries
 */
@SuppressWarnings("rawtypes")
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
default Stream getResultStream() {
    return getResultList().stream();
```

```
}

/**
 * Execute a {@code SELECT} query or a native query that returns
 * a result set, returning a single untyped result. If necessary,
 * first synchronize changes with the database by flushing the
 * persistence context.
 *
 * @return the result
 * @throws NoResultException if there is no result
 * @throws NonUniqueResultException if more than one result
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language UPDATE or DELETE statement
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws TransactionRequiredException if a lock mode other than
 *         {@code NONE} has been set and there is no transaction
 *         or the persistence context has not been joined to the
 *         transaction
 * @throws PessimisticLockException if pessimistic locking
 *         fails and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking
 *         fails and only the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 * @deprecated Use {@link TypedQuery#getSingleResult}
 *             to execute queries
 */
@Deprecated(since = "4.0", forRemoval = true)
Object getSingleResult();

/**
 * Execute a {@code SELECT} query or a native query that returns
 * a result set, returning a single untyped result, or {@code null}
 * if the query has no results. If necessary, first synchronize
 * changes with the database by flushing the persistence context.
 *
 * @return the result, or {@code null} if there is no result
 * @throws NonUniqueResultException if more than one result
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language UPDATE or DELETE statement
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws TransactionRequiredException if a lock mode other than
 *         {@code NONE} has been set and there is no transaction
 *         or the persistence context has not been joined to the
 *         transaction
 * @throws PessimisticLockException if pessimistic locking
 *         fails and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking
 *         fails and only the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 *
 * @since 3.2
 * @deprecated Use {@link TypedQuery#getSingleResult}
 *             to execute queries
```

```
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nullable
Object getSingleResultOrNull();

/**
 * Execute an {@code UPDATE} or {@code DELETE} statement or a
 * native SQL statement that returns a row count.
 * <p>
 * After execution of a bulk update or delete operation, the
 * persistence provider is not required to resynchronize state
 * held in memory with the effects of the operation on data
 * held in the database. However, when this method is called
 * within a transaction, the persistence context is joined to
 * the transaction, and {@link FlushModeType#AUTO} is in effect,
 * the persistence provider must ensure that every modification
 * to the state of every entity associated with the persistence
 * context which could possibly alter the effects of the bulk
 * update or delete operation is visible to the processing of
 * the operation.
 *
 * @return the number of entities updated or deleted, or the
 *         row count of the native SQL statement
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language SELECT statement or for
 *         a criteria query
 * @throws TransactionRequiredException if there is
 *         no transaction or the persistence context has not
 *         been joined to the transaction
 * @throws QueryTimeoutException if the statement execution
 *         exceeds the query timeout value set and only
 *         the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 * @deprecated Use {@link Statement#execute}.
 */
@Deprecated(since = "4.0", forRemoval = true)
int executeUpdate();

/**
 * Set the maximum number of results returned to the client.
 * If the query has more results than the given maximum,
 * results are excluded from the list returned by
 * {@link #getResultList()}, so that only the given number
 * of results is returned. If the query returns results
 * with a well-defined order, the excluded results must be
 * those which would otherwise occur later in the list.
 *
 * @param maxResult The maximum number of results
 * @return the same query instance
 * @throws IllegalArgumentException if the argument is negative
 * @deprecated Use {@link TypedQuery#setMaxResults}
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
Query setMaxResults(int maxResult);

/**
 * The maximum number of results returned to the client,
 * as specified by {@link #setMaxResults}, or
 * {@value Integer#MAX_VALUE} if {@link #setMaxResults}
 * was not called.
 *
```

```
 * @return the maximum number of results
 * @since 2.0
 * @deprecated Use {@link TypedQuery#getMaxResults}
 */
@Deprecated(since = "4.0", forRemoval = true)
int getMaxResults();

/**
 * Set the position of the first query result returned to
 * the client. The given number of results is excluded
 * from the list returned by {@link #getResultList()}.
 * If the query returns results with a well-defined order,
 * the excluded results must be those which would otherwise
 * occur earlier in the list.
 *
 * @param startPosition The position of the first result,
 *                      numbered from {@code 0}
 * @return the same query instance
 * @throws IllegalArgumentException if the argument is negative
 * @deprecated Use {@link TypedQuery#setFirstResult}
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
Query setFirstResult(int startPosition);

/**
 * The position of the first result returned to the client,
 * as specified by {@link #setFirstResult}, or {@code 0} if
 * {@link #setFirstResult} was not called.
 *
 * @return the position of the first result
 * @since 2.0
 * @deprecated Use {@link TypedQuery#getFirstResult}
 */
@Deprecated(since = "4.0", forRemoval = true)
int getFirstResult();

/**
 * Set a query property or hint. Properties defined by this
 * specification must be observed by the persistence provider.
 * Vendor-specific hints that are not recognized by a provider
 * must be silently ignored. Portable applications should not
 * rely on the standard timeout hint; depending on the database
 * in use and the locking mechanisms used by the provider,
 * this hint may or may not be observed.
 *
 * @param hintName The name of the property or hint
 * @param value The value for the property or hint
 * @return the same query instance
 * @throws IllegalArgumentException if the given property or
 *         hint name is recognized by the provider, but the
 *         second argument is not a legal value for the given
 *         property or hint
 *
 * @apiNote The use of named query properties and hints lacks
 *          type safety compared to the use of
 *          {@linkplain TypedQuery.Option query options} or
 *          {@linkplain Statement.Option statement options}.
 */
@Nonnull
Query setHint(@Nonnull String hintName, @Nullable Object value);

/**
 * Get the properties and hints and associated values that are in
 * effect for the query instance. Modifying the contents of the
 * returned map does not change the configuration in effect.
 *
```

```
 * @return a copy of the query properties and hints currently in
 *         effect
 * @since 2.0
 */
@Nonnull
Map<String, Object> getHints();

/**
 * Bind an argument to a parameter of this query respresented as
 * a {@link Parameter} object.
 *
 * @param parameter The parameter object
 * @param value The argument to the parameter
 * @param <T> the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter
 *         does not correspond to a parameter of the
 *         query
 * @since 2.0
 */
@Nonnull
<T> Query setParameter(@Nonnull Parameter<T> parameter, @Nullable T value);

/**
 * Bind an instance of {@link Calendar} to a {@link Parameter} object.
 *
 * @param param The parameter object
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter does not
 *         correspond to a parameter of the query
 * @since 2.0
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2")
@Nonnull
Query setParameter(@Nonnull Parameter<Calendar> param,
                   @Nullable Calendar value,
                   @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a {@link Parameter} object.
 *
 * @param param The parameter object
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter does not
 *         correspond to a parameter of the query
 * @since 2.0
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2")
@Nonnull
Query setParameter(@Nonnull Parameter<Date> param,
                   @Nullable Date value,
                   @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @return the same query instance
```

```
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 */
@Nonnull
Query setParameter(@Nonnull String name, @Nullable Object value);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the argument might be {@code null}, especially in the case
 * of a native query.
 * {@snippet :
 * em.createNativeQuery("update books set pub_date = :date where isbn = :ISBN")
 *     .setParameter("date", optionalPublicationDate, LocalDate.class)
 *     .setParameter("ISBN", isbn)
 *     .executeUpdate();
 * }
 * {@snippet :
 * var books =
 *     em.createNativeQuery("select * from books where :limit is null or pub_date > :limit",
 *                                 Book.class)
 *         .setParameter("limit", optionalDateLimit, LocalDate.class)
 *         .getResultList();
 * }
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param type A class object representing the parameter type
 * @param <P> the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Nonnull
<P> Query setParameter(@Nonnull String name, @Nullable P value,
                       @Nonnull Class<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the binding is affected by an attribute converter.
 * {@snippet :
 * var amount = MonetaryAmount.of(priceLimit, currency);
 * var affordableBooks =
 *     em.createQuery("from Book where price < :amount")
 *         .setParameter("amount", amount, Book_.price.getType())
 *         .getResultList();
 * }
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param type The {@link Type} of the parameter
 * @param <P> the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Nonnull
<P> Query setParameter(@Nonnull String name, @Nullable P value,
                       @Nonnull Type<P> type);

/**
```

```
 * Bind an argument value to a named parameter, explicitly
 * specifying an {@linkplain AttributeConverter attribute
 * converter} to use.
 * {@snippet :
 * var amount = MonetaryAmount.of(priceLimit, currency);
 * var affordableBooks =
 *     em.createQuery("from Book where price < :amount")
 *         .setConvertedParameter("amount", amount,
 *                 MonetaryAmountConverter.class)
 *         .getResultList();
 * }
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param converter The class of the attribute converter
 * @param <P> the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Nonnull
<P> Query setConvertedParameter(@Nonnull String name, @Nullable P value,
                                @Nonnull Class<? extends AttributeConverter<P,?>> converter);

/**
 * Bind an instance of {@link Calendar} to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the value argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2")
@Nonnull
Query setParameter(@Nonnull String name, @Nullable Calendar value,
                   @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the value argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2")
@Nonnull
Query setParameter(@Nonnull String name, @Nullable Date value,
                   @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a positional parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
```

```
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 */
@Nonnull
Query setParameter(int position, @Nullable Object value);

/**
 * Bind an argument value to a positional parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the argument might be {@code null}, especially in the case
 * of a native SQL query.
 * {@snippet :
 * em.createNativeQuery("update books set pub_date = ?1 where isbn = ?2")
 *     .setParameter(1, optionalPublicationDate, LocalDate.class)
 *     .setParameter(2, isbn)
 *     .executeUpdate();
 * }
 * {@snippet :
 * var books =
 *     em.createNativeQuery("select * from books where ?1 is null or pub_date > ?1",
 *                                 Book.class)
 *         .setParameter(1, optionalDateLimit, LocalDate.class)
 *         .getResultList();
 * }
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param type A class object representing the parameter type
 * @param <P> the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Nonnull
<P> Query setParameter(int position, @Nullable P value,
                       @Nonnull Class<P> type);

/**
 * Bind an argument value to a positional parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the binding is affected by an attribute converter.
 * {@snippet :
 * var amount = MonetaryAmount.of(priceLimit, currency);
 * var affordableBooks =
 *     em.createQuery("from Book where price < ?1")
 *         .setParameter(1, amount, Book_.price.getType())
 *         .getResultList();
 * }
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param type The {@link Type} of the parameter
 * @param <P> the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @since 4.0
 */
@Nonnull
<P> Query setParameter(int position, @Nullable P value,
                       @Nonnull Type<P> type);
```

```
/**
 * Bind an argument value to a named parameter, explicitly
 * specifying an {@linkplain AttributeConverter attribute
 * converter} to use.
 * {@snippet :
 * var amount = MonetaryAmount.of(priceLimit, currency);
 * var affordableBooks =
 *     em.createQuery("from Book where price < ?1")
 *         .setConvertedParameter(1, amount,
 *                 MonetaryAmountConverter.class)
 *         .getResultList();
 * }
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param converter The class of the attribute converter
 * @param <P> the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Nonnull
<P> Query setConvertedParameter(int position, @Nullable P value,
                                @Nonnull Class<? extends AttributeConverter<P,?>> converter);

/**
 * Bind an instance of {@link Calendar} to a positional
 * parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2")
@Nonnull
Query setParameter(int position, @Nullable Calendar value,
                   @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a positional
 * parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2")
@Nonnull
Query setParameter(int position, @Nullable  Date value,
                   @Nonnull TemporalType temporalType);

/**
 * Get the {@link Parameter} objects representing the declared
```

```
 * parameters of the query or an empty set if the query has no
 * parameters.
 * This method is not required to be supported for native
 * queries.
 *
 * @return a set containing the parameter objects
 * @throws IllegalStateException if invoked on a native
 *         query when the implementation does not support
 *         this use
 * @since 2.0
 */
@Nonnull
Set<Parameter<?>> getParameters();

/**
 * Get the {@link Parameter} object representing the declared
 * named parameter with the given name.
 * This method is not required to be supported for native
 * queries.
 *
 * @param name The name of the parameter
 * @return The parameter object representing the named
 *         parameter
 * @throws IllegalArgumentException if the parameter of the
 *         specified name does not exist
 * @throws IllegalStateException if invoked on a native
 *         query when the implementation does not support
 *         this use
 * @since 2.0
 */
@Nonnull
Parameter<?> getParameter(@Nonnull String name);

/**
 * Get the {@link Parameter} object representing the declared
 * named parameter with the given name and type.
 * This method is required to be supported for criteria queries
 * only.
 *
 * @param name The name of the parameter
 * @param type A class object representing the parameter type
 * @param <T> the parameter type
 * @return The parameter object representing the named
 *         parameter
 * @throws IllegalArgumentException if the parameter of the
 *         specified name does not exist or is not assignable
 *         to the type
 * @throws IllegalStateException if invoked on a native
 *         query or Jakarta Persistence query language query when
 *         the implementation does not support this use
 * @since 2.0
 */
@Nonnull
<T> Parameter<T> getParameter(@Nonnull String name,
                              @Nonnull Class<T> type);

/**
 * Get the {@link Parameter} object representing the declared
 * positional parameter with the given position.
 * This method is not required to be supported for native
 * queries.
 *
 * @param position The parameter position
 * @return The parameter object representing the positional
 *         parameter
 * @throws IllegalArgumentException if the parameter with the
 *         specified position does not exist
```

```
 * @throws IllegalStateException if invoked on a native
 *         query when the implementation does not support
 *         this use
 * @since 2.0
 */
@Nonnull
Parameter<?> getParameter(int position);

/**
 * Get the {@link Parameter} object corresponding to the declared
 * positional parameter with the given position and type.
 * This method is not required to be supported by the provider.
 *
 * @param position The parameter position
 * @param type A class object representing the parameter type
 * @param <T> the parameter type
 * @return The parameter object representing the positional
 *         parameter
 * @throws IllegalArgumentException if the parameter with the
 *         specified position does not exist or is not assignable
 *         to the type
 * @throws IllegalStateException if invoked on a native
 *         query or Jakarta Persistence query language query when
 *         the implementation does not support this use
 * @since 2.0
 */
@Nonnull
<T> Parameter<T> getParameter(int position, @Nonnull Class<T> type);

/**
 * Return a boolean value indicating whether an argument has
 * been bound to the parameter represented by the given
 * parameter object.
 *
 * @param parameter The parameter object
 * @return {@code true} an argument has been bound, or
 *         {@code false} otherwise
 * @since 2.0
 */
boolean isBound(@Nonnull Parameter<?> parameter);

/**
 * Return the input value bound to the parameter.
 * (Note that OUT parameters are unbound.)
 *
 * @param parameter The parameter object
 * @param <T> the parameter type
 * @return parameter value
 * @throws IllegalArgumentException if the parameter is not
 *         a parameter of the query
 * @throws IllegalStateException if the parameter has not
 *         been bound
 * @since 2.0
 */
@Nullable
<T> T getParameterValue(@Nonnull Parameter<T> parameter);

/**
 * Return the input value bound to the named parameter.
 * (Note that OUT parameters are unbound.)
 *
 * @param name The name of the parameter
 * @return parameter value
 * @throws IllegalStateException if the parameter has not
 *         been bound
 * @throws IllegalArgumentException if the parameter of the
 *         specified name does not exist
```

```
 * @since 2.0
 */
@Nullable
Object getParameterValue(@Nonnull String name);

/**
 * Return the input value bound to the positional parameter.
 * (Note that OUT parameters are unbound.)
 *
 * @param position The parameter position
 * @return parameter value
 * @throws IllegalStateException if the parameter has not
 *          been bound
 * @throws IllegalArgumentException if the parameter with the
 *          specified position does not exist
 * @since 2.0
 */
@Nullable
Object getParameterValue(int position);

/**
 * Set the {@linkplain QueryFlushMode query flush mode} to be
 * used when the query is executed. This flush mode overrides
 * the {@linkplain EntityManager#getFlushMode flush mode type
 * of the entity manager}.
 *
 * @param flushMode The new flush mode
 * @return the same query instance
 *
 * @since 4.0
 */
@Nonnull
Query setQueryFlushMode(@Nonnull QueryFlushMode flushMode);

/**
 * Get the flush mode which will be in effect when the query
 * is executed. If a flush mode has not been set for this query
 * object, return {@link QueryFlushMode#DEFAULT}.
 *
 * @return flush mode
 * @since 4.0
 */
@Nonnull
QueryFlushMode getQueryFlushMode();

/**
 * Set the {@linkplain FlushModeType flush mode type} to be
 * used when the query is executed. This flush mode overrides
 * the {@linkplain EntityManager#getFlushMode flush mode type
 * of the entity manager}.
 *
 * @param flushMode The new flush mode
 * @return the same query instance
 *
 * @deprecated Use {@link #setQueryFlushMode(QueryFlushMode)}.
 */
@Deprecated(since = "4.0")
@Nonnull
Query setFlushMode(@Nonnull FlushModeType flushMode);

/**
 * Get the flush mode which will be in effect when the query
 * is executed. If a flush mode has not been set for this query
 * object, return the {@linkplain EntityManager#getFlushMode
 * current flush mode type of the entity manager}.
 *
 * @return flush mode
```

```
 * @since 2.0
 *
 * @deprecated Use {@link #getQueryFlushMode()}.
 */
@Deprecated(since = "4.0")
@Nonnull
FlushModeType getFlushMode();

/**
 * Set the {@linkplain LockModeType lock mode type} to use
 * when the query is executed.
 *
 * @param lockMode The new lock mode
 * @return the same query instance
 * @throws IllegalStateException if the query is not a Jakarta
 *         Persistence query language {@code SELECT} query or a
 *         {@link jakarta.persistence.criteria.CriteriaQuery}
 * @see #getLockMode
 * @since 2.0
 * @deprecated Use {@link TypedQuery#setLockMode}
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
Query setLockMode(@Nonnull LockModeType lockMode);

/**
 * The current {@linkplain LockModeType lock mode} for the
 * query or {@code null} if a lock mode has not been set.
 * <p>The lock mode affects every entity occurring as an
 * item in the SELECT clause, including entities occurring
 * as arguments to constructors. The effect on association
 * join tables, collection tables, and primary and secondary
 * tables of join fetched entities is determined by the
 * lock scope in effect. If no lock scope was explicitly
 * specified, the lock scope defaults to
 * {@link PessimisticLockScope#NORMAL NORMAL}.
 * <p>If the given lock mode is
 * {@link LockModeType#PESSIMISTIC_READ PESSIMISTIC_READ},
 * {@link LockModeType#PESSIMISTIC_WRITE PESSIMISTIC_WRITE},
 * or {@link LockModeType#PESSIMISTIC_FORCE_INCREMENT
 * PESSIMISTIC_FORCE_INCREMENT}, the lock also affects every
 * entity with an attribute reference occurring in the SELECT
 * clause, except when the attribute reference occurs as an
 * argument to an aggregate function.
 *
 * @return the current lock mode
 * @throws IllegalStateException if the query is not a Jakarta
 *         Persistence query language {@code SELECT} query or a
 *         {@link jakarta.persistence.criteria.CriteriaQuery}
 * @since 2.0
 * @deprecated Use {@link TypedQuery#getLockMode}
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nullable
LockModeType getLockMode();

/**
 * Set the {@linkplain CacheRetrieveMode cache retrieval mode}
 * in effect during query execution. This cache retrieval mode
 * overrides the {@linkplain EntityManager#getCacheRetrieveMode
 * cache retrieve mode of the entity manager}.
 *
 * @param cacheRetrieveMode The new cache retrieval mode
 * @return the same query instance
 * @since 3.2
 * @deprecated Use {@link TypedQuery#setCacheRetrieveMode}
 */
```

```java
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
Query setCacheRetrieveMode(@Nonnull CacheRetrieveMode cacheRetrieveMode);

/**
 * Set the {@linkplain CacheStoreMode cache storage mode} in
 * effect during query execution. This cache storage mode
 * overrides the {@linkplain EntityManager#getCacheStoreMode
 * cache storage mode of the entity manager}.
 *
 * @param cacheStoreMode The new cache storage mode
 * @return the same query instance
 * @since 3.2
 * @deprecated Use {@link TypedQuery#setCacheStoreMode}
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
Query setCacheStoreMode(@Nonnull CacheStoreMode cacheStoreMode);

/**
 * The {@linkplain CacheRetrieveMode cache retrieval mode} in
 * effect during query execution.
 *
 * @return the current cache retrieval mode set by calling
 *         {@link #setCacheRetrieveMode} or the cache retrieval
 *         mode of the persistence context if no cache retrieval
 *         mode has been explicitly specified for this query.
 * @since 3.2
 * @deprecated Use {@link TypedQuery#getCacheRetrieveMode}
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
CacheRetrieveMode getCacheRetrieveMode();

/**
 * The {@linkplain CacheStoreMode cache storage mode} in effect
 * during query execution.
 *
 * @return the current cache storage mode set by calling
 *         {@link #setCacheStoreMode} or the cache storage
 *         mode of the persistence context if no cache storage
 *         mode has been explicitly specified for this query.
 * @since 3.2
 * @deprecated Use {@link TypedQuery#getCacheStoreMode}
 */
@Deprecated(since = "4.0", forRemoval = true)
@Nonnull
CacheStoreMode getCacheStoreMode();

/**
 * Set the query timeout, in milliseconds. This is a hint,
 * and is an alternative to {@linkplain #setHint setting
 * the hint} {@code jakarta.persistence.query.timeout}.
 *
 * @param timeout the timeout, in milliseconds, or null to
 *                indicate no timeout
 * @return the same query instance
 * @since 3.2
 */
@Nonnull
Query setTimeout(@Nullable Integer timeout);

/**
 * Set the query timeout. This is a hint.
 *
 * @param timeout the timeout, or null to indicate no timeout
 * @return the same query instance
```

```
     * @since 4.0
     */
    @Nonnull
    Query setTimeout(@Nullable Timeout timeout);

    /**
     * The query timeout, in milliseconds, or null for no timeout.
     *
     * @since 3.2
     */
    @Nullable
    Integer getTimeout();

    /**
     * Return an object of the specified type to allow access to
     * a provider-specific API. If the provider implementation of
     * {@code Query} does not support the given type, the
     * {@link PersistenceException} is thrown.
     *
     * @param type The type of the object to be returned.
     *             This is usually either the underlying class
     *             implementing {@code Query} or an interface it
     *             implements.
     * @param <T> the type of the object to be returned
     * @return an instance of the specified class
     * @throws PersistenceException if the provider does not support
     *         the given type
     * @since 2.0
     */
    @Nonnull
    <T> T unwrap(@Nonnull Class<T> type);
}
```

B.10. Statement

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.Type;

import java.util.Calendar;
import java.util.Date;
import java.util.Set;

/**
 * Interface used to control the execution of executable statements.
 * In the Jakarta Persistence query language only {@code UPDATE} and
 * {@code DELETE} statements are executable statements. On the other
 * hand, a native SQL statement is considered executable if it returns
 * a row count instead of a result set.
 *
 * @apiNote Just like any other data access API, including JDBC
 * itself, native SQL statement strings and Jakarta Persistence
 * query language strings executed via this API must never be
 * composed by concatenating user input or other untrusted data.
 * User input should be properly validated and passed as a
 * {@linkplain #setParameter(String, Object) parameter} to a
 * parameterized statement string. In particular, native SQL
 * statement strings executed via this API are typically executed
 * verbatim with no additional validation or sanitization beyond
 * that performed by the JDBC driver.
 *
 * @see EntityHandler#createStatement(String)
 * @see EntityHandler#createStatement(jakarta.persistence.criteria.CriteriaStatement)
 * @see EntityHandler#createStatement(StatementReference)
 * @see EntityHandler#createNativeStatement(String)
```

```
 * @see EntityHandler#createNamedStatement(String)
 * @see StatementOrTypedQuery#asStatement()
 *
 * @since 4.0
 */
public interface Statement extends Query {
    /**
     * Execute an {@code UPDATE} or {@code DELETE} statement or a
     * native SQL statement that returns a row count.
     * <p>
     * After execution of a bulk update or delete operation, the
     * persistence provider is not required to resynchronize state
     * held in memory with the effects of the operation on data
     * held in the database. However, when this method is called
     * within a transaction, the persistence context is joined to
     * the transaction, and {@link FlushModeType#AUTO} is in effect,
     * the persistence provider must ensure that every modification
     * to the state of every entity associated with the persistence
     * context which could possibly alter the effects of the bulk
     * update or delete operation is visible to the processing of
     * the operation.
     *
     * @return the number of entities updated or deleted, or the
     *         row count of the native SQL statement
     * @throws IllegalStateException if called for a Jakarta
     *         Persistence query language SELECT statement or for
     *         a criteria query
     * @throws TransactionRequiredException if there is
     *         no transaction or the persistence context has not
     *         been joined to the transaction
     * @throws QueryTimeoutException if the statement execution
     *         exceeds the query timeout value set and only
     *         the statement is rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     * @throws PersistenceException if the flush fails
     * @throws OptimisticLockException if an optimistic locking
     *         conflict is detected during the flush
     */
    int execute();

    /**
     * Set a query property or hint. Properties defined by this
     * specification must be observed by the persistence provider.
     * Vendor-specific hints that are not recognized by a provider
     * must be silently ignored. Portable applications should not
     * rely on the standard timeout hint; depending on the database
     * in use and the locking mechanisms used by the provider,
     * this hint may or may not be observed.
     *
     * @param hintName The name of the property or hint
     * @param value The value for the property or hint
     * @return the same statement instance
     * @throws IllegalArgumentException if the given property or
     *         hint name is recognized by the provider, but the
     *         second argument is not a legal value for the given
     *         property or hint
     *
     * @apiNote The use of named hints lacks type safety compared
     *          to the use of {@linkplain Option options}.
     */
    @Override
    @Nonnull
    Statement setHint(@Nonnull String hintName, @Nullable Object value);

    /**
```

```
 * Set the query timeout, in milliseconds. This is a hint,
 * and is an alternative to {@linkplain #setHint setting
 * the hint} {@code jakarta.persistence.query.timeout}.
 *
 * @param timeout The timeout, in milliseconds, or null to
 *                indicate no timeout
 * @return the same statement instance
 * @since 3.2
 */
@Override
@Nonnull
Statement setTimeout(@Nullable Integer timeout);

/**
 * Set the query timeout. This is a hint.
 *
 * @param timeout the timeout, or null to indicate no timeout
 * @return the same statement instance
 * @since 4.0
 */
@Override
@Nonnull
Statement setTimeout(@Nullable Timeout timeout);

/**
 * Set the {@linkplain QueryFlushMode query flush mode} to be
 * used when the query is executed. This flush mode overrides
 * the {@linkplain EntityManager#getFlushMode flush mode type
 * of the entity manager}.
 *
 * @param flushMode The new flush mode
 * @return the same statement instance
 *
 * @since 4.0
 */
@Override
@Nonnull
Statement setQueryFlushMode(@Nonnull QueryFlushMode flushMode);

/**
 * Set the {@linkplain FlushModeType flush mode type} to be
 * used when the query is executed. This flush mode overrides
 * the {@linkplain EntityManager#getFlushMode flush mode type
 * of the entity manager}.
 *
 * @param flushMode The new flush mode
 * @return the same statement instance
 *
 * @deprecated Use {@link #setQueryFlushMode(QueryFlushMode)}.
 */
@Override @Deprecated
@Nonnull
Statement setFlushMode(@Nonnull FlushModeType flushMode);

/**
 * Bind an argument to a parameter of this query respresented as
 * a {@link Parameter} object.
 *
 * @param parameter The parameter object
 * @param value The argument to the parameter
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter
 *         does not correspond to a parameter of the
 *         query
 * @since 2.0
 */
@Override
```

```
@Nonnull
<T> Statement setParameter(@Nonnull Parameter<T> parameter,
                           @Nullable T value);

/**
 * Bind an instance of {@link Calendar} to a {@link Parameter} object.
 *
 * @param param The parameter object
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter does not
 *         correspond to a parameter of the query
 * @since 2.0
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2") @Override
@Nonnull
Statement setParameter(@Nonnull Parameter<Calendar> param,
                       @Nullable Calendar value,
                       @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a {@link Parameter} object.
 *
 * @param param The parameter object
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter does not
 *         correspond to a parameter of the query
 * @since 2.0
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2") @Override
@Nonnull
Statement setParameter(@Nonnull Parameter<Date> param,
                       @Nullable Date value,
                       @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 */
@Override
@Nonnull
Statement setParameter(@Nonnull String name, @Nullable Object value);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the argument might be {@code null}, especially in the case
 * of a native query.
 * {@snippet :
 * em.createNativeStatement("update books set pub_date = :date where isbn = :ISBN")
 *     .setParameter("date", optionalPublicationDate, LocalDate.class)
 *     .setParameter("ISBN", isbn)
 *     .execute();
 * }
```

```
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param type A class object representing the parameter type
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> Statement setParameter(@Nonnull String name, @Nullable P value,
                           @Nonnull Class<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the binding is affected by an attribute converter.
 * {@snippet :
 * em.createNativeStatement("update books set pub_date = :date where isbn = :ISBN")
 *     .setParameter("date", optionalPublicationDate, Book_.publicationDate.getType())
 *     .setParameter("ISBN", isbn, Book_.isbn.getType())
 *     .execute();
 * }
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param type The {@link Type} of the parameter
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> Statement setParameter(@Nonnull String name, @Nullable P value,
                           @Nonnull Type<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying an {@linkplain AttributeConverter attribute
 * converter} to use.
 * {@snippet :
 * em.createNativeStatement("update books set pub_date = :date where isbn = :ISBN")
 *     .setParameter("date", publicationDate)
 *     .setParameter("ISBN", isbn, IsbnConverter.class)
 *     .execute();
 * }
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param converter The class of the attribute converter
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> Statement setConvertedParameter(@Nonnull String name, @Nullable P value,
                                    @Nonnull Class<? extends AttributeConverter<P,?>> converter);

/**
 * Bind an instance of {@link Calendar} to a named parameter.
```

```
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the value argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Override
@Deprecated(since = "3.2")
@Nonnull
Statement setParameter(@Nonnull String name, @Nullable Calendar value,
                       @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same statement instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the value argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Override
@Deprecated(since = "3.2")
@Nonnull
Statement setParameter(@Nonnull String name, @Nullable Date value,
                       @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a positional parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @return the same statement instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 */
@Override
@Nonnull
Statement setParameter(int position, @Nullable Object value);

/**
 * Bind an argument value to a positional parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the argument might be {@code null}, especially in the case
 * of a native SQL query.
 * {@snippet :
 * em.createNativeStatement("update books set pub_date = ?1 where isbn = ?2")
 *     .setParameter(1, optionalPublicationDate, LocalDate.class)
 *     .setParameter(2, isbn)
 *     .execute();
 * }
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param type A class object representing the parameter type
 * @return the same statement instance
 * @throws IllegalArgumentException if the given position does
```

```
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> Statement setParameter(int position, @Nullable P value,
                           @Nonnull Class<P> type);

/**
 * Bind an argument value to a positional parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the binding is affected by an attribute converter.
 * {@snippet :
 * em.createNativeStatement("update books set pub_date = ?1 where isbn = ?2")
 *     .setParameter(1, optionalPublicationDate, Book_.publicationDate.getType())
 *     .setParameter(2, isbn, Book_.isbn.getType())
 *     .execute();
 * }
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param type The {@link Type} of the parameter
 * @return the same statement instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> Statement setParameter(int position, @Nullable P value,
                           @Nonnull Type<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying an {@linkplain AttributeConverter attribute
 * converter} to use.
 * {@snippet :
 * em.createNativeStatement("update books set pub_date = ?1 where isbn = ?2")
 *     .setParameter(1, publicationDate)
 *     .setParameter(2, isbn, IsbnConverter.class)
 *     .execute();
 * }
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param converter The class of the attribute converter
 * @return the same statement instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> Statement setConvertedParameter(int position, @Nullable P value,
                                    @Nonnull Class<? extends AttributeConverter<P,?>> converter);

/**
 * Bind an instance of {@link Calendar} to a positional
 * parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same statement instance
```

```
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2") @Override
@Nonnull
Statement setParameter(int position, @Nullable Calendar value,
                       @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a positional
 * parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same statement instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2") @Override
@Nonnull
Statement setParameter(int position, @Nullable Date value,
                       @Nonnull TemporalType temporalType);

/**
 * Specify an {@linkplain Option option} influencing execution
 * of this statement, overwriting any existing option of the
 * same type.
 *
 * @param option the option
 * @return the same statement instance
 * @since 4.0
 */
@Nonnull
Statement addOption(@Nonnull Option option);

/**
 * Get the {@linkplain Option options} influencing execution of
 * this statement. The returned set includes options set via
 * {@link #addOption}, along with options specified via
 * {@link #setTimeout} or {@link #setQueryFlushMode}. Mutation of
 * the returned set does not affect the options of the statement.
 *
 * @return the options for this statement
 * @since 4.0
 */
@Nonnull
Set<Option> getOptions();

/**
 * An option influencing execution of a statement. This provides
 * a more type safe alternative to the use of {@linkplain #setHint
 * hints}.
 *
 * <p>This interface may be implemented by custom provider-specific
 * options which extend the options defined by the specification.
 *
 * @see QueryFlushMode
 * @see Timeout
 *
 * @since 4.0
```

```
     */
    interface Option {
    }
}
```

B.11. TypedQuery

```
//     Linda DeMichiel - 2.0

package jakarta.persistence;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.Type;

import java.util.List;
import java.util.Date;
import java.util.Calendar;
import java.util.Set;
import java.util.stream.Stream;

/**
 * Interface used to control the execution of typed queries. In
 * the Jakarta Persistence query language, only {@code SELECT}
 * queries are typed queries, since only a {@code SELECT} query
 * can return a result. A {@code DELETE} or {@code UPDATE} query
 * is not a typed query and is always represented by an untyped
 * instance of {@link Query}, usually, by a {@link Statement}.
 * On the other hand, a native SQL query is considered a typed
 * query if it returns a result set.
 *
 * @apiNote Just like any other data access API, including JDBC
 * itself, native SQL query strings and Jakarta Persistence query
 * language strings executed via this API must never be composed
 * by concatenating user input or other untrusted data. User
 * input should be properly validated and passed as a
 * {@linkplain #setParameter(String, Object) parameter} to a
 * parameterized query string. In particular, native SQL query
 * strings executed via this API are typically executed verbatim
 * with no additional validation or sanitization beyond that
 * performed by the JDBC driver.
 *
 * @param <X> query result type
 *
 * @see EntityHandler#createQuery(String,Class)
 * @see EntityHandler#createQuery(String,EntityGraph)
 * @see EntityHandler#createQuery(jakarta.persistence.criteria.CriteriaSelect)
 * @see EntityHandler#createQuery(TypedQueryReference)
 * @see EntityHandler#createNativeQuery(String,Class)
 * @see EntityHandler#createNativeQuery(String,jakarta.persistence.sql.ResultSetMapping)
 * @see EntityHandler#createNamedQuery(String,Class)
 * @see StatementOrTypedQuery#ofType(Class)
 * @see StatementOrTypedQuery#withEntityGraph(EntityGraph)
 * @see StatementOrTypedQuery#withResultSetMapping(jakarta.persistence.sql.ResultSetMapping)
 *
 * @since 2.0
 */
public interface TypedQuery<X> extends Query {

    /**
     * <p>Determine the maximum number of results that could in
     * principle be returned by the query if no
     * {@linkplain #getFirstResult() offset} or
     * {@linkplain #getMaxResults() limit} were applied.</p>
     *
```

```
 * <p>The {@code getResultCount} method should not cause query
 * results to be fetched from the database.</p>
 *
 * @return the maximum number of results that could in principle
 *         be returned by the query if no offset or limit were
 *         applied
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language UPDATE or DELETE statement
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @since 4.0
 */
long getResultCount();

/**
 * Execute the query and return the query results as a typed
 * {@link List List&lt;X&gt;}. If necessary, first synchronize
 * changes with the database by flushing the persistence context.
 *
 * @return a list of the results, each of type {@link X}, or an
 *         empty list if there are no results
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language UPDATE or DELETE statement
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws TransactionRequiredException if a lock mode other than
 *         {@code NONE} has been set and there is no transaction
 *         or the persistence context has not been joined to the
 *         transaction
 * @throws PessimisticLockException if pessimistic locking
 *         fails and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking
 *         fails and only the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 */
@Override @SuppressWarnings("removal")
@Nonnull
List<X> getResultList();

/**
 * Execute the query, returning a single typed result.
 * If necessary, first synchronize changes with the database by
 * flushing the persistence context.
 *
 * @return the result, of type {@link X}
 * @throws NoResultException if there is no result
 * @throws NonUniqueResultException if more than one result
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language UPDATE or DELETE statement
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws TransactionRequiredException if a lock mode other than
 *         {@code NONE} has been set and there is no transaction
 *         or the persistence context has not been joined to the
 *         transaction
 * @throws PessimisticLockException if pessimistic locking
```

```
 *         fails and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking
 *         fails and only the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 */
@Override @SuppressWarnings("removal")
X getSingleResult();

/**
 * Execute the query and return the query results as a typed
 * {@link Stream Stream&lt;X&gt;}. If necessary, first synchronize
 * changes with the database by flushing the persistence context.
 * <p>By default, this method delegates to {@link List#stream()
 * getResultList().stream()}. The persistence provider may choose
 * to override this method to provide additional capabilities.
 *
 * @return a stream of the results, each of type {@link X}, or an
 *         empty stream if there are no results
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language UPDATE or DELETE statement
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws TransactionRequiredException if a lock mode other than
 *         {@code NONE} has been set and there is no transaction
 *         or the persistence context has not been joined to the
 *         transaction
 * @throws PessimisticLockException if pessimistic locking
 *         fails and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking
 *         fails and only the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 * @see Stream
 * @see #getResultList()
 * @since 2.2
 */
@Override @SuppressWarnings("removal")
@Nonnull
default Stream<X> getResultStream() {
    return getResultList().stream();
}

/**
 * Execute the query, returning a single typed result, or
 * {@code null} if the query has no results. If necessary, first
 * synchronize changes with the database by flushing the persistence
 * context.
 *
 * @return the result, of type {@link X}, or {@code null} if there
 *         is no result
 * @throws NonUniqueResultException if more than one result
 * @throws IllegalStateException if called for a Jakarta
 *         Persistence query language UPDATE or DELETE statement
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws TransactionRequiredException if a lock mode other than
```

```
 *         {@code NONE} has been set and there is no transaction
 *         or the persistence context has not been joined to the
 *         transaction
 * @throws PessimisticLockException if pessimistic locking
 *         fails and the transaction is rolled back
 * @throws LockTimeoutException if pessimistic locking
 *         fails and only the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @throws PersistenceException if the flush fails
 * @throws OptimisticLockException if an optimistic locking
 *         conflict is detected during the flush
 *
 * @since 3.2
 */
@Override @SuppressWarnings("removal")
@Nullable
X getSingleResultOrNull();

/**
 * Set the maximum number of results returned to the client.
 * If the query has more results than the given maximum,
 * results are excluded from the list returned by
 * {@link #getResultList()}, so that only the given number
 * of results is returned. If the query returns results
 * with a well-defined order, the excluded results must be
 * those which would otherwise occur later in the list.
 *
 * @param maxResult The maximum number of results
 * @return the same query instance
 * @throws IllegalArgumentException if the argument is negative
 */
@Override @SuppressWarnings("removal")
@Nonnull
TypedQuery<X> setMaxResults(int maxResult);

/**
 * Set the position of the first query result returned to
 * the client. The given number of results is excluded
 * from the list returned by {@link #getResultList()}.
 * If the query returns results with a well-defined order,
 * the excluded results must be those which would otherwise
 * occur earlier in the list.
 *
 * @param startPosition The position of the first result,
 *                      numbered from {@code 0}
 * @return the same query instance
 * @throws IllegalArgumentException if the argument is negative
 */
@Override @SuppressWarnings("removal")
@Nonnull
TypedQuery<X> setFirstResult(int startPosition);

/**
 * The maximum number of results returned to the client,
 * as specified by {@link #setMaxResults}, or
 * {@value Integer#MAX_VALUE} if {@link #setMaxResults}
 * was not called.
 *
 * @return the maximum number of results
 * @since 2.0
 */
@Override @SuppressWarnings("removal")
int getMaxResults();

/**
```

```
 * The position of the first result returned to the client,
 * as specified by {@link #setFirstResult}, or {@code 0} if
 * {@link #setFirstResult} was not called.
 *
 * @return the position of the first result
 * @since 2.0
 */
@Override @SuppressWarnings("removal")
int getFirstResult();

/**
 * Set a query property or hint. Properties defined by this
 * specification must be observed by the persistence provider.
 * Vendor-specific hints that are not recognized by a provider
 * must be silently ignored. Portable applications should not
 * rely on the standard timeout hint; depending on the database
 * in use and the locking mechanisms used by the provider,
 * this hint may or may not be observed.
 *
 * @param hintName The name of the property or hint
 * @param value The value for the property or hint
 * @return the same query instance
 * @throws IllegalArgumentException if the given property or
 *         hint name is recognized by the provider, but the
 *         second argument is not a legal value for the given
 *         property or hint
 *
 * @apiNote The use of named hints lacks type safety compared
 *          to the use of {@linkplain Option options}.
 */
@Override
@Nonnull
TypedQuery<X> setHint(@Nonnull String hintName, @Nullable Object value);

/**
 * Bind an argument to a parameter of this query respresented as
 * a {@link Parameter} object.
 *
 * @param parameter The parameter object
 * @param value The argument to the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter
 *         does not correspond to a parameter of the
 *         query
 */
@Override
@Nonnull
<T> TypedQuery<X> setParameter(@Nonnull Parameter<T> parameter,
                               @Nullable T value);

/**
 * Bind an instance of {@link Calendar} to a {@link Parameter} object.
 *
 * @param param The parameter object
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter does not
 *         correspond to a parameter of the query
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2") @Override
@Nonnull
TypedQuery<X> setParameter(@Nonnull Parameter<Calendar> param,
                           @Nullable Calendar value,
                           @Nonnull TemporalType temporalType);
```

```java
/**
 * Bind an instance of {@link Date} to a {@link Parameter} object.
 *
 * @param param The parameter object
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter does not
 *         correspond to a parameter of the query
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2") @Override
@Nonnull
TypedQuery<X> setParameter(@Nonnull Parameter<Date> param,
                           @Nullable Date value,
                           @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 */
@Override
@Nonnull
TypedQuery<X> setParameter(@Nonnull String name, @Nullable Object value);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the argument might be {@code null}.
 * {@snippet :
 * var books =
 *     em.createNativeQuery("select * from books where :limit is null or pub_date > :limit",
 *                          Book.class)
 *         .setParameter("limit", optionalDateLimit, LocalDate.class)
 *         .getResultList();
 * }
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param type A class object representing the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> TypedQuery<X> setParameter(@Nonnull String name, @Nullable P value,
                               @Nonnull Class<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the binding is affected by an attribute converter.
 * {@snippet :
 * var amount = MonetaryAmount.of(priceLimit, currency);
 * var affordableBooks =
 *     em.createQuery("from Book where price < :amount", Book.class)
```

```
 *         .setParameter("amount", amount, Book_.price.getType())
 *         .getResultList();
 * }
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param type The {@link Type} of the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> TypedQuery<X> setParameter(@Nonnull String name, @Nullable P value,
                               @Nonnull Type<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying an {@linkplain AttributeConverter attribute
 * converter} to use.
 * {@snippet :
 * var amount = MonetaryAmount.of(priceLimit, currency);
 * var affordableBooks =
 *     em.createQuery("from Book where price < :amount", Book.class)
 *         .setConvertedParameter("amount", amount,
 *                 MonetaryAmountConverter.class)
 *         .getResultList();
 *}
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param converter The class of the attribute converter
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> TypedQuery<X> setConvertedParameter(@Nonnull String name, @Nullable P value,
                                        @Nonnull Class<? extends AttributeConverter<P, ?>> converter);

/**
 * Bind an instance of {@link Calendar} to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the value argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Override
@Deprecated(since = "3.2")
@Nonnull
TypedQuery<X> setParameter(@Nonnull String name, @Nullable Calendar value,
                           @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a named parameter.
 *
```

```
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the value argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Override
@Deprecated(since = "3.2")
@Nonnull
TypedQuery<X> setParameter(@Nonnull String name, @Nullable Date value,
                           @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a positional parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 */
@Override
@Nonnull
TypedQuery<X> setParameter(int position, @Nullable Object value);

/**
 * Bind an argument value to a positional parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the argument might be {@code null}.
 * {@snippet :
 * var books =
 *     em.createNativeQuery("select * from books where ?1 is null or pub_date > ?1",
 *                                Book.class)
 *         .setParameter(1, optionalDateLimit, LocalDate.class)
 *         .getResultList();
 * }
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param type A class object representing the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> TypedQuery<X> setParameter(int position, @Nullable P value,
                               @Nonnull Class<P> type);

/**
 * Bind an argument value to a positional parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the binding is affected by an attribute converter.
 * {@snippet :
 * var amount = MonetaryAmount.of(priceLimit, currency);
 * var affordableBooks =
 *     em.createQuery("from Book where price < ?1", Book.class)
 *         .setParameter(1, amount, Book_.price.getType())
 *         .getResultList();
 * }
```

```
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param type The {@link Type} of the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> TypedQuery<X> setParameter(int position, @Nullable P value,
                               @Nonnull Type<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying an {@linkplain AttributeConverter attribute
 * converter} to use.
 * {@snippet :
 * var amount = MonetaryAmount.of(priceLimit, currency);
 * var affordableBooks =
 *     em.createQuery("from Book where price < ?1", Book.class)
 *         .setConvertedParameter(1, amount,
 *                 MonetaryAmountConverter.class)
 *         .getResultList();
 *}
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param converter The class of the attribute converter
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> TypedQuery<X> setConvertedParameter(int position, @Nullable P value,
                                        @Nonnull Class<? extends AttributeConverter<P, ?>> converter);

/**
 * Bind an instance of {@link Calendar} to a positional
 * parameter.
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Override
@Deprecated(since = "3.2")
@Nonnull
TypedQuery<X> setParameter(int position, @Nullable Calendar value,
                           @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a positional
 * parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
```

```
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Override
@Deprecated(since = "3.2")
@Nonnull
TypedQuery<X> setParameter(int position, @Nullable Date value,
                           @Nonnull TemporalType temporalType);

/**
 * Set the {@linkplain QueryFlushMode query flush mode} to be
 * used when the query is executed. This flush mode overrides
 * the {@linkplain EntityManager#getFlushMode flush mode type
 * of the entity manager}.
 *
 * @param flushMode The new flush mode
 * @return the same query instance
 *
 * @since 4.0
 */
@Override
@Nonnull
TypedQuery<X> setQueryFlushMode(@Nonnull QueryFlushMode flushMode);

/**
  * Set the {@linkplain FlushModeType flush mode type} to be
  * used when the query is executed. This flush mode overrides
  * the {@linkplain EntityManager#getFlushMode flush mode type
  * of the entity manager}.
  *
  * @param flushMode The new flush mode
  * @return the same query instance
 *
 * @deprecated Use {@link #setQueryFlushMode(QueryFlushMode)}.
  */
 @Override
 @Deprecated(since = "4.0")
 @Nonnull
 TypedQuery<X> setFlushMode(@Nonnull FlushModeType flushMode);

 /**
  * Set the {@linkplain LockModeType lock mode type} to use
  * when the query is executed.
  *
  * @param lockMode The new lock mode
  * @return the same query instance
  * @throws IllegalStateException if the query is not a Jakarta
  *         Persistence query language {@code SELECT} query or a
  *         {@link jakarta.persistence.criteria.CriteriaQuery}
  * @see #getLockMode
  * @since 2.0
  */
 @Override @SuppressWarnings("removal")
 @Nonnull
 TypedQuery<X> setLockMode(@Nonnull LockModeType lockMode);

/**
 * Set the {@linkplain PessimisticLockScope pessimistic lock scope}
 * to use when the query is executed if a pessimistic lock mode
 * is specified via {@link #setLockMode}.
 * If the query is executed without a pessimistic lock mode,
 * the pessimistic lock scope has no effect.
```

```
 *
 * @since 4.0
 * @param lockScope The scope of the pessimistic lock
 * @return the same query instance
 * @throws IllegalStateException if the query is not a Jakarta
 *         Persistence query language {@code SELECT} query or a
 *         {@link jakarta.persistence.criteria.CriteriaQuery}
 */
@Nonnull
TypedQuery<X> setLockScope(@Nonnull PessimisticLockScope lockScope);

/**
 * The current {@linkplain LockModeType lock mode} for the
 * query or {@code null} if a lock mode has not been set.
 * <p>The lock mode affects every entity occurring as an
 * item in the SELECT clause, including entities occurring
 * as arguments to constructors. The effect on association
 * join tables, collection tables, and primary and secondary
 * tables of join fetched entities is determined by the
 * specified {@linkplain #getLockScope lock scope}. If no
 * lock scope was explicitly specified, the lock scope
 * defaults to {@link PessimisticLockScope#NORMAL NORMAL}.
 * <p>If the given lock mode is
 * {@link LockModeType#PESSIMISTIC_READ PESSIMISTIC_READ},
 * {@link LockModeType#PESSIMISTIC_WRITE PESSIMISTIC_WRITE},
 * or {@link LockModeType#PESSIMISTIC_FORCE_INCREMENT
 * PESSIMISTIC_FORCE_INCREMENT}, the lock also affects every
 * entity with an attribute reference occurring in the SELECT
 * clause, except when the attribute reference occurs as an
 * argument to an aggregate function.
 *
 * @return the current lock mode
 * @throws IllegalStateException if the query is not a Jakarta
 *         Persistence query language {@code SELECT} query or a
 *         {@link jakarta.persistence.criteria.CriteriaQuery}
 * @see #getLockScope
 * @since 2.0
 */
@Override @SuppressWarnings("removal")
@Nullable
LockModeType getLockMode();

/**
 * The current {@linkplain PessimisticLockScope pessimistic
 * lock scope} for the query or {@code null} if a scope has
 * not been set.
 * <p>The lock scope determines the effect of
 * {@linkplain #getLockMode locking} on association join
 * tables, collection tables, and primary and secondary tables
 * of join fetched entities. If no lock scope was explicitly
 * specified, locking behaves as if the lock scope were set
 * to {@link PessimisticLockScope#NORMAL NORMAL}.
 * <p>The pessimistic lock scope has no effect if the lock
 * mode is {@code null} or {@link LockModeType#NONE NONE}.
 *
 * @return the current pessimistic lock scope
 * @throws IllegalStateException if the query is not a Jakarta
 *         Persistence query language {@code SELECT} query or a
 *         {@link jakarta.persistence.criteria.CriteriaQuery}
 * @since 4.0
 */
@Nullable
PessimisticLockScope getLockScope();

/**
 * Set the {@linkplain CacheRetrieveMode cache retrieval mode}
 * in effect during query execution. This cache retrieval mode
```

```
 * overrides the {@linkplain EntityManager#getCacheRetrieveMode
 * cache retrieve mode of the entity manager}.
 *
 * @param cacheRetrieveMode The new cache retrieval mode
 * @return the same query instance
 * @since 3.2
 */
@Override @SuppressWarnings("removal")
@Nonnull
TypedQuery<X> setCacheRetrieveMode(@Nonnull CacheRetrieveMode cacheRetrieveMode);

/**
 * Set the {@linkplain CacheStoreMode cache storage mode} in
 * effect during query execution. This cache storage mode
 * overrides the {@linkplain EntityManager#getCacheStoreMode
 * cache storage mode of the entity manager}.
 *
 * @param cacheStoreMode The new cache storage mode
 * @return the same query instance
 * @since 3.2
 */
@Override @SuppressWarnings("removal")
@Nonnull
TypedQuery<X> setCacheStoreMode(@Nonnull CacheStoreMode cacheStoreMode);

/**
 * The {@linkplain CacheRetrieveMode cache retrieval mode} in
 * effect during query execution.
 *
 * @return The current cache retrieval mode set by calling
 *         {@link #setCacheRetrieveMode} or the cache retrieval
 *         mode of the persistence context if no cache retrieval
 *         mode has been explicitly specified for this query.
 * @since 3.2
 */
@Override @SuppressWarnings("removal")
@Nonnull
CacheRetrieveMode getCacheRetrieveMode();

/**
 * The {@linkplain CacheStoreMode cache storage mode} in effect
 * during query execution.
 *
 * @return The current cache storage mode set by calling
 *         {@link #setCacheStoreMode} or the cache storage
 *         mode of the persistence context if no cache storage
 *         mode has been explicitly specified for this query.
 * @since 3.2
 */
@Override @SuppressWarnings("removal")
@Nonnull
CacheStoreMode getCacheStoreMode();

/**
 * Set the query timeout, in milliseconds. This is a hint,
 * and is an alternative to {@linkplain #setHint setting
 * the hint} {@code jakarta.persistence.query.timeout}.
 *
 * @param timeout the timeout, in milliseconds, or null to
 *                indicate no timeout
 * @return the same query instance
 * @since 3.2
 */
@Override
@Nonnull
TypedQuery<X> setTimeout(@Nullable Integer timeout);
```

```
/**
 * Set the query timeout. This is a hint.
 *
 * @param timeout the timeout, or null to indicate no timeout
 * @return the same query instance
 * @since 4.0
 */
@Override
@Nonnull
TypedQuery<X> setTimeout(@Nullable Timeout timeout);

/**
 * @deprecated
 * This operation should never be called on a {@code TypedQuery}.
 * Any {@code DELETE} or {@code UPDATE} query should be represented
 * by an untyped instance of {@link Query} or, preferably, by an
 * instance of {@link Statement}.
 */
@Deprecated(since = "4.0", forRemoval = true)
@Override @SuppressWarnings("removal")
int executeUpdate();

/**
 * Specify an {@linkplain Option option} influencing execution
 * of this typed query, overwriting any existing option of the
 * same type.
 *
 * @param option the option
 * @return the same query instance
 * @since 4.0
 */
@Nonnull
TypedQuery<X> addOption(@Nonnull Option option);

/**
 * Get the {@linkplain Option options} influencing execution of
 * this typed query. The returned set includes options set via
 * {@link #addOption} along with options specified via
 * {@link #setTimeout}, {@link #setQueryFlushMode},
 * {@link #setCacheRetrieveMode}, {@link #setCacheStoreMode},
 * {@link #setLockMode}, and {@link #setLockScope}. Mutation
 * of the returned set does not affect the options of the query.
 *
 * @return the options for this query
 * @since 4.0
 */
@Nonnull
Set<Option> getOptions();

/**
 * An option influencing execution of a typed query. This provides
 * a more type safe alternative to the use of {@linkplain #setHint
 * hints}.
 *
 * <p>This interface may be implemented by custom provider-specific
 * options which extend the options defined by the specification.
 *
 * @see CacheRetrieveMode
 * @see CacheStoreMode
 * @see LockModeType
 * @see PessimisticLockScope
 * @see QueryFlushMode
 * @see Timeout
 *
 * @since 4.0
 */
interface Option {
```

```
    }
}
```

B.12. StoredProcedureQuery

```
package jakarta.persistence;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.Type;
import jakarta.persistence.sql.ResultSetMapping;

import java.util.Calendar;
import java.util.Date;
import java.util.List;
import java.util.Set;

/**
 * Interface used to control execution of a stored procedure query.
 * <p>
 * Before a stored procedure can be executed, each of its parameters
 * must be explicitly registered by specifying:
 * <ul>
 * <li>the type of the parameter,
 * <li>its position or name, and
 * <li>whether its mode is {@link ParameterMode#IN IN},
 *     {@link ParameterMode#OUT OUT}, {@link ParameterMode#INOUT INOUT},
 *     or {@link ParameterMode#REF_CURSOR REF_CURSOR}.
 * </ul>
 * <p>
 * Each parameter is registered by calling a registration method
 * such as {@link #registerParameter(int, Class, ParameterMode)}
 * or {@link #registerParameter(String, Class, ParameterMode)},
 * depending on whether it is a positional or named parameter.
 * <p>
 * Stored procedure query execution may be controlled in accordance with
 * the following:
 * <ul>
 * <li>The {@link #setParameter} methods are used to set the values of
 * all required {@code IN} and {@code INOUT} parameters. It is not
 * required to set the values of stored procedure parameters for which
 * default values have been defined by the stored procedure.</li>
 * <li> When {@link #getResultList} and {@link #getSingleResult} are
 * called on a {@code StoredProcedureQuery} object, the provider calls
 * {@link #execute} on an unexecuted stored procedure query before
 * processing {@code getResultList} or {@code getSingleResult}.</li>
 * <li> When {@link #executeUpdate} is called on a
 * {@code StoredProcedureQuery} object, the provider will call
 * {@link #execute} on an unexecuted stored procedure query, followed
 * by {@link #getUpdateCount}. The results of {@code executeUpdate} will
 * be those of {@code getUpdateCount}.</li>
 * <li> The {@link #execute} method supports both the simple case where
 * scalar results are passed back only via {@code INOUT} and {@code OUT}
 * parameters as well as the most general case (multiple result sets
 * and/or update counts, possibly also in combination with output
 * parameter values).</li>
 * <li> The {@code execute} method returns true if the first result is
 * a result set, and false if it is an update count or there are no
 * results other than through {@code INOUT} and {@code OUT} parameters,
 * if any.</li>
 * <li> If the {@code execute} method returns true, the pending result
 * set can be obtained by calling {@link #getResultList} or
 * {@link #getSingleResult}.</li>
 * <li> The {@link #hasMoreResults} method can then be used to test for
 * further results.</li>
```

```
 * <li> If {@code execute} or {@code hasMoreResults} returns false, the
 * {@link #getUpdateCount} method can be called to obtain the pending
 * result if it is an update count. The {@code getUpdateCount} method
 * will return either the update count (zero or greater) or -1 if there
 * is no update count (i.e., either the next result is a result set or
 * there is no next update count).</li>
 * <li> For portability, results that correspond to JDBC result sets
 * and update counts need to be processed before the values of any
 * {@code INOUT} or {@code OUT} parameters are extracted.</li>
 * <li> After results returned through {@link #getResultList} and
 * {@link #getUpdateCount} have been exhausted, results returned through
 * {@code INOUT} and {@code OUT} parameters can be retrieved.</li>
 * <li> The {@link #getOutputParameterValue} methods are used to
 * retrieve the values passed back from the procedure through
 * {@code INOUT} and {@code OUT} parameters.</li>
 * <li> When using {@code REF_CURSOR} parameters for result sets the
 * update counts should be exhausted before calling {@link #getResultList}
 * to retrieve the result set. Alternatively, the {@code REF_CURSOR}
 * result set can be retrieved through {@link #getOutputParameterValue}.
 * Result set mappings are applied to results corresponding to
 * {@code REF_CURSOR} parameters in the order the {@code REF_CURSOR}
 * parameters were registered with the query.</li>
 * <li> In the simplest case, where results are returned only via
 * {@code INOUT} and {@code OUT} parameters, {@code execute} can be
 * followed immediately by calls to {@link #getOutputParameterValue}.
 * </li>
 * </ul>
 *
 * @see Query
 * @see Parameter
 *
 * @since 2.1
 */
public interface StoredProcedureQuery extends Query, AutoCloseable {

    /**
     * Set a query property or hint. Properties defined by this
     * specification must be observed by the persistence provider.
     * Vendor-specific hints that are not recognized by a provider
     * must be silently ignored. Portable applications should not
     * rely on the standard timeout hint; depending on the database
     * in use, this hint may or may not be observed.
     *
     * @param hintName The name of the property or hint
     * @param value The value for the property or hint
     * @return the same query instance
     * @throws IllegalArgumentException if the given property or
     *         hint name is recognized by the provider, but the
     *         second argument is not a legal value for the given
     *         property or hint
     *
     * @apiNote The use of named hints lacks type safety compared
     *          to the use of {@linkplain Option options}.
     */
    @Override
    @Nonnull
    StoredProcedureQuery setHint(@Nonnull String hintName, @Nullable Object value);

    /**
     * Bind an argument to a parameter of this query respresented as
     * a {@link Parameter} object.
     *
     * @param parameter The parameter object
     * @param value The argument to the parameter
     * @return the same query instance
     * @throws IllegalArgumentException if the parameter does not
     *         correspond to a parameter of the query
```

```
 */
@Override
@Nonnull
<T> StoredProcedureQuery setParameter(@Nonnull Parameter<T> parameter,
                                      @Nullable T value);

/**
 * Bind an instance of {@link Calendar} to a {@link Parameter} object.
 *
 * @param param The parameter object
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter does not
 *         correspond to a parameter of the query
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2")
@Override
@Nonnull
StoredProcedureQuery setParameter(@Nonnull Parameter<Calendar> param,
                                  @Nullable Calendar value,
                                  @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a {@link Parameter} object.
 *
 * @param param The parameter object
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter does not
 *         correspond to a parameter of the query
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2")
@Override
@Nonnull
StoredProcedureQuery setParameter(@Nonnull Parameter<Date> param,
                                  @Nullable Date value,
                                  @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if the
 *         argument is of incompatible type
 */
@Override
@Nonnull
StoredProcedureQuery setParameter(@Nonnull String name, @Nullable Object value);

/**
 * Bind an instance of {@link Calendar} to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if the
```

```
 *         value argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2") @Override
@Nonnull
StoredProcedureQuery setParameter(@Nonnull String name,
                                  @Nullable Calendar value,
                                  @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a named parameter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if the
 *         value argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Deprecated(since = "3.2") @Override
@Nonnull
StoredProcedureQuery setParameter(@Nonnull String name,
                                  @Nullable Date value,
                                  @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a positional parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 */
@Override
@Nonnull
StoredProcedureQuery setParameter(int position, @Nullable Object value);

/**
 * Bind an instance of {@link Calendar} to a positional
 * parameter.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Override
@Deprecated(since = "3.2")
@Nonnull
StoredProcedureQuery setParameter(int position,
                                  @Nullable Calendar value,
                                  @Nonnull TemporalType temporalType);

/**
 * Bind an instance of {@link Date} to a positional parameter.
 *
```

```
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param temporalType A {@linkplain TemporalType temporal type}
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @deprecated Newly written code should use the date/time types
 *             defined in {@link java.time}.
 */
@Override
@Deprecated(since = "3.2")
@Nonnull
StoredProcedureQuery setParameter(int position,
                                  @Nullable Date value,
                                  @Nonnull TemporalType temporalType);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the argument might be null.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param type A class object representing the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> StoredProcedureQuery setParameter(@Nonnull String name, @Nullable P value,
                                      @Nonnull Class<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the binding is affected by an attribute converter.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param type The {@link Type} of the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> StoredProcedureQuery setParameter(@Nonnull String name, @Nullable P value,
                                      @Nonnull Type<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying an {@linkplain AttributeConverter attribute
 * converter} to use.
 *
 * @param name The name of the parameter
 * @param value The argument to the parameter
 * @param converter The class of the attribute converter
 * @return the same query instance
 * @throws IllegalArgumentException if the parameter name does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
```

```
 * @since 4.0
 */
@Override
@Nonnull
<P> StoredProcedureQuery setConvertedParameter(@Nonnull String name, @Nullable P value,
                                               @Nonnull Class<? extends AttributeConverter<P, ?>> converter);

/**
 * Bind an argument value to a positional parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the argument might be null.
 *
 * @param position position
 * @param value The argument to the parameter
 * @param type A class object representing the parameter type
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> StoredProcedureQuery setParameter(int position, @Nullable P value,
                                      @Nonnull Class<P> type);

/**
 * Bind an argument value to a positional parameter, explicitly
 * specifying the parameter type. This is most useful when
 * the binding is affected by an attribute converter.
 *
 * @param position position
 * @param value The argument to the parameter
 * @param type The {@link Type} of the parameter
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a positional parameter of the query,
 *         or if the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> StoredProcedureQuery setParameter(int position, @Nullable P value,
                                      @Nonnull Type<P> type);

/**
 * Bind an argument value to a named parameter, explicitly
 * specifying an {@linkplain AttributeConverter attribute
 * converter} to use.
 *
 * @param position The parameter position
 * @param value The argument to the parameter
 * @param converter The class of the attribute converter
 * @return the same query instance
 * @throws IllegalArgumentException if the given position does
 *         not correspond to a parameter of the query, or if
 *         the argument is of incompatible type
 * @since 4.0
 */
@Override
@Nonnull
<P> StoredProcedureQuery setConvertedParameter(int position, @Nullable P value,
                                               @Nonnull Class<? extends AttributeConverter<P, ?>> converter);

/**
 * Set the {@linkplain QueryFlushMode query flush mode} to be
 * used when the query is executed. This flush mode overrides
```

```
 * the {@linkplain EntityManager#getFlushMode flush mode type
 * of the entity manager}.
 *
 * @param flushMode The new flush mode
 * @return the same query instance
 *
 * @since 4.0
 */
@Override
@Nonnull
StoredProcedureQuery setQueryFlushMode(@Nonnull QueryFlushMode flushMode);

/**
 * Set the {@linkplain FlushModeType flush mode type} to be
 * used when the query is executed. This flush mode overrides
 * the {@linkplain EntityManager#getFlushMode flush mode type
 * of the entity manager}.
 *
 * @param flushMode The new flush mode
 * @return the same query instance
 *
 * @deprecated Use {@link #setQueryFlushMode(QueryFlushMode)}.
 */
@Override
@Deprecated
@Nonnull
StoredProcedureQuery setFlushMode(@Nonnull FlushModeType flushMode);

/**
 * Set the {@linkplain CacheRetrieveMode cache retrieval mode}
 * in effect during query execution. This cache retrieval mode
 * overrides the {@linkplain EntityManager#getCacheRetrieveMode
 * cache retrieve mode of the entity manager}.
 *
 * @param cacheRetrieveMode The new cache retrieval mode
 * @return the same query instance
 * @since 3.2
 */
@Override @SuppressWarnings("removal")
@Nonnull
StoredProcedureQuery setCacheRetrieveMode(@Nonnull CacheRetrieveMode cacheRetrieveMode);

/**
 * Set the {@linkplain CacheStoreMode cache storage mode} in
 * effect during query execution. This cache storage mode
 * overrides the {@linkplain EntityManager#getCacheStoreMode
 * cache storage mode of the entity manager}.
 *
 * @param cacheStoreMode The new cache storage mode
 * @return the same query instance
 * @since 3.2
 */
@Override @SuppressWarnings("removal")
@Nonnull
StoredProcedureQuery setCacheStoreMode(@Nonnull CacheStoreMode cacheStoreMode);

/**
 * Set the query timeout, in milliseconds. This is a hint,
 * and is an alternative to {@linkplain #setHint setting
 * the hint} {@code jakarta.persistence.query.timeout}.
 *
 * @param timeout the timeout, in milliseconds, or null to
 *                indicate no timeout
 * @return the same query instance
 * @since 3.2
 */
@Override
```

```java
@Nonnull
StoredProcedureQuery setTimeout(@Nullable Integer timeout);

/**
 * Set the query timeout. This is a hint.
 *
 * @param timeout the timeout, or null to indicate no timeout
 * @return the same query instance
 * @since 4.0
 */
@Override
@Nonnull
StoredProcedureQuery setTimeout(@Nullable Timeout timeout);

/**
 * Mark this as a call to a stored procedure with a result
 * parameter and register the type of the result parameter.
 * This is typically required when calling a stored function.
 * The result may be retrieved after execution by calling
 * {@link #getOutputParameterValue(Parameter)}.
 * @param resultType the type of the result parameter
 * @param <T> the result parameter type
 * @return an object representing the parameter, which may
 *         be passed to {@link #setParameter(Parameter, Object)}
 *         and {@link #getOutputParameterValue(Parameter)}
 * @since 4.0
 */
@Nonnull
<T> Parameter<T> registerResultParameter(@Nonnull Class<T> resultType);

/**
 * Register a positional parameter. The result of an
 * {@code OUT} parameter may be retrieved after execution
 * by calling {@link #getOutputParameterValue(Parameter)}.
 * All parameters must be registered.
 * @param position the parameter position
 * @param type the type of the parameter
 * @param mode the parameter mode
 * @param <T> the parameter type
 * @return an object representing the parameter, which may
 *         be passed to {@link #setParameter(Parameter, Object)}
 *         and {@link #getOutputParameterValue(Parameter)}
 * @since 4.0
 */
@Nonnull
<T> Parameter<T> registerParameter(int position, @Nonnull Class<T> type,
                                   @Nonnull ParameterMode mode);

/**
 * Register a named parameter. The result of an
 * {@code OUT} parameter may be retrieved after execution
 * by calling {@link #getOutputParameterValue(Parameter)}.
 * All parameters must be registered.
 * @param parameterName the name of the parameter as registered
 *                      or specified in metadata
 * @param type the type of the parameter
 * @param mode the parameter mode
 * @param <T> the parameter type
 * @return an object representing the parameter, which may
 *         be passed to {@link #setParameter(Parameter, Object)}
 *         and {@link #getOutputParameterValue(Parameter)}
 * @since 4.0
 */
@Nonnull
<T> Parameter<T> registerParameter(@Nonnull String parameterName,
                                   @Nonnull Class<T> type,
                                   @Nonnull ParameterMode mode);
```

```
/**
 * Register a positional parameter whose value is bound via
 * a {@linkplain AttributeConverter converter}. The result of
 * an {@code OUT} parameter may be retrieved after execution
 * by calling {@link #getOutputParameterValue(Parameter)}.
 * All parameters must be registered.
 * @param position the parameter position
 * @param converter the class of the attribute converter
 * @param mode the parameter mode
 * @param <T> the parameter type
 * @return an object representing the parameter, which may
 *         be passed to {@link #setParameter(Parameter, Object)}
 *         and {@link #getOutputParameterValue(Parameter)}
 * @since 4.0
 */
@Nonnull
<T> Parameter<T> registerConvertedParameter(int position,
                                            @Nonnull Class<? extends AttributeConverter<T,?>> converter,
                                            @Nonnull ParameterMode mode);

/**
 * Register a named parameter whose value is bound via a
 * {@linkplain AttributeConverter converter}. The result of
 * an {@code OUT} parameter may be retrieved after execution
 * by calling {@link #getOutputParameterValue(Parameter)}.
 * All parameters must be registered.
 * @param parameterName the name of the parameter as registered
 *                      or specified in metadata
 * @param converter the class of the attribute converter
 * @param mode the parameter mode
 * @param <T> the parameter type
 * @return an object representing the parameter, which may
 *         be passed to {@link #setParameter(Parameter, Object)}
 *         and {@link #getOutputParameterValue(Parameter)}
 * @since 4.0
 */
@Nonnull
<T> Parameter<T> registerConvertedParameter(@Nonnull String parameterName,
                                            @Nonnull Class<? extends AttributeConverter<T,?>> converter,
                                            @Nonnull ParameterMode mode);

/**
 * Register a positional parameter.
 * All parameters must be registered.
 * @param position  parameter position
 * @param type  type of the parameter
 * @param mode  parameter mode
 * @return the same query instance
 */
@Nonnull
StoredProcedureQuery registerStoredProcedureParameter(int position,
                                                      @Nonnull Class<?> type,
                                                      @Nonnull ParameterMode mode);

/**
 * Register a named parameter.
 * All parameters must be registered.
 * @param parameterName  name of the parameter as registered or
 *            specified in metadata
 * @param type  type of the parameter
 * @param mode  parameter mode
 * @return the same query instance
 */
@Nonnull
StoredProcedureQuery registerStoredProcedureParameter(@Nonnull String parameterName,
                                                      @Nonnull Class<?> type,
```

```
                                                        @Nonnull ParameterMode mode);

    /**
     * Retrieve a value passed back from the procedure
     * through an {@code INOUT} or {@code OUT} parameter.
     * For portability, all results corresponding to result sets
     * and update counts must be retrieved before the values of
     * output parameters.
     * @param position  parameter position
     * @return the result that is passed back through the parameter
     * @throws IllegalArgumentException if the position does
     *         not correspond to a parameter of the query or is
     *         not an INOUT or OUT parameter
     */
    Object getOutputParameterValue(int position);

    /**
     * Retrieve a value passed back from the procedure
     * through an {@code INOUT} or {@code OUT} parameter.
     * For portability, all results corresponding to result sets
     * and update counts must be retrieved before the values of
     * output parameters.
     *
     * @param parameterName The name of the parameter as registered
     *                      or specified in metadata
     * @return the result that is passed back through the parameter
     * @throws IllegalArgumentException if the parameter name does
     *         not correspond to a parameter of the query or is
     *         not an INOUT or OUT parameter
     */
    Object getOutputParameterValue(@Nonnull String parameterName);

    /**
     * Retrieve a value passed back from the procedure
     * through an {@code INOUT} or {@code OUT} parameter.
     * For portability, all results corresponding to result sets
     * and update counts must be retrieved before the values of
     * output parameters.
     *
     * @param parameter The parameter object
     * @param <T> the parameter type
     * @return the result that is passed back through the parameter
     * @throws IllegalArgumentException if the parameter name does
     *         not correspond to a parameter of the query or is
     *         not an {@code INOUT} or {@code OUT} parameter
     * @since 4.0
     */
    <T> T getOutputParameterValue(@Nonnull Parameter<T> parameter);

    /**
     * Return true if the first result corresponds to a result set,
     * and false if it is an update count or if there are no results
     * other than through {@code INOUT} and {@code OUT} parameters,
     * if any.
     *
     * @return {@code true} if the first result is a result set
     * @throws QueryTimeoutException if the query execution exceeds
     *         the query timeout value set and only the statement is
     *         rolled back
     * @throws PersistenceException if the query execution exceeds
     *         the query timeout value set and the transaction
     *         is rolled back
     */
    boolean execute();

    /**
     * Return the update count or {@code -1} if there is no pending
```

```
 * result or if the first result is not an update count. The
 * provider calls {@link #execute} if necessary.
 *
 * @return the update count or {@code -1} if there is no pending
 *         result or if the next result is not an update count.
 * @throws TransactionRequiredException if there is
 *         no transaction or the persistence context has not
 *         been joined to the transaction
 * @throws QueryTimeoutException if the statement execution
 *         exceeds the query timeout value set and only
 *         the statement is rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 */
@Override @SuppressWarnings("removal")
int executeUpdate();

/**
 * Retrieve the list of results from the next result set.
 * The provider calls {@link #execute} if necessary.
 * A {@code REF_CURSOR} result set, if any, is retrieved
 * in the order the {@code REF_CURSOR} parameter was
 * registered with the query.
 *
 * @return a list of the results or null if the next item is not
 *         a result set
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @deprecated This method returns a raw {@code List}.
 *             Use {@link #getResultList(Class)} instead.
 */
@Override @SuppressWarnings({"rawtypes", "removal"})
@Deprecated(since = "4.0", forRemoval = true)
@Nullable
List getResultList();

/**
 * Retrieve a single result from the next result set.
 * The provider calls {@link #execute} if necessary.
 * A {@code REF_CURSOR} result set, if any, is retrieved
 * in the order the {@code REF_CURSOR} parameter was
 * registered with the query.
 *
 * @return the result or null if the next item is not a result set
 * @throws NoResultException if there is no result in the next
 *         result set
 * @throws NonUniqueResultException if more than one result
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 */
@Override @SuppressWarnings("removal")
@Nullable
Object getSingleResult();

/**
 * Retrieve a single result from the next result set.
 * The provider calls {@link #execute} if necessary.
 * A {@code REF_CURSOR} result set, if any, is retrieved in the
```

```
 * order the {@code REF_CURSOR} parameter was registered with
 * the query.
 *
 * @return the result or null if the next item is not a result
 *         set or if there is no result in the next result set
 * @throws NonUniqueResultException if more than one result
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 */
@Override @SuppressWarnings("removal")
@Nullable
Object getSingleResultOrNull();

/**
 * Retrieve the list of results from the next result set, returning
 * instances of the given result class, which overrides any result
 * class already specified. The provider calls {@link #execute} if
 * necessary. A {@code REF_CURSOR} result set, if any, is retrieved
 * in the order the {@code REF_CURSOR} parameter was registered with
 * the query.
 * <p>Either:
 * <ul>
 * <li>the result class is an entity class and is interpreted as a
 *     managed {@linkplain EntityResult entity result} with implicit
 *     field mappings determined by the names of the columns in the
 *     result set and the object/relational mapping of the entity,
 * <li>the result class is the class of a {@linkplain Basic basic}
 *     type and the result set must have a single column which is
 *     interpreted as a {@linkplain ColumnResult scalar result},
 * <li>the result class is a non-abstract class or record type
 *     with a constructor with the same number of parameters as the
 *     result set has columns, and is interpreted as a
 *     {@linkplain ConstructorResult constructor result} including
 *     all the columns of the result set, or
 * <li>the result class is {@code Object[].class} and each query
 *     result is packaged in an array of type {@code Object[]},
 *     with the array elements corresponding by position with the
 *     columns of the select list and column values obtained
 *     according to the default type mappings defined by the JDBC
 *     specification.
 * </ul>
 *
 * @param resultClass The type of the query result
 * @param <R> the query result type
 * @return a list of the results or null if the next item is not
 *         a result set
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @since 4.0
 */
@Nullable
<R> List<R> getResultList(@Nonnull Class<R> resultClass);

/**
 * Retrieve the list of results from the next result set, specifying
 * a {@linkplain ResultSetMapping result set mapping} which overrides
 * any mapping or result class already specified. The provider calls
 * {@link #execute} if necessary. A {@code REF_CURSOR} result set,
 * if any, is retrieved in the order the {@code REF_CURSOR} parameter
```

```
 * was registered with the query.
 *
 * @param mapping The result set mapping to apply to the results
 * @param <R> the query result type
 * @return a list of the results or null if the next item is not a
 *         result set
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @since 4.0
 */
@Nullable
<R> List<R> getResultList(@Nonnull ResultSetMapping<R> mapping);

/**
 * Retrieve a single result from the next result set, returning
 * instances of the given result class, which overrides any result
 * class already specified. The provider calls {@link #execute} if
 * necessary. A {@code REF_CURSOR} result set, if any, is retrieved
 * in the order the {@code REF_CURSOR} parameter was registered with
 * the query.
 * <p>Either:
 * <ul>
 * <li>the result class is an entity class and is interpreted as a
 *     managed {@linkplain EntityResult entity result} with implicit
 *     field mappings determined by the names of the columns in the
 *     result set and the object/relational mapping of the entity,
 * <li>the result class is the class of a {@linkplain Basic basic}
 *     type and the result set must have a single column which is
 *     interpreted as a {@linkplain ColumnResult scalar result},
 * <li>the result class is a non-abstract class or record type
 *     with a constructor with the same number of parameters as the
 *     result set has columns, and is interpreted as a
 *     {@linkplain ConstructorResult constructor result} including
 *     all the columns of the result set, or
 * <li>the result class is {@code Object[].class} and each query
 *     result is packaged in an array of type {@code Object[]},
 *     with the array elements corresponding by position with the
 *     columns of the select list and column values obtained
 *     according to the default type mappings defined by the JDBC
 *     specification.
 * </ul>
 *
 * @param resultClass The type of the query result
 * @param <R> the query result type
 * @return the result or null if the next item is not a result set
 * @throws NoResultException if there is no result in the next
 *         result set
 * @throws NonUniqueResultException if more than one result
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @since 4.0
 */
@Nullable
<R> R getSingleResult(@Nonnull Class<R> resultClass);

/**
 * Retrieve a single result from the next result set, specifying a
 * {@linkplain ResultSetMapping result set mapping} which overrides
 * any mapping or result class already specified. The provider calls
```

```
 * {@link #execute} if necessary. A {@code REF_CURSOR} result set,
 * if any, is retrieved in the order the {@code REF_CURSOR} parameter
 * was registered with the query.
 *
 * @param mapping The result set mapping to apply to the results
 * @param <R> the query result type
 * @return the result or null if the next item is not a result set
 * @throws NoResultException if there is no result in the next
 *         result set
 * @throws NonUniqueResultException if more than one result
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @since 4.0
 */
@Nullable
<R> R getSingleResult(@Nonnull ResultSetMapping<R> mapping);

/**
 * Retrieve a single result from the next result set, returning
 * instances of the given result class, which overrides any result
 * class already specified. The provider calls {@link #execute} if
 * necessary. A {@code REF_CURSOR} result set, if any, is retrieved
 * in the order the {@code REF_CURSOR} parameter was registered with
 * the query.
 * <p>Either:
 * <ul>
 * <li>the result class is an entity class and is interpreted as a
 *     managed {@linkplain EntityResult entity result} with implicit
 *     field mappings determined by the names of the columns in the
 *     result set and the object/relational mapping of the entity,
 * <li>the result class is the class of a {@linkplain Basic basic}
 *     type and the result set must have a single column which is
 *     interpreted as a {@linkplain ColumnResult scalar result},
 * <li>the result class must be a non-abstract class or record type
 *     with a constructor with the same number of parameters as the
 *     result set has columns, and is interpreted as a
 *     {@linkplain ConstructorResult constructor result} including
 *     all the columns of the result set, or
 * <li>the result class is {@code Object[].class} and each query
 *     result is packaged in an array of type {@code Object[]},
 *     with the array elements corresponding by position with the
 *     columns of the select list and column values obtained
 *     according to the default type mappings defined by the JDBC
 *     specification.
 * </ul>
 *
 * @param resultClass The type of the query result
 * @param <R> the query result type
 * @return the result or null if the next item is not a result set
 *         or if there is no result in the next result set
 * @throws NonUniqueResultException if more than one result
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @since 4.0
 */
@Nullable
<R> R getSingleResultOrNull(@Nonnull Class<R> resultClass);

/**
```

```
 * Retrieve a single result from the next result set, specifying a
 * {@linkplain ResultSetMapping result set mapping} which overrides
 * any mapping or result class already specified. The provider calls
 * {@link #execute} if necessary. A {@code REF_CURSOR} result set,
 * if any, is retrieved in the order the {@code REF_CURSOR} parameter
 * was registered with the query.
 * <p>Either:
 * <ul>
 * <li>the result class is an entity class and is interpreted as a
 *     managed {@linkplain EntityResult entity result} with implicit
 *     field mappings determined by the names of the columns in the
 *     result set and the object/relational mapping of the entity,
 * <li>the result class is the class of a {@linkplain Basic basic}
 *     type and the result set must have a single column which is
 *     interpreted as a {@linkplain ColumnResult scalar result}, or
 * <li>the result class must be a non-abstract class or record type
 *     with a constructor with the same number of parameters as the
 *     result set has columns, and is interpreted as a
 *     {@linkplain ConstructorResult constructor result} including
 *     all the columns of the result set.
 * </ul>
 *
 * @param mapping The result set mapping to apply to the results
 * @param <R> the query result type
 * @return the result or null if the next item is not a result set
 *         or if there is no result in the next result set
 * @throws NonUniqueResultException if more than one result
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 * @since 4.0
 */
@Nullable
<R> R getSingleResultOrNull(@Nonnull ResultSetMapping<R> mapping);

/**
 * Return true if the next result corresponds to a result set,
 * and false if it is an update count or if there are no results
 * other than through {@code INOUT} and {@code OUT} parameters,
 * if any.
 *
 * @return {@code true} if the next result is a result set
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 */
boolean hasMoreResults();

/**
 * Return the update count or {@code -1} if there is no pending
 * result or if the next result is not an update count.
 *
 * @return the update count or {@code -1} if there is no pending
 *         result or if the next result is not an update count
 * @throws QueryTimeoutException if the query execution exceeds
 *         the query timeout value set and only the statement is
 *         rolled back
 * @throws PersistenceException if the query execution exceeds
 *         the query timeout value set and the transaction
 *         is rolled back
 */
```

```
    int getUpdateCount();

    /**
     * Immediately destroy all resources associated with this query.
     * If the client does not call this method before the end of the
     * current transaction, the behavior is undefined. A provider
     * might release resources when all query results are exhausted,
     * at the end of the current transaction, or when the entity
     * manager or entity agent is closed.
     * <p>After invocation of {@code close()}, every method of the
     * {@code StoredProcedureQuery} throws {@code IllegalStateException}.
     *
     * @since 4.0
     */
    @Override
    void close();

    /**
     * Specify an {@linkplain Option option} influencing execution
     * of this stored procedure, overwriting any existing option of
     * the same type.
     *
     * @param option the option
     * @return the same statement instance
     * @since 4.0
     */
    @Nonnull
    StoredProcedureQuery addOption(@Nonnull Option option);

    /**
     * Get the {@linkplain Option options} influencing execution of
     * this stored procedure. The returned set includes options set
     * via {@link #addOption}, along with options specified via
     * {@link #setTimeout} or {@link #setQueryFlushMode}. Mutation of
     * the returned set does not affect the options of the stored
     * procedure.
     *
     * @return the options for this stored procedure
     * @since 4.0
     */
    @Nonnull
    Set<Option> getOptions();

    /**
     * An option influencing execution of a stored procedure. This
     * provides a more type safe alternative to the use of
     * {@linkplain #setHint hints}.
     *
     * <p>This interface may be implemented by custom provider-specific
     * options which extend the options defined by the specification.
     *
     * @see QueryFlushMode
     * @see Timeout
     *
     * @since 4.0
     */
    interface Option {
    }
}
```

B.13. Tuple

```
import jakarta.annotation.Nonnull;

import java.util.List;
```

```java
/**
 * Interface for extracting the elements of a query result tuple.
 *
 * @see TupleElement
 *
 * @since 2.0
 */
public interface Tuple {

    /**
     * Get the value of the specified tuple element.
     * @param tupleElement  tuple element
     * @param <X> the type of the tuple element
     * @return value of tuple element
     * @throws IllegalArgumentException if tuple element
     *         does not correspond to an element in the
     *         query result tuple
     */
    <X> X get(@Nonnull TupleElement<X> tupleElement);

    /**
     * Get the value of the tuple element to which the
     * specified alias has been assigned.
     * @param alias  alias assigned to tuple element
     * @param type of the tuple element
     * @param <X> the type of the tuple element
     * @return value of the tuple element
     * @throws IllegalArgumentException if alias
     *         does not correspond to an element in the
     *         query result tuple or element cannot be
     *         assigned to the specified type
     */
    <X> X get(@Nonnull String alias, @Nonnull Class<X> type);

    /**
     * Get the value of the tuple element to which the
     * specified alias has been assigned.
     * @param alias  alias assigned to tuple element
     * @return value of the tuple element
     * @throws IllegalArgumentException if alias
      *        does not correspond to an element in the
     *         query result tuple
     */
    Object get(@Nonnull String alias);

    /**
     * Get the value of the element at the specified
     * position in the result tuple. The first position
     * is 0.
     * @param i  position in result tuple
     * @param type  type of the tuple element
     * @param <X> the type of the tuple element
     * @return value of the tuple element
     * @throws IllegalArgumentException if the position
     *         exceeds the length of the result tuple or
     *         if the element cannot be assigned to the
     *         specified type
     */
    <X> X get(int i, @Nonnull Class<X> type);

    /**
     * Get the value of the element at the specified
     * position in the result tuple. The first position
     * is 0.
     * @param i  position in result tuple
     * @return value of the tuple element
```

```
     * @throws IllegalArgumentException if i exceeds
     *         length of result tuple
     */
    Object get(int i);

    /**
     * Return the values of the result tuple elements as
     * an array.
     * @return tuple element values
     */
    @Nonnull
    Object[] toArray();

    /**
     * Return the tuple elements.
     * @return tuple elements
     */
    @Nonnull
    List<TupleElement<?>> getElements();
}
```

B.14. TupleElement

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

/**
 * The {@code TupleElement} interface defines an element that is
 * returned in a query result tuple.
 *
 * @param <X> the type of the element
 *
 * @see Tuple
 *
 * @since 2.0
 */
public interface TupleElement<X> {

    /**
     * Return the Java type of the tuple element.
     * @return the Java type of the tuple element
     */
    @Nonnull
    Class<? extends X> getJavaType();

    /**
     * Return the alias assigned to the tuple element or null,
     * if no alias has been assigned.
     * @return alias
     */
    @Nullable
    String getAlias();
}
```

B.15. Parameter

```
/**
 * Represents a parameter of a {@linkplain Query query}.
 * <p>
 * A portable application should not attempt to reuse a {@code Parameter}
 * object obtained from one instance of {@code Query} by passing it to a
 * method of a different instance of {@code Query}.
 *
```

```
 * @param <T> the type of the parameter
 *
 * @see Query
 * @see TypedQuery
 *
 * @since 2.0
 */
public interface Parameter<T> {

    /**
     * Return the parameter name, or null if the parameter is
     * not a named parameter or no name has been assigned.
     * @return parameter name
     */
    String getName();

    /**
     * Return the parameter position, or null if the parameter
     * is not a positional parameter.
     * @return position of parameter
     */
    Integer getPosition();

    /**
     * Return the Java type of the parameter. Values bound to
     * the parameter must be assignable to this type.
     * This method is required to be supported for criteria
     * queries only. Applications that use this method for
     * Jakarta Persistence query language queries and native
     * queries will not be portable.
     * @return the Java type of the parameter
     * @throws IllegalStateException if invoked on a parameter
     *         obtained from a query language query or native
     *         query when the implementation does not support
     *         this usage
     */
     Class<T> getParameterType();
}
```

B.16. Graph

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.Attribute;
import jakarta.persistence.metamodel.ManagedType;
import jakarta.persistence.metamodel.MapAttribute;
import jakarta.persistence.metamodel.PluralAttribute;

import java.util.List;

/**
 * Declares operations common to {@link EntityGraph} and {@link Subgraph},
 * most importantly, operations for adding and removing attribute nodes
 * and for creating subgraphs.
 * <p>
 * Any method creating a {@linkplain Subgraph subgraph} of the graph also
 * implicitly creates the {@linkplain AttributeNode attribute node} at
 * which the subgraph is rooted; it is redundant to separately create such
 * an attribute node. Removing an attribute node also removes any subgraph
 * rooted at that node.
 * <p>
 * When a graph is to be interpreted as a load graph, it is meaningful to
 * remove an attribute node which has not been explicitly added to the graph.
 * This operation is interpreted to suppress the inclusion of an attribute
 * mapped for eager fetching, which would otherwise be included by default.
 *
```

```
 * @see EntityGraph
 * @see Subgraph
 *
 * @param <T> The type of the root entity.
 *
 * @since 3.2
 */
public interface Graph<T> {

    /**
     * The managed type at which this graph or subgraph is rooted.
     *
     * @return The metamodel object representing the graphed type
     *
     * @since 4.0
     */
    @Nonnull
    ManagedType<T> getGraphedType();

    /**
     * Get an existing attribute node for the attribute with the given
     * name, or add a new attribute node if there is no existing node,
     * cancelling the effect of any prior
     * {@linkplain #removeAttributeNode removal}.
     *
     * <p>Added nodes are reflected in the list of
     * {@linkplain #getAttributeNodes child nodes} as instances of
     * {@link AttributeNode} with option {@link FetchType#EAGER}.
     *
     * @param attributeName the name of an attribute of the managed type
     * @param <Y> the type of the attribute
     * @return the attribute node
     * @throws IllegalArgumentException if there is no attribute
     *         for the given name
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #addAttributeNode(Attribute)}.
     *          Use of the typesafe version is strongly preferred.
     *
     * @since 3.2
     */
    @Nonnull
    <Y> AttributeNode<Y> addAttributeNode(@Nonnull String attributeName);

    /**
     * Get an existing attribute node for the given attribute, or add
     * a new attribute node if there is no existing node, cancelling
     * the effect of any prior {@linkplain #removeAttributeNode removal}.
     *
     * <p>Added nodes are reflected in the list of
     * {@linkplain #getAttributeNodes child nodes} as instances of
     * {@link AttributeNode} with option {@link FetchType#EAGER}.
     *
     * @param attribute  attribute
     * @param <Y> the type of the attribute
     * @return the attribute node
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     *
     * @since 3.2
     */
    @Nonnull
    <Y> AttributeNode<Y> addAttributeNode(@Nonnull Attribute<? super T, Y> attribute);
```

```java
/**
 * Determine if there is an existing attribute node for the attribute
 * with the given name.
 *
 * @param attributeName the name of an attribute of the managed type
 * @return true if there is an existing attribute node
 * @throws IllegalArgumentException if the attribute is not an
 *         attribute of this entity.
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #hasAttributeNode(Attribute)}.
 *          Use of the typesafe version is strongly preferred.
 *
 * @since 3.2
 */
boolean hasAttributeNode(@Nonnull String attributeName);

/**
 * Determine if there is an existing attribute node for the given
 * attribute.
 *
 * @param attribute  attribute
 * @return true if there is an existing attribute node
 *
 * @since 3.2
 */
boolean hasAttributeNode(@Nonnull Attribute<? super T, ?> attribute);

/**
 * Get an existing attribute node for the attribute with the given
 * name.
 *
 * @param attributeName the name of an attribute of the managed type
 * @return the attribute node
 * @throws IllegalArgumentException if the attribute is not an
 *         attribute of this entity.
 * @throws java.util.NoSuchElementException if there is no existing
 *         node for the attribute
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #getAttributeNode(Attribute)}.
 *          Use of the typesafe version is strongly preferred.
 *
 * @param <Y> the type of the attribute
 * @since 3.2
 */
@Nonnull
<Y> AttributeNode<Y> getAttributeNode(@Nonnull String attributeName);

/**
 * Get an existing attribute node for the given attribute.
 *
 * @param attribute  attribute
 * @param <Y> the type of the attribute
 * @return the attribute node
 * @throws java.util.NoSuchElementException if there is no existing
 *         node for the attribute
 *
 * @since 3.2
 */
@Nonnull
```

```
<Y> AttributeNode<Y> getAttributeNode(@Nonnull Attribute<? super T, Y> attribute);

/**
 * Remove an attribute node from the entity graph, cancelling the
 * effect of any prior {@linkplain #addAttributeNode addition}.
 * When this graph is interpreted as a load graph, this operation
 * suppresses inclusion of an attribute mapped for eager fetching.
 * The effect of this call may be overridden by subsequent
 * invocations of {@link #addAttributeNode} or {@link #addSubgraph}.
 *
 * <p>Removed nodes are reflected in the list of
 * {@linkplain #getAttributeNodes child nodes} as instances of
 * {@link AttributeNode} with option {@link FetchType#LAZY}.
 *
 * @param attributeName the name of an attribute of the managed
 *                      type
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #removeAttributeNode(Attribute)}.
 *          Use of the typesafe version is strongly preferred.
 *
 * @since 3.2
 */
void removeAttributeNode(@Nonnull String attributeName);

/**
 * Remove an attribute node from the entity graph, cancelling the
 * effect of any prior {@linkplain #addAttributeNode addition}.
 * When this graph is interpreted as a load graph, this operation
 * suppresses inclusion of an attribute mapped for eager fetching.
 * The effect of this call may be overridden by subsequent
 * invocations of {@link #addAttributeNode} or {@link #addSubgraph}.
 *
 * <p>Removed nodes are reflected in the list of
 * {@linkplain #getAttributeNodes child nodes} as instances of
 * {@link AttributeNode} with option {@link FetchType#LAZY}.
 *
 * @apiNote This operation does not literally remove a node from
 * the entity graph object; instead, it suppresses the inclusion
 * of an edge in the logical graph of fetched associations that
 * is inferred from this entity graph when it is interpreted as
 * a load graph or fetch graph in the context of mapping metadata.
 *
 * @param attribute  attribute
 *
 * @since 3.2
 */
void removeAttributeNode(@Nonnull Attribute<? super T, ?> attribute);

/**
 * Remove all attribute nodes of the given attribute type, cancelling
 * the effect of any prior {@linkplain #addAttributeNode additions}.
 * When this graph is interpreted as a load graph, this operation
 * suppresses inclusion of attributes mapped for eager fetching.
 * The effect of this call may be overridden by subsequent
 * invocations of {@link #addAttributeNode} or {@link #addSubgraph}.
 *
 * <p>Removed nodes are reflected in the list of
 * {@linkplain #getAttributeNodes child nodes} as instances of
 * {@link AttributeNode} with option {@link FetchType#LAZY}.
 *
 * @param nodeType the type of attribute to remove
 *
 * @since 3.2
```

```
 */
void removeAttributeNodes(@Nonnull Attribute.PersistentAttributeType nodeType);

/**
 * Add one or more attribute nodes to the entity graph,
 * cancelling the effect of any prior
 * {@linkplain #removeAttributeNode removals}.
 * If there is already an existing node for one of the given
 * attribute names, and it is not marked for removal, that
 * particular argument is ignored and has no effect.
 *
 * <p>Added nodes are reflected in the list of
 * {@linkplain #getAttributeNodes child nodes} as instances of
 * {@link AttributeNode} with option {@link FetchType#EAGER}.
 *
 * @param attributeName the name of an attribute of the managed
 *                      type
 * @throws IllegalArgumentException if there is no attribute
 *         for one of the given names
 * @throws IllegalStateException if the EntityGraph has been
 *         statically defined
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #addAttributeNodes(Attribute...)}.
 *          Use of the typesafe version is strongly preferred.
 */
void addAttributeNodes(@Nonnull String... attributeName);

/**
 * Add one or more attribute nodes to the entity graph,
 * cancelling the effect of any prior
 * {@linkplain #removeAttributeNode removals}.
 * If there is already an existing node for one of the given
 * attributes, and it is not marked for removal, that particular
 * argument is ignored and has no effect.
 *
 * <p>Added nodes are reflected in the list of
 * {@linkplain #getAttributeNodes child nodes} as instances of
 * {@link AttributeNode} with option {@link FetchType#EAGER}.
 *
 * @param attribute  attribute
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 */
void addAttributeNodes(@Nonnull Attribute<? super T, ?>... attribute);

/**
 * Add a node to the graph representing a managed type. This
 * allows for construction of multi-node entity graphs that
 * include related managed types.
 *
 * @param attribute the attribute
 * @param <X> the managed type of the attribute
 * @return subgraph for the attribute
 * @throws IllegalArgumentException if the attribute's target
 *         type is not a managed type
 * @throws IllegalStateException if the EntityGraph has been
 *         statically defined
 */
@Nonnull
<X> Subgraph<X> addSubgraph(@Nonnull Attribute<? super T, X> attribute);

/**
 * Add a node to the graph representing a managed type with
 * inheritance. This allows for multiple subclass subgraphs
 * to be defined for this node of the entity graph. Subclass
 * subgraphs automatically include the specified attributes
```

```
 * of superclass subgraphs.
 *
 * @param attribute the attribute
 * @param type an entity subclass of the attribute type
 * @param <Y> the type of the subclass
 * @return subgraph for the attribute
 * @throws IllegalArgumentException if the attribute's target
 *         type is not a managed type
 * @throws IllegalStateException if the EntityGraph has been
 *         statically defined
 *
 * @since 3.2
 */
@Nonnull
<Y> Subgraph<Y> addTreatedSubgraph(@Nonnull Attribute<? super T, ? super Y> attribute,
                                   @Nonnull Class<Y> type);

/**
 * Add a node to the graph representing a managed type. This
 * allows for construction of multi-node entity graphs that
 * include related managed types.
 *
 * @param attributeName the name of an attribute of the managed
 *                      type
 * @param <X> the managed type of the attribute
 * @return subgraph for the attribute
 * @throws IllegalArgumentException if there is no attribute
 *         for the given name
 * @throws IllegalArgumentException if the attribute's target
 *         type is not a managed type
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #addSubgraph(Attribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<X> Subgraph<X> addSubgraph(@Nonnull String attributeName);

/**
 * Add a node to the graph representing a managed type with
 * inheritance. This allows for multiple subclass subgraphs
 * to be defined for this node of the entity graph. Subclass
 * subgraphs automatically include the specified attributes
 * of superclass subgraphs.
 *
 * @param attributeName the name of an attribute of the managed
 *                      type
 * @param type an entity subclass of the collection element type
 * @param <X> the type of the subclass
 * @return subgraph for the attribute
 * @throws IllegalArgumentException if there is no attribute
 *         for the given name
 * @throws IllegalArgumentException if the attribute's target
 *         type is not a managed type
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #addSubgraph(Attribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<X> Subgraph<X> addSubgraph(@Nonnull String attributeName,
```

```
                                @Nonnull Class<X> type);

/**
 * Add a node to the graph representing a collection element
 * that is a managed type. This allows for construction of
 * multi-node entity graphs that include related managed types.
 *
 * @param attribute the attribute
 * @param <E> the managed type of the collection element
 * @return subgraph for the element attribute
 * @throws IllegalArgumentException if the attribute's target type
 *         is not an entity
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 *
 * @since 3.2
 */
@Nonnull
<E> Subgraph<E> addElementSubgraph(@Nonnull PluralAttribute<? super T, ?, E> attribute);

/**
 * Add a node to the graph representing a collection element
 * that is a managed type. This allows for construction of
 * multi-node entity graphs that include related managed types.
 *
 * @param attribute the attribute
 * @param type an entity subclass of the collection element type
 * @param <E> the managed type of the collection element
 * @return subgraph for the element attribute
 * @throws IllegalArgumentException if the attribute's target type
 *         is not an entity
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 *
 * @since 3.2
 */
@Nonnull
<E> Subgraph<E> addTreatedElementSubgraph(@Nonnull PluralAttribute<? super T, ?, ? super E> attribute,
                                          @Nonnull Class<E> type);

/**
 * Add a node to the graph representing a collection element
 * that is a managed type. This allows for construction of
 * multi-node entity graphs that include related managed types.
 *
 * @param attributeName the name of an attribute of the managed
 *                      type
 * @param <X> the managed type of the collection element
 * @return subgraph for the element attribute
 * @throws IllegalArgumentException if there is no attribute
 *         for the given name
 * @throws IllegalArgumentException if the attribute's target
 *         type is not a managed type
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #addElementSubgraph(PluralAttribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<X> Subgraph<X> addElementSubgraph(@Nonnull String attributeName);

/**
 * Add a node to the graph that representing a collection element
 * that is a managed type. This allows for construction of
```

```
 * multi-node entity graphs that include related managed types.
 *
 * @param attributeName the name of an attribute of the managed
 *                      type
 * @param type an entity subclass of the collection element type
 * @param <X> the type of the subclass
 * @return subgraph for the element attribute
 * @throws IllegalArgumentException if there is no attribute
 *         for the given name
 * @throws IllegalArgumentException if the attribute's target
 *         type is not a managed type
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #addElementSubgraph(PluralAttribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<X> Subgraph<X> addElementSubgraph(@Nonnull String attributeName,
                                   @Nonnull Class<X> type);

/**
 * Add a node to the graph representing a map key that is a
 * managed type. This allows for construction of multi-node
 * entity graphs that include related managed types.
 *
 * @param attribute the attribute
 * @param <K> the managed type of the map key
 * @return subgraph for the key attribute
 * @throws IllegalArgumentException if the attribute's target
 *         type is not a managed type entity
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 */
@Nonnull
<K> Subgraph<K> addMapKeySubgraph(@Nonnull MapAttribute<? super T, K, ?> attribute);

/**
 * Add a node to the graph representing a map key that is a
 * managed type with inheritance. This allows for construction
 * of multi-node entity graphs that include related managed
 * types. Subclass subgraphs automatically include the
 * specified attributes of superclass subgraphs.
 *
 * @param attribute the attribute
 * @param type an entity subclass of the map key type
 * @param <K> the type of the subclass
 * @return subgraph for the attribute
 * @throws IllegalArgumentException if the attribute's target
 *         type is not a managed type entity
 * @throws IllegalStateException if this EntityGraph has been
 *         statically defined
 */
@Nonnull
<K> Subgraph<K> addTreatedMapKeySubgraph(@Nonnull MapAttribute<? super T, ? super K, ?> attribute,
                                         @Nonnull Class<K> type);

/**
 * Add a node to the graph representing a map key that is a
 * managed type. This allows for construction of multi-node
 * entity graphs that include related managed types.
 *
 * @param attributeName the name of an attribute of the managed
 *                      type
 * @param <X> the managed type of the map key
```

```
     * @return subgraph for the key attribute
     * @throws IllegalArgumentException if there is no attribute
     *         for the given name
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #addMapKeySubgraph(MapAttribute)}.
     *          Use of the typesafe version is strongly preferred.
     */
    @Nonnull
    <X> Subgraph<X> addKeySubgraph(@Nonnull String attributeName);

    /**
     * Add a node to the graph representing a map key that is a
     * managed type with inheritance. This allows for construction
     * of multi-node entity graphs that include related managed types.
     * Subclass subgraphs will include the specified attributes of
     * superclass subgraphs
     *
     * @param attributeName the name of an attribute of the managed
     *                      type
     * @param type an entity subclass of the map key type
     * @param <X> the type of the subclass
     * @return subgraph for the attribute
     * @throws IllegalArgumentException if there is no attribute
     *         for the given name
     * @throws IllegalArgumentException if the attribute's target
     *         type is not a managed type
     * @throws IllegalStateException if this EntityGraph has been
     *         statically defined
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #addMapKeySubgraph(MapAttribute)}.
     *          Use of the typesafe version is strongly preferred.
     */
    @Nonnull
    <X> Subgraph<X> addKeySubgraph(@Nonnull String attributeName,
                                   @Nonnull Class<X> type);

    /**
     * Return the attribute nodes representing the persistent
     * attributes of this managed type that were explicitly
     * {@linkplain #addAttributeNode included in (added to)} or
     * {@linkplain #removeAttributeNode excluded from (removed from)}
     * the graph. The {@linkplain AttributeNode#getOptions options}
     * of the returned nodes may be used to distinguish added nodes
     * from removed nodes.
     * <ul>
     * <li>Added nodes are represented by instances of
     *     {@link AttributeNode} with option {@link FetchType#EAGER}.
     * <li>Removed nodes are represented by instances of
     *     {@link AttributeNode} with option {@link FetchType#LAZY}.
     * </ul>
     * @return list of attribute nodes added to or removed from the
     *         graph, or an empty list if no nodes have been defined
     */
    @Nonnull
    List<AttributeNode<?>> getAttributeNodes();

}
```

B.17. EntityGraph

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

/**
 * An entity graph is a template that captures the boundaries of an
 * operation or query. Every entity graph has a <em>root entity</em>.
 * An instance of {@code EntityGraph} is the root node of a graph and
 * represents this entity. The child nodes of the root node represent
 * persistent attributes, embedded objects, or associations to other
 * entities. A node representing an embedded object or an association
 * to an entity might be the root of a subgraph with its own child
 * nodes.
 * <p>
 * An entity graph is usually used to control the data fetched from
 * the database. The {@linkplain AttributeNode attribute nodes} and
 * {@linkplain Subgraph subgraphs} of an entity graph determine the
 * limits of the graph of associated attributes and entities fetched
 * when an operation retrieves an instance or instances of the root
 * entity of the graph.
 * <p>
 * When used to specify fetching, an entity graph has two possible
 * interpretations:
 * <ul>
 * <li>As a <em>load graph</em>, where every node explicitly added
 *     to or explicitly removed from the graph overrides the
 *     {@linkplain FetchType fetching strategy} of the attribute
 *     which was specified via annotations or XML descriptor, but
 *     the graph does not affect the fetching strategy of any
 *     attribute which was neither added to nor removed from the
 *     graph.
 * <li>As a <em>fetch graph</em>, where the graph completely
 *     overrides every fetching strategy specified via annotations
 *     or XML descriptor, and every attribute not explicitly added
 *     to the graph is treated as {@link FetchType#LAZY}.
 * </ul>
 * <p>
 * An entity graph passed as the first argument to
 * {@link EntityManager#find(EntityGraph, Object, FindOption...)},
 * {@link EntityManager#get(EntityGraph, Object, FindOption...)},
 * {@link EntityHandler#createQuery(String, EntityGraph)}, or
 * {@link StatementOrTypedQuery#withEntityGraph(EntityGraph)} is
 * interpreted as a load graph.
 * {@snippet :
 * // create the root node of an entity graph
 * var employeeGraph = Employee_.class_.createEntityGraph();
 * // define a subgraph rooted at a @ManyToOne association
 * var employerGraph = employeeGraph.addSubgraph(Employee_.employer);
 * employerGraph.addAttributeNode(Employer_.industry);
 * // define a subgraph rooted at a @OneToMany association
 * var projectGraph = employeeGraph.addElementSubgraph(Employee_.projects);
 * projectGraph.addAttributeNode(Project_.lead);
 * projectGraph.addAttributeNode(Project_.tasks);
 * // load an employee, making use of the graph to fetch a tree of
 * // associated objects (the graph is interpreted as a load graph)
 * Employee employee = entityManager.get(employeeGraph, employeeId);
 *}
 * <p>
 * The persistence provider is always permitted to fetch additional
 * entity state beyond that specified by a fetch graph or load graph.
 * It is required, however, that the persistence provider fetch all
 * state specified by the fetch or load graph.
 *
 * @apiNote Support for lazy fetching of {@linkplain Basic basic}
 * fields and properties varies between persistence providers and
```

```
 * even between container environments. Therefore, the behavior of
 * a fetch graph, or of a load graph which explicitly removes a
 * {@linkplain Basic basic} attribute, is likely to lack portability.
 *
 * @param <T> The type of the root entity.
 *
 * @see AttributeNode
 * @see Subgraph
 * @see NamedEntityGraph
 *
 * @see EntityHandler#createEntityGraph(Class)
 * @see EntityHandler#getEntityGraph(String)
 * @see EntityHandler#getEntityGraph(Class,String)
 * @see EntityManagerFactory#addNamedEntityGraph(String, EntityGraph)
 * @see EntityManagerFactory#getNamedEntityGraphs(Class)
 * @see EntityHandler#find(EntityGraph, Object, FindOption...)
 * @see EntityHandler#get(EntityGraph, Object, FindOption...)
 * @see EntityHandler#createQuery(String, EntityGraph)
 * @see StatementOrTypedQuery#withEntityGraph(EntityGraph)
 *
 * @since 2.1
 */
public interface EntityGraph<T> extends Graph<T> {

    /**
     * Return the name of a named {@code EntityGraph} (an entity
     * graph defined by means of the {@link NamedEntityGraph}
     * annotation, XML descriptor element, or added by means of the
     * {@link EntityManagerFactory#addNamedEntityGraph} method).
     * Returns null if the {@code EntityGraph} is not a named
     * {@code EntityGraph}.
     */
    @Nullable
    String getName();

    /**
     * Add additional attributes to this entity graph that
     * correspond to attributes of subclasses of the entity type of
     * this {@code EntityGraph}. Subclass subgraphs automatically
     * include the specified attributes of superclass subgraphs.
     *
     * @param type an entity subclass of the root entity class
     * @param <S> the type of the subclass
     * @return subgraph for the subclass
     * @throws IllegalArgumentException if the type is not an entity type
     * @throws IllegalStateException if the EntityGraph has been
     *         statically defined
     */
    @Nonnull
    <S extends T> Subgraph<S> addTreatedSubgraph(@Nonnull Class<S> type);

}
```

B.18. Subgraph

```
import jakarta.annotation.Nonnull;

/**
 * This type represents a subgraph for an attribute node that
 * corresponds to a managed type. Using this class, an entity
 * subgraph can be embedded within an {@link EntityGraph}.
 *
 * @param <T> The type of the attribute.
 *
 * @see EntityGraph
```

```
 * @see AttributeNode
 * @see NamedSubgraph
 *
 * @since 2.1
 */
public interface Subgraph<T> extends Graph<T> {

    /**
     * Return the type for which this subgraph was defined.
     * @return managed type referenced by the subgraph
     */
    @Nonnull
    Class<T> getClassType();

}
```

B.19. AttributeNode

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.Attribute;

import java.util.Map;
import java.util.Set;

/**
 * Represents an attribute node of an entity graph.
 *
 * @param <T> The type of the attribute.
 *
 * @see EntityGraph
 * @see Subgraph
 * @see NamedAttributeNode
 *
 * @since 2.1
 */
public interface AttributeNode<T> {

    /**
     * Return the name of the attribute corresponding to the
     * attribute node.
     * @return name of the attribute
     */
    @Nonnull
    String getAttributeName();

    /**
     * The attribute from which this branch of the graph or subgraph is forked.
     *
     * @return The metamodel object representing the graphed attribute
     *
     * @since 4.0
     */
    @Nonnull
    Attribute<?,T> getAttribute();

    /**
     * Add a subgraph rooted at this attribute node, which
     * must be an embedded attribute or an association to
     * an entity class.
     * @return the added subgraph
     * @see EntityGraph#addSubgraph(Attribute)
     * @since 4.0
     */
    @Nonnull
    Subgraph<T> addSubgraph();
```

```java
    /**
     * Add a subgraph rooted at this attribute node, which
     * must be an association to an entity class, for an
     * entity subclass of the entity class.
     * @param type The subclass of the entity class
     * @param <S> The type of the subclass
     * @return the added subgraph
     * @see EntityGraph#addTreatedSubgraph(Attribute, Class)
     * @since 4.0
     */
    @Nonnull
    <S extends T> Subgraph<S> addTreatedSubgraph(@Nonnull Class<S> type);

    /**
     * Return a map of subgraphs associated with this attribute
     * node.
     * @return a {@link Map} of subgraphs associated with this
     * attribute node or an empty {@code Map} if none have been
     * defined
     */
    @Nonnull
    Map<Class<?>, Subgraph<?>> getSubgraphs();

    /**
     * Return a map of subgraphs associated with this attribute
     * node's map key.
     * @return a {@link Map} of subgraphs associated with this
     * attribute node's map key or an empty {@code Map} if none
     * have been defined
     */
    @Nonnull
    Map<Class<?>, Subgraph<?>> getKeySubgraphs();

    /**
     * Specify an {@linkplain FetchOption option} controlling
     * how this node is fetched, overwriting any existing
     * option of the same type.
     * @param option The option
     * @return the receiving instance
     * @since 4.0
     */
    @Nonnull
    AttributeNode<T> addOption(@Nonnull FetchOption option);

    /**
     * Return the {@linkplain FetchOption options} controlling
     * how this node is fetched.
     * <ul>
     * <li>If this is an {@linkplain Graph#addAttributeNode added}
     *     node, the returned options contain {@link FetchType#EAGER}.
     * <li>If this is a {@linkplain Graph#removeAttributeNode removed}
     *     node, the returned options contain {@link FetchType#LAZY}.
     * </ul>
     * Mutation of the returned set does not affect the options
     * of the node.
     * @return The options for this node
     * @since 4.0
     */
    @Nonnull
    Set<FetchOption> getOptions();
}
```

B.20. AttributeConverter

```
/**
 * Interface implemented by custom attribute <em>converters</em>. A
 * converter is a class whose methods convert between:
 * <ul>
 * <li>the <em>target type</em> of the converter, an arbitrary Java
 *     type which may be used as the type of a persistent field or
 *     property, and
 * <li>a {@linkplain Basic basic type} used as an intermediate step
 *     in mapping to the database representation.
 * </ul>
 *
 * <p>A converted field or property is considered {@link Basic}, since,
 * with the aid of the converter, its values can be represented as
 * instances of a basic type.
 *
 * <p>A converter class must be annotated {@link Converter} or declared
 * as a converter in the object/relational mapping descriptor. The value
 * of {@link Converter#autoApply autoApply} determines if the converter
 * is automatically applied to persistent fields and properties of the
 * target type. The {@link Convert} annotation may be used to apply a
 * converter which is declared {@code autoApply=false}, to explicitly
 * {@linkplain Convert#disableConversion disable conversion}, or to
 * resolve ambiguities when multiple converters would otherwise apply.
 *
 * <p>Note that the target type {@code X} and the converted basic type
 * {@code Y} may be the same Java type.
 *
 * @param <X> the target type, that is, the type of the entity attribute
 * @param <Y> a basic type representing the type of the database column
 *
 * @see Converter
 * @see Convert#converter
 */
public interface AttributeConverter<X,Y> {

    /**
     * Converts the value stored in the entity attribute into the
     * data representation to be stored in the database.
     *
     * @param attribute  the entity attribute value to be converted
     * @return  the converted data to be stored in the database column
     */
    Y convertToDatabaseColumn(X attribute);

    /**
     * Converts the data stored in the database column into the value
     * to be stored in the entity attribute.
     *
     * <p>Note that it is the responsibility of the converter writer
     * to specify the correct {@code dbData} type for the corresponding
     * column for use by the JDBC driver: i.e., persistence providers
     * are not expected to do such type conversion.
     *
     * @param dbData  the data from the database column to be converted
     * @return  the converted value to be stored in the entity attribute
     */
    X convertToEntityAttribute(Y dbData);
}
```

B.21. SchemaManager

```java
import java.util.Map;

/**
 * Allows programmatic {@linkplain #create schema creation},
 * {@linkplain #validate schema validation},
 * {@linkplain #truncate data cleanup}, and
 * {@linkplain #populate data loading}, and
 * {@linkplain #drop schema cleanup} for entities belonging
 * to a certain persistence unit.
 *
 * <p>Properties are inherited from the {@link EntityManagerFactory},
 * that is, they may be specified via {@code persistence.xml} or
 * {@link Persistence#createEntityManagerFactory(String, Map)}.
 *
 * @see EntityManagerFactory#getSchemaManager()
 *
 * @since 3.2
 */
public interface SchemaManager {
    /**
     * Create database objects mapped by entities belonging to the
     * persistence unit.
     *
     * <p>If a DDL operation fails, the behavior is undefined.
     * A provider may throw an exception, or it may ignore the problem
     * and continue.
     *
     * @param createSchemas if {@code true}, attempt to create schemas,
     *                      otherwise, assume the schemas already exist
     */
    void create(boolean createSchemas);

    /**
     * Drop database objects mapped by entities belonging to the
     * persistence unit, undoing the effects of the
     * {@linkplain #create(boolean) previous creation}.
     *
     * <p>If a DDL operation fails, the behavior is undefined.
     * A provider may throw an exception, or it may ignore the problem
     * and continue.
     *
     * @param dropSchemas if {@code true}, drop schemas,
     *                    otherwise, leave them be
     */
    void drop(boolean dropSchemas);

    /**
     * Validate that the database objects mapped by entities belonging
     * to the persistence unit have the expected definitions.
     *
     * <p>The persistence provider is not required to perform
     * any specific validation, so the semantics of this operation are
     * entirely provider-specific.
     *
     * @throws SchemaValidationException if a database object is missing or
     * does not have the expected definition
     */
    void validate() throws SchemaValidationException;

    /**
     * Truncate the database tables mapped by entities belonging to
     * the persistence unit, and then re-import initial data from any
     * configured SQL scripts for data loading.
     *
```

```
     * <p>If a SQL operation fails, the behavior is undefined.
     * A provider may throw an exception, or it may ignore the problem
     * and continue.
     */
    void truncate();

    /**
     * Import initial data using any configured
     * {@linkplain PersistenceConfiguration#SCHEMAGEN_LOAD_SCRIPT_SOURCE
     * SQL scripts for data loading}, without executing any DDL.
     *
     * <p>If a SQL operation fails, the behavior is undefined.
     * A provider may throw an exception, or it may ignore the problem
     * and continue.
     *
     * @see PersistenceConfiguration#SCHEMAGEN_LOAD_SCRIPT_SOURCE
     *
     * @since 4.0
     */
    void populate();
}
```

B.22. Persistence

```
//     Linda DeMichiel - 2.0

package jakarta.persistence;

import java.util.List;
import java.util.Map;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.spi.PersistenceProvider;

import static jakarta.persistence.spi.PersistenceProviderResolverHolder.getPersistenceProviderResolver;

/**
 * Bootstrap class used to obtain an {@link EntityManagerFactory} in
 * Java SE environments.
 * {@snippet :
 * EntityManagerFactory factory =
 *         Persistence.createEntityManagerFactory("Library", // name of the persistence unit
 *                 // property overrides
 *                 Map.of(Persistence.ConnectionProperties.JDBC_USER, userName,
 *                        Persistence.ConnectionProperties.JDBC_PASSWORD, password));
 * }
 * Since {@link #createEntityManagerFactory} is an expensive operation,
 * and since the {@code EntityManagerFactory} is a heavyweight object,
 * the application program should create no more than one factory for
 * each persistence unit.
 *
 * <p>This class also allows schema management to be decoupled from
 * creation of the entity manager factory.
 * {@snippet :
 * Persistence.generateSchema("Library", // name of the persistence unit
 *         // property overrides
 *         Map.of(Persistence.SchemaManagementProperties.SCHEMAGEN_DATABASE_ACTION, "create",
 *                Persistence.ConnectionProperties.JDBC_USER, userName,
 *                Persistence.ConnectionProperties.JDBC_PASSWORD, password));
 * }
 *
 * <p>The {@code Persistence} class may be used to obtain a
 * {@link PersistenceUtil PersistenceUtil} instance in both Jakarta EE
 * and Java SE environments.
```

```
 * {@snippet :
 * boolean loaded = Persistence.getPersistenceUtil().isLoaded(entity);
 * }
 *
 * <p>The nested interfaces of this class list the standard configuration
 * properties defined by the Jakarta Persistence specification.
 *
 * @apiNote The {@code Persistence} class is available in a Jakarta EE
 * container environment. However, support for the Java SE bootstrapping
 * APIs is not required in container environments.
 *
 * @see PersistenceConfiguration
 * @see PersistenceProvider
 *
 * @since 1.0
 */
public final class Persistence {

    private Persistence() {
    }

    private static List<PersistenceProvider> getPersistenceProviders() {
        return getPersistenceProviderResolver().getPersistenceProviders();
    }

    /**
     * Create and return an {@link EntityManagerFactory} for the named
     * persistence unit.
     *
     * @param unitName the name of the persistence unit; usually the
     *                 name specified by the {@code name} attribute of
     *                 the {@code <persistence-unit>} element in the
     *                 {@code persistence.xml} file
     * @return an {@link EntityManagerFactory} for the named persistence
     *         unit
     * @apiNote This operation is very expensive. It should usually be
     *          called just once for each persistence unit.
     */
    @Nonnull
    public static EntityManagerFactory createEntityManagerFactory(@Nonnull String unitName) {
        return createEntityManagerFactory(unitName, null);
    }

    /**
     * Create and return an {@link EntityManagerFactory} for the named
     * persistence unit, using the given properties. Standard properties
     * are enumerated by {@link UnitProperties}, {@link CacheProperties},
     * {@link ConnectionProperties}, {@link JdbcProperties},
     * {@link DatabaseProperties}, {@link SchemaManagementProperties},
     * {@link ValidationProperties}, and {@link BeanManagementProperties}.
     * A provider might recognize additional vendor-specific properties.
     * Property values specified via the second argument of this method
     * override property values specified in the {@code persistence.xml}
     * file.
     *
     * @param unitName the name of the persistence unit; usually the name
     *                 specified by the {@code name} attribute of the
     *                 {@code <persistence-unit>} element in the
     *                 {@code persistence.xml} file
     * @param properties additional properties to use when creating the
     *                   factory. These properties may include properties
     *                   to control schema generation. The values of these
     *                   properties override any values that may have been
     *                   configured elsewhere.
     * @return an {@link EntityManagerFactory} for the named persistence
     *         unit
     * @apiNote This operation is very expensive. It should usually be
```

```
     *          called just once for each persistence unit.
     */
    @Nonnull
    public static EntityManagerFactory createEntityManagerFactory(@Nonnull String unitName,
                                                                  @Nullable Map<?,?> properties) {
        for (var provider : getPersistenceProviders()) {
            var entityManagerFactory = provider.createEntityManagerFactory(unitName, properties);
            if (entityManagerFactory != null) {
                return entityManagerFactory;
            }
        }

        throw new PersistenceException("No Persistence provider for EntityManager named " + unitName);
    }

    /**
     * Create and return an {@link EntityManagerFactory} for the named
     * persistence unit, using the given properties.
     *
     * @param configuration configuration of the persistence unit
     * @return the factory that creates {@link EntityManager}s configured
     *         according to the specified persistence unit
     * @apiNote This operation is very expensive. It should usually be
     *          called just once for each persistence unit.
     *
     * @since 3.2
     */
    @Nonnull
    public static EntityManagerFactory createEntityManagerFactory(@Nonnull PersistenceConfiguration configuration)
{
        for (var provider : getPersistenceProviders()) {
            var entityManagerFactory = provider.createEntityManagerFactory(configuration);
            if (entityManagerFactory != null) {
                return entityManagerFactory;
            }
        }

        throw new PersistenceException("No Persistence provider for EntityManager named " + configuration.name());
    }

    /**
     * Create database schemas and/or tables and/or create DDL scripts
     * as determined by the supplied properties. Standard properties
     * are enumerated by {@link SchemaManagementProperties},
     * {@link DatabaseProperties}, {@link ConnectionProperties}, and
     * {@link UnitProperties}. A provider might recognize additional
     * vendor-specific properties. Property values specified via the
     * second argument of this method override property values
     * specified in the {@code persistence.xml} file.
     * <p>
     * Called when schema generation is to occur as a separate phase
     * from creation of the entity manager factory.
     *
     * @param unitName the name of the persistence unit; usually the
     *                 name specified by the {@code name} attribute
     *                 of the {@code <persistence-unit>} element in
     *                 the {@code persistence.xml} file
     * @param map properties for schema generation; these may also
     *            contain provider-specific properties. The values
     *            of these properties override any values that may
     *            have been configured elsewhere.
     * @throws PersistenceException if insufficient or inconsistent
     *         configuration information is provided or if schema
     *         generation otherwise fails.
     *
     * @since 2.1
     */
```

```
public static void generateSchema(@Nonnull String unitName,
                                  @Nullable Map<?,?> map) {
    for (var provider : getPersistenceProviders()) {
        if (provider.generateSchema(unitName, map)) {
            return;
        }
    }

    throw new PersistenceException("No Persistence provider to generate schema named " + unitName);
}

/**
 * Create database schemas and/or tables and/or create DDL scripts
 * as determined by the supplied properties.
 * <p>
 * Called when schema generation is to occur as a separate phase
 * from creation of the entity manager factory.
 *
 * @param configuration configuration of the persistence unit
 * @throws PersistenceException if insufficient or inconsistent
 *         configuration information is provided or if schema
 *         generation otherwise fails.
 *
 * @since 4.0
 */
public static void generateSchema(@Nonnull PersistenceConfiguration configuration) {
    for (var provider : getPersistenceProviders()) {
        if (provider.generateSchema(configuration)) {
            return;
        }
    }

    throw new PersistenceException("No Persistence provider to generate schema named " + configuration.name());
}

/**
 * Return the {@link PersistenceUtil} instance
 * @return {@link PersistenceUtil} instance
 * @since 2.0
 */
@Nonnull
public static PersistenceUtil getPersistenceUtil() {
   return new PersistenceUtilImpl();
}


/**
 * Implementation of the {@link PersistenceUtil} interface
 * @since 2.0
 */
private static class PersistenceUtilImpl implements PersistenceUtil {
    public boolean isLoaded(Object entity, String attributeName) {
        var providers = getPersistenceProviders();

        for (var provider : providers) {
            switch (provider.getProviderUtil().isLoadedWithoutReference(entity, attributeName)) {
                case LOADED -> {
                    return true;
                }
                case NOT_LOADED -> {
                    return false;
                }
            }
        }

        //None of the providers could determine the load state try isLoadedWithReference
        for (var provider : providers) {
```

```
            switch (provider.getProviderUtil().isLoadedWithReference(entity, attributeName)) {
                case LOADED -> {
                    return true;
                }
                case NOT_LOADED -> {
                    return false;
                }
            }
        }

        //None of the providers could determine the load state.
        return true;
    }

    public boolean isLoaded(Object entity) {
        for (var provider : getPersistenceProviders()) {
            switch (provider.getProviderUtil().isLoaded(entity)) {
                case LOADED -> {
                    return true;
                }
                case NOT_LOADED -> {
                    return false;
                }
            }
        }
        //None of the providers could determine the load state
        return true;
    }
}

/**
 * Properties used to override elements the configuration of a persistence unit
 * in {@code persistence.xml}. Clients of the {@link PersistenceConfiguration}
 * class should not use these properties.
 * @since 4.0
 */
public interface UnitProperties {
    /**
     * The name of a class implementing {@link PersistenceProvider}.
     */
    String PERSISTENCE_PROVIDER = "jakarta.persistence.provider";

    /**
     * String specifying a {@link PersistenceUnitTransactionType}.
     */
    String PERSISTENCE_UNIT_TRANSACTION_TYPE = "jakarta.persistence.transactionType";

    /**
     * The JNDI name of a JTA data source used by the persistence provider to obtain database
     * connections.
     */
    String PERSISTENCE_UNIT_JTA_DATASOURCE = "jakarta.persistence.jtaDataSource";

    /**
     * The JNDI name of a non-JTA data source used by the persistence provider to obtain database
     * connections.
     */
    String PERSISTENCE_UNIT_NON_JTA_DATASOURCE = "jakarta.persistence.nonJtaDataSource";

    /**
     * The fully-qualified class name of an annotation annotated {@code jakarta.inject.Qualifier}.
     * This qualifier annotation may be used to disambiguate the persistence unit for the
     * purposes of dependency injection.
     */
    String PERSISTENCE_UNIT_BEAN_QUALIFIERS = "jakarta.persistence.qualifiers";

    /**
```

```
     * The fully-qualified class name of an annotation annotated {@code jakarta.inject.Scope}
     * or {@code jakarta.enterprise.context.NormalScope}. This scope annotation may be used to
     * determine the scope of a persistence context for the purposes of dependency injection.
     */
    String PERSISTENCE_UNIT_BEAN_SCOPE = "jakarta.persistence.scope";
}

/**
 * Properties used to configure the second-level cache.
 * @since 4.0
 */
public interface CacheProperties {
    /**
     * String specifying a {@link SharedCacheMode}.
     *
     * <p>Defined for use with
     * {@link Persistence#createEntityManagerFactory(String, Map)}.
     * Clients of the {@link PersistenceConfiguration} class
     * should use {@link PersistenceConfiguration#sharedCacheMode(SharedCacheMode)}.
     */
    String CACHE_MODE = "jakarta.persistence.sharedCache.mode";
}

/**
 * Properties used to connect to the database via JDBC in Java SE.
 * @since 4.0
 */
public interface ConnectionProperties {
    /**
     * Fully qualified name of the JDBC driver class.
     */
    String JDBC_DRIVER = "jakarta.persistence.jdbc.driver";

    /**
     * JDBC URL.
     */
    String JDBC_URL = "jakarta.persistence.jdbc.url";

    /**
     * Username for JDBC authentication.
     */
    String JDBC_USER = "jakarta.persistence.jdbc.user";

    /**
     * Password for JDBC authentication.
     */
    String JDBC_PASSWORD = "jakarta.persistence.jdbc.password";

    /**
     * An instance of {@code javax.sql.DataSource}.
     */
    String JDBC_DATASOURCE = "jakarta.persistence.dataSource";
}

/**
 * Properties used to optimize the interaction with JDBC.
 * @since 4.0
 */
public interface JdbcProperties {
    /**
     * Override the default {@linkplain java.sql.Statement#setFetchSize JDBC fetch size}.
     * @since 4.0
     */
    String JDBC_FETCH_SIZE = "jakarta.persistence.jdbc.fetchSize";

    /**
     * Enable {@linkplain java.sql.Statement#executeBatch JDBC statement batching}
```

```
     * by setting a batch size.
     * <p>This setting is a hint.
     * @since 4.0
     */
    String JDBC_BATCH_SIZE = "jakarta.persistence.jdbc.batchSize";

    /**
     * Default pessimistic lock timeout hint.
     */
    String LOCK_TIMEOUT = "jakarta.persistence.lock.timeout";

    /**
     * Default {@linkplain java.sql.Statement#setQueryTimeout query timeout} hint.
     */
    String QUERY_TIMEOUT = "jakarta.persistence.query.timeout";
}

/**
 * Properties used to specify the database platform when JDBC metadata is not available.
 * @since 4.0
 */
public interface DatabaseProperties {
    /**
     * The value returned by {@link java.sql.DatabaseMetaData#getDatabaseProductName()},
     * for use when JDBC metadata is not available.
     * @since 4.0
     */
    String DATABASE_PRODUCT_NAME = "jakarta.persistence.database-product-name";

    /**
     * The value returned by {@link java.sql.DatabaseMetaData#getDatabaseMajorVersion()},
     * for use when JDBC metadata is not available.
     * @since 4.0
     */
    String DATABASE_MAJOR_VERSION = "jakarta.persistence.database-major-version";

    /**
     * The value returned by {@link java.sql.DatabaseMetaData#getDatabaseMinorVersion()},
     * for use when JDBC metadata is not available.
     * @since 4.0
     */
    String DATABASE_MINOR_VERSION = "jakarta.persistence.database-minor-version";
}

/**
 * Properties used to control the schema management tooling.
 * @since 4.0
 */
public interface SchemaManagementProperties {
    /**
     * The action to be performed against the database.
     *
     * <p>Standard actions are: {@code none}, {@code create},
     * {@code drop}, {@code drop-and-create}, {@code validate},
     * {@code populate}.
     *
     * @see SchemaManagementAction
     * @see PersistenceConfiguration#schemaManagementDatabaseAction(SchemaManagementAction)
     */
    String SCHEMAGEN_DATABASE_ACTION = "jakarta.persistence.schema-generation.database.action";

    /**
     * The action to be generated as a SQL script.
     *
     * <p>The script is generated in the location specified by
     * {@value #SCHEMAGEN_CREATE_TARGET} or {@value #SCHEMAGEN_DROP_TARGET}.
     *
```

```
 * <p>Standard actions are: {@code none}, {@code create},
 * {@code drop}, {@code drop-and-create}.
 *
 * @see SchemaManagementAction
 * @see PersistenceConfiguration#schemaManagementScriptsAction(SchemaManagementAction)
 */
String SCHEMAGEN_SCRIPTS_ACTION = "jakarta.persistence.schema-generation.scripts.action";

/**
 * The source of artifacts to be created.
 *
 * <p>Standard sources are: {@code metadata}, {@code script},
 * {@code metadata-then-script}, {@code script-then-metadata}.
 *
 * <p>The location of the script source is specified by
 * {@value #SCHEMAGEN_CREATE_SCRIPT_SOURCE}.
 */
String SCHEMAGEN_CREATE_SOURCE = "jakarta.persistence.schema-generation.create-source";

/**
 * The source of artifacts to be dropped.
 *
 * <p>Standard sources are: {@code metadata}, {@code script},
 * {@code metadata-then-script}, {@code script-then-metadata}.
 *
 * <p>The location of the script source is specified by
 * {@value #SCHEMAGEN_DROP_SCRIPT_SOURCE}.
 */
String SCHEMAGEN_DROP_SOURCE = "jakarta.persistence.schema-generation.drop-source";

/**
 * An application-provided SQL script to be executed when the
 * schema is created.
 * <p>
 * An instance of {@link java.io.Reader} or a string specifying
 * the file URL of the DDL script.
 */
String SCHEMAGEN_CREATE_SCRIPT_SOURCE = "jakarta.persistence.schema-generation.create-script-source";

/**
 * An application-provided SQL script to be executed when the
 * schema is dropped.
 * <p>
 * An instance of {@link java.io.Reader} or a string specifying
 * the file URL of the DDL script.
 */
String SCHEMAGEN_DROP_SCRIPT_SOURCE = "jakarta.persistence.schema-generation.drop-script-source";

/**
 * An application-provided SQL script to be executed after the
 * schema is created, typically used for loading data.
 * <p>
 * An instance of {@link java.io.Reader} or a string specifying
 * the file URL of the DML script.
 */
String SCHEMAGEN_LOAD_SCRIPT_SOURCE = "jakarta.persistence.sql-load-script-source";

/**
 * The provider-generated SQL script which creates the schema
 * when {@value SCHEMAGEN_SCRIPTS_ACTION} is set.
 * <p>
 * An instance of {@link java.io.Writer} or a string specifying
 * the file URL of the DDL script.
 */
String SCHEMAGEN_CREATE_TARGET = "jakarta.persistence.schema-generation.scripts.create-target";

/**
```

```
     * The provider-generated SQL script which drops the schema
     * when {@value SCHEMAGEN_SCRIPTS_ACTION} is set.
     * <p>
     * An instance of {@link java.io.Writer} or a string specifying
     * the file URL of the DDL script.
     */
    String SCHEMAGEN_DROP_TARGET = "jakarta.persistence.schema-generation.scripts.drop-target";

    /**
     * Specifies whether the persistence provider creates database schemas
     * in addition to creating database objects such as tables, sequences,
     * constraints, and so on. The value {@code true} specifies that the
     * persistence provider creates schemas in the database or generates
     * DDL containing {@code CREATE SCHEMA} commands.
     * <p>
     * If this property is not supplied, the provider does not create database
     * schemas.
     * @since 4.0
     */
    String SCHEMAGEN_CREATE_SCHEMAS = "jakarta.persistence.schema-generation.create-database-schemas";

    /**
     * Supplies a JDBC {@code Connection} to use for executing schema
     * management {@linkplain #SCHEMAGEN_DATABASE_ACTION actions} on the
     * database. If not specified, the persistence provider uses the
     * {@code DataSource} configured for use by the persistence unit to
     * obtain a JDBC connection.
     * @since 4.0
     */
    String SCHEMAGEN_CONNECTION = "jakarta.persistence.schema-generation.connection";
}

/**
 * Properties used to control the integration with Bean Validation.
 * @since 4.0
 */
public interface ValidationProperties {
    /**
     * An instance of {@code jakarta.validation.ValidatorFactory}.
     */
    String VALIDATION_FACTORY = "jakarta.persistence.validation.factory";

    /**
     * Target groups for validation at {@link PrePersist}.
     */
    String VALIDATION_GROUP_PRE_PERSIST = "jakarta.persistence.validation.group.pre-persist";

    /**
     * Target groups for validation at {@link PreMerge}.
     *
     * @since 4.0
     */
    String VALIDATION_GROUP_PRE_MERGE = "jakarta.persistence.validation.group.pre-merge";

    /**
     * Target groups for validation at {@link PreRemove}.
     */
    String VALIDATION_GROUP_PRE_REMOVE = "jakarta.persistence.validation.group.pre-remove";

    /**
     * Target groups for validation at {@link PreInsert}.
     *
     * @since 4.0
     */
    String VALIDATION_GROUP_PRE_INSERT = "jakarta.persistence.validation.group.pre-insert";

    /**
```

```
         * Target groups for validation at {@link PreUpdate}.
         */
        String VALIDATION_GROUP_PRE_UPDATE = "jakarta.persistence.validation.group.pre-update";

        /**
         * Target groups for validation at {@link PreUpsert}.
         *
         * @since 4.0
         */
        String VALIDATION_GROUP_PRE_UPSERT = "jakarta.persistence.validation.group.pre-upsert";

        /**
         * Target groups for validation at {@link PreDelete}.
         *
         * @since 4.0
         */
        String VALIDATION_GROUP_PRE_DELETE = "jakarta.persistence.validation.group.pre-delete";

        /**
         * String specifying a {@link ValidationMode}.
         *
         * <p>Defined for use with
         * {@link Persistence#createEntityManagerFactory(String, Map)}.
         * Clients of the {@link PersistenceConfiguration} class
         * should use {@link PersistenceConfiguration#validationMode(ValidationMode)}.
         */
        String VALIDATION_MODE = "jakarta.persistence.validation.mode";
    }

    /**
     * Properties used to control the integration with CDI.
     * @since 4.0
     */
    public interface BeanManagementProperties {
        /**
         * An instance of {@code jakarta.enterprise.inject.spi.BeanManager}.
         */
        String BEAN_MANAGER = "jakarta.persistence.bean.manager";
    }
}
```

B.23. PersistenceConfiguration

```
package jakarta.persistence;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Objects;

import static java.util.Objects.requireNonNull;

/**
 * Represents a configuration of a persistence unit, allowing programmatic
 * creation of an {@link EntityManagerFactory}. The configuration options
 * available via this API reflect the similarly named elements of the
 * {@code persistence.xml} file.
 *
 * <p>This API may not be used to configure a container-managed persistence
 * unit. That is, the configured persistence unit should be considered a
 * Java SE persistence unit, even when this API is used within the Jakarta
```

```
 * EE environment.
 *
 * <p>If injection of the {@link EntityManagerFactory} is required, a CDI
 * {@code Producer} may be used to make the {@link EntityManagerFactory}
 * available as a CDI managed bean.
 * {@snippet :
 * @Produces @ApplicationScoped @Documents
 * EntityManagerFactory configure() {
 *     return new PersistenceConfiguration("DocumentData")
 *             .nonJtaDataSource("java:global/jdbc/DocumentDatabase")
 *             .defaultToOneFetchType(FetchType.LAZY)
 *             .managedClass(Document.class)
 *             .createEntityManagerFactory();
 * }
 * }
 *
 * <p>Similarly, if injection of an {@link EntityManager} is required,
 * a CDI {@code Producer} method/{@code Disposer} method pair may be
 * used to make the {@link EntityManager} available as a CDI managed
 * bean.
 * {@snippet :
 * @Produces @TransactionScoped @Documents
 * EntityManager create(@Documents EntityManagerFactory factory) {
 *     return factory.createEntityManager();
 * }
 *
 * void close(@Disposes @Documents EntityManager entityManager) {
 *     entityManager.close();
 * }
 * }
 *
 * <p>It is intended that persistence providers define subclasses of
 * this class with vendor-specific configuration options. A provider
 * must support configuration via any instance of this class or of any
 * subclass of this class.
 *
 * @see Persistence#createEntityManagerFactory(PersistenceConfiguration)
 *
 * @since 3.2
 */
public class PersistenceConfiguration
        implements Persistence.JdbcProperties,
                   Persistence.ConnectionProperties,
                   Persistence.DatabaseProperties,
                   Persistence.SchemaManagementProperties,
                   Persistence.ValidationProperties,
                   Persistence.CacheProperties,
                   Persistence.BeanManagementProperties {

    private final String name;

    private String provider;
    private String jtaDataSource;
    private String nonJtaDataSource;

    private SharedCacheMode sharedCacheMode = SharedCacheMode.UNSPECIFIED;
    private ValidationMode validationMode = ValidationMode.AUTO;
    private PersistenceUnitTransactionType transactionType = PersistenceUnitTransactionType.RESOURCE_LOCAL;
    private FetchType defaultToOneFetchType = FetchType.EAGER;

    private SchemaManagementAction schemaManagementDatabaseAction = SchemaManagementAction.NONE;
    private SchemaManagementAction schemaManagementScriptsAction = SchemaManagementAction.NONE;

    private final List<Class<?>> managedClasses = new ArrayList<>();
    private final List<String> mappingFileNames = new ArrayList<>();
    private final Map<String,Object> properties = new HashMap<>();
```

```java
/**
 * Create a new empty configuration. An empty configuration does not
 * typically hold enough information for successful invocation of
 * {@link #createEntityManagerFactory()}.
 *
 * @param name the name of the persistence unit, which may be used by
 *             the persistence provider for logging and error reporting
 */
public PersistenceConfiguration(@Nonnull String name) {
    requireNonNull(name, "Persistence unit name should not be null");
    this.name = name;
}

/**
 * Create a new {@link EntityManagerFactory} based on this configuration.
 * @throws PersistenceException if required configuration is missing or
 *                              if the factory could not be created
 * @apiNote This operation is very expensive. It should usually be called
 *          just once for each persistence unit.
 */
@Nonnull
public EntityManagerFactory createEntityManagerFactory() {
    return Persistence.createEntityManagerFactory(this);
}

/**
 * Execute the schema management action specified by the property
 * {@link #SCHEMAGEN_DATABASE_ACTION} or {@link #SCHEMAGEN_SCRIPTS_ACTION},
 * or by {@link #schemaManagementDatabaseAction(SchemaManagementAction)}
 * or {@link #schemaManagementScriptsAction(SchemaManagementAction)}
 * without creating an {@link EntityManagerFactory}.
 *
 * @throws PersistenceException if insufficient or inconsistent
 *         configuration information is provided or if schema
 *         generation otherwise fails.
 *
 * @since 4.0
 */
public void exportSchema() {
    Persistence.generateSchema( this );
}

/**
 * The name of the persistence unit, which may be used by the persistence
 * provider for logging and error reporting.
 * @return the name of the persistence unit.
 */
@Nonnull
public String name() {
    return name;
}

/**
 * Specify the persistence provider.
 * @param providerClassName the qualified name of the persistence provider class
 * @return this configuration
 */
@Nonnull
public PersistenceConfiguration provider(@Nullable String providerClassName) {
    this.provider = providerClassName;
    return this;
}

/**
 * The fully-qualified name of a concrete class implementing
 * {@link jakarta.persistence.spi.PersistenceProvider}.
 * @return the qualified name of the persistence provider class.
```

```
 */
@Nullable
public String provider() {
    return provider;
}

/**
 * Specify the JNDI name of a JTA {@code javax.sql.DataSource}.
 * @param dataSourceJndiName the JNDI name of a JTA datasource
 * @return this configuration
 */
@Nonnull
public PersistenceConfiguration jtaDataSource(@Nullable String dataSourceJndiName) {
    this.jtaDataSource = dataSourceJndiName;
    return this;
}

/**
 * The JNDI name of a JTA {@code javax.sql.DataSource}.
 * @return the configured JTA datasource, if any, or null
 */
@Nullable
public String jtaDataSource() {
    return jtaDataSource;
}

/**
 * Specify the JNDI name of a non-JTA {@code javax.sql.DataSource}.
 * @param dataSourceJndiName the JNDI name of a non-JTA datasource
 * @return this configuration
 */
@Nonnull
public PersistenceConfiguration nonJtaDataSource(@Nullable String dataSourceJndiName) {
    this.nonJtaDataSource = dataSourceJndiName;
    return this;
}

/**
 * The JNDI name of a non-JTA {@code javax.sql.DataSource}.
 * @return the configured non-JTA datasource, if any, or null
 */
@Nullable
public String nonJtaDataSource() {
    return nonJtaDataSource;
}

/**
 * Add a managed class (an {@link Entity}, {@link Embeddable},
 * {@link MappedSuperclass}, or {@link Converter}) to the
 * configuration.
 * @param managedClass the managed class
 * @return this configuration
 */
@Nonnull
public PersistenceConfiguration managedClass(@Nonnull Class<?> managedClass) {
    requireNonNull(managedClass, "managedClass cannot be null");
    managedClasses.add(managedClass);
    return this;
}

/**
 * The configured managed classes, that is, a list of classes
 * annotated {@link Entity}, {@link Embeddable},
 * {@link MappedSuperclass}, or {@link Converter}.
 * @return all configured managed classes
 */
@Nonnull
```

```java
public List<Class<?>> managedClasses() {
    return managedClasses;
}

/**
 * Add the path of an XML mapping file loaded as a resource to
 * the configuration.
 * @param name the resource path of the mapping file
 * @return this configuration
 */
@Nonnull
public PersistenceConfiguration mappingFile(@Nonnull String name) {
    requireNonNull(name, "name cannot be null");
    mappingFileNames.add(name);
    return this;
}

/**
 * The configured resource paths of XML mapping files.
 * @return all configured mapping file resource paths
 */
@Nonnull
public List<String> mappingFiles() {
    return mappingFileNames;
}

/**
 * Specify the transaction type for the persistence unit.
 * @param transactionType the transaction type
 * @return this configuration
 */
@Nonnull
public PersistenceConfiguration transactionType(@Nonnull PersistenceUnitTransactionType transactionType) {
    requireNonNull(transactionType, "transactionType cannot be null");
    this.transactionType = transactionType;
    return this;
}

/**
 * The {@linkplain PersistenceUnitTransactionType transaction type}.
 * <ul>
 * <li>If {@link PersistenceUnitTransactionType#JTA}, a JTA data
 *    source must be provided via {@link #jtaDataSource()},
 *    or by the container.
 * <li>If {@link PersistenceUnitTransactionType#RESOURCE_LOCAL},
 *    database connection properties may be specified via
 *    {@link #properties()}, or a non-JTA datasource may be
 *    provided via {@link #nonJtaDataSource()}.
 * </ul>
 * @return the transaction type
 */
@Nonnull
public PersistenceUnitTransactionType transactionType() {
    return transactionType;
}

/**
 * Specify the shared cache mode for the persistence unit.
 * @param sharedCacheMode the shared cache mode
 * @return this configuration
 */
@Nonnull
public PersistenceConfiguration sharedCacheMode(@Nonnull SharedCacheMode sharedCacheMode) {
    requireNonNull(sharedCacheMode, "sharedCacheMode cannot be null");
    this.sharedCacheMode = sharedCacheMode;
    return this;
}
```

```java
/**
 * The shared cache mode. The default behavior is unspecified
 * and {@linkplain SharedCacheMode#UNSPECIFIED provider-specific}.
 * @return the shared cache mode
 */
@Nonnull
public SharedCacheMode sharedCacheMode() {
    return sharedCacheMode;
}

/**
 * Specify the validation mode for the persistence unit.
 * @param validationMode the shared cache mode
 * @return this configuration
 */
@Nonnull
public PersistenceConfiguration validationMode(@Nonnull ValidationMode validationMode) {
    requireNonNull(validationMode, "validationMode cannot be null");
    this.validationMode = validationMode;
    return this;
}

/**
 * The validation mode, {@link ValidationMode#AUTO} by default.
 * @return the validation mode
 */
@Nonnull
public ValidationMode validationMode() {
    return validationMode;
}

/**
 * Specify the {@linkplain FetchType#DEFAULT default fetch type}
 * for one-to-one and many-to-one associations. The given fetch type
 * value must be {@link FetchType#EAGER} or {@link FetchType#LAZY}.
 * @param defaultToOneFetchType the default fetch type
 * @return this configuration
 * @since 4.0
 */
@Nonnull
public PersistenceConfiguration defaultToOneFetchType(@Nonnull FetchType defaultToOneFetchType) {
    requireNonNull(defaultToOneFetchType, "defaultToOneFetchType cannot be null");
    if (defaultToOneFetchType == FetchType.DEFAULT) {
        throw new IllegalArgumentException("defaultToOneFetchType: " + defaultToOneFetchType);
    }
    this.defaultToOneFetchType = defaultToOneFetchType;
    return this;
}

/**
 * The {@linkplain FetchType#DEFAULT default fetch type},
 * {@link FetchType#EAGER} by default.
 * @since 4.0
 */
@Nonnull
public FetchType defaultToOneFetchType() {
    return defaultToOneFetchType;
}

/**
 * Set the schema management action to be performed against the database.
 *
 * @param action The schema management action to be performed
 * @see #SCHEMAGEN_DATABASE_ACTION
 * @since 4.0
 */
```

```java
@Nonnull
public PersistenceConfiguration schemaManagementDatabaseAction(@Nonnull SchemaManagementAction action) {
    requireNonNull(action, "action cannot be null");
    this.schemaManagementDatabaseAction = action;
    return this;
}

/**
 * The schema management action to be performed against the database.
 *
 * @return The schema management action
 * @see #SCHEMAGEN_DATABASE_ACTION
 * @since 4.0
 */
@Nonnull
public SchemaManagementAction schemaManagementDatabaseAction() {
    return schemaManagementDatabaseAction;
}

/**
 * Set the schema management action to be performed by generated DDL scripts.
 *
 * @param action The schema management action to be performed
 * @see #SCHEMAGEN_SCRIPTS_ACTION
 * @since 4.0
 */
@Nonnull
public PersistenceConfiguration schemaManagementScriptsAction(@Nonnull SchemaManagementAction action) {
    requireNonNull(action, "action cannot be null");
    this.schemaManagementScriptsAction = action;
    return this;
}

/**
 * The schema management action to be performed by generated DDL scripts.
 *
 * @return The schema management action
 * @see #SCHEMAGEN_SCRIPTS_ACTION
 * @since 4.0
 */
@Nonnull
public SchemaManagementAction schemaManagementScriptsAction() {
    return schemaManagementScriptsAction;
}

/**
 * Set a property of this persistence unit.
 * @param name the property name
 * @param value the property value
 * @return this configuration
 * @see Persistence.JdbcProperties
 * @see Persistence.ConnectionProperties
 * @see Persistence.DatabaseProperties
 * @see Persistence.SchemaManagementProperties
 * @see Persistence.ValidationProperties
 * @see Persistence.CacheProperties
 * @see Persistence.BeanManagementProperties
 */
@Nonnull
public PersistenceConfiguration property(@Nonnull String name,
                                         @Nullable Object value) {
    properties.put(name, value);
    return this;
}

/**
 * Set multiple properties of this persistence unit.
```

```java
     * @param properties the properties
     * @return this configuration
     */
    @Nonnull
    public PersistenceConfiguration properties(@Nonnull Map<String,?> properties) {
        this.properties.putAll(properties);
        return this;
    }

    /**
     * Standard and vendor-specific property settings.
     * @return the configured properties
     */
    @Nonnull
    public Map<String, Object> properties() {
        return properties;
    }
}
```

B.24. PersistenceUtil

```java
package jakarta.persistence;

import jakarta.annotation.Nonnull;

/**
 * Utility interface between the application and the persistence
 * provider(s).
 *
 * <p>The {@code PersistenceUtil} interface instance obtained from
 * the {@link Persistence} class is used to determine the load state
 * of an entity or entity attribute regardless of which persistence
 * provider in the environment created the entity.
 *
 * @since 2.0
 */
public interface PersistenceUtil {

    /**
     * Determine the load state of a given persistent attribute.
     * @param entity  entity containing the attribute
     * @param attributeName name of attribute whose load state is
     *        to be determined
     * @return false if entity's state has not been loaded or if
     * the attribute state has not been loaded, else true
     */
    boolean isLoaded(@Nonnull Object entity,
                     @Nonnull String attributeName);

    /**
     * Determine the load state of an entity.
     * This method can be used to determine the load state of an
     * entity passed as a reference. An entity is considered loaded
     * if all attributes for which {@link FetchType#EAGER} has been
     * specified have been loaded.
     * <p>The {@link #isLoaded(Object, String)} method should be
     * used to determine the load state of an attribute. Not doing
     * so might lead to unintended loading of state.
     * @param entity whose load state is to be determined
     * @return false if the entity has not been loaded, else true
     */
    boolean isLoaded(@Nonnull Object entity);
}
```

B.25. PersistenceUnitUtil

```java
package jakarta.persistence;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.Attribute;

/**
 * Utility interface between the application and the persistence
 * provider managing the persistence unit.
 *
 * <p>The methods of this interface should only be invoked on
 * entity instances obtained from or managed by entity managers
 * for this persistence unit or on new entity instances.
 *
 * @since 2.0
 */
public interface PersistenceUnitUtil extends PersistenceUtil {

    /**
     * Determine the load state of a given persistent attribute
     * of an entity belonging to the persistence unit.
     * @param entity  entity instance containing the attribute
     * @param attributeName name of attribute whose load state is
     *        to be determined
     * @return false if entity's state has not been loaded or if
     *         the attribute state has not been loaded, else true
     */
    boolean isLoaded(@Nonnull Object entity,
                     @Nonnull String attributeName);

    /**
     * Determine the load state of a given persistent attribute
     * of an entity belonging to the persistence unit.
     * @param entity  entity instance containing the attribute
     * @param attribute  attribute whose load state is to be determined
     * @param <E> the entity type
     * @return false if entity's state has not been loaded or if
     *         the attribute state has not been loaded, else true
     * @since 3.2
     */
    <E> boolean isLoaded(@Nonnull E entity,
                         @Nonnull Attribute<? super E, ?> attribute);

    /**
     * Determine the load state of an entity belonging to the
     * persistence unit. This method can be used to determine the
     * load state of an entity passed as a reference. An entity is
     * considered loaded if all attributes for which
     * {@link FetchType#EAGER} has been specified have been loaded.
     * <p> The {@link #isLoaded(Object, String)} method should be
     * used to determine the load state of an attribute. Not doing
     * so might lead to unintended loading of state.
     * @param entity   entity instance whose load state is to be determined
     * @return false if the entity has not been loaded, else true
     */
    boolean isLoaded(@Nonnull Object entity);

    /**
     * Load the persistent value of a given persistent attribute
     * of an entity belonging to the persistence unit and to an
     * open persistence context.
     * After this method returns, {@link #isLoaded(Object,String)}
     * must return true with the given entity instance and attribute.
     * @param entity  entity instance
```

```
 * @param attributeName  the name of the attribute to be loaded
 * @throws IllegalArgumentException if the given object is not an
 * instance of an entity class belonging to the persistence unit
 * @throws PersistenceException if the entity is not associated
 * with an open persistence context or cannot be loaded from the
 * database
 * @since 3.2
 */
void load(@Nonnull Object entity,
          @Nonnull String attributeName);

/**
 * Load the persistent value of a given persistent attribute
 * of an entity belonging to the persistence unit and to an
 * open persistence context.
 * After this method returns, {@link #isLoaded(Object,Attribute)}
 * must return true with the given entity instance and attribute.
 * @param entity  entity instance to be loaded
 * @param attribute  the attribute to be loaded
 * @param <E> the entity type
 * @throws IllegalArgumentException if the given object is not an
 * instance of an entity class belonging to the persistence unit
 * @throws PersistenceException if the entity is not associated
 * with an open persistence context or cannot be loaded from the
 * database
 * @since 3.2
 */
<E> void load(@Nonnull E entity,
              @Nonnull Attribute<? super E, ?> attribute);

/**
 * Load the persistent state of an entity belonging to the
 * persistence unit and to an open persistence context.
 * After this method returns, {@link #isLoaded(Object)} must
 * return true with the given entity instance.
 * @param entity  entity instance to be loaded
 * @throws IllegalArgumentException if the given object is not an
 * instance of an entity class belonging to the persistence unit
 * @throws PersistenceException if the entity is not associated
 * with an open persistence context or cannot be loaded from the
 * database
 * @since 3.2
 */
void load(@Nonnull Object entity);

/**
 * Return true if the given entity belonging to the persistence
 * unit and to an open persistence context is an instance of the
 * given entity class, or false otherwise. This method may, but
 * is not required to, load the given entity by side effect.
 * @param entity  entity instance
 * @param entityClass  an entity class belonging to the persistence
 * unit
 * @throws IllegalArgumentException if the given object is not an
 * instance of an entity class belonging to the persistence unit
 * or if the given class is not an entity class belonging to the
 * persistence unit
 * @throws PersistenceException if the entity is not associated
 * with an open persistence context or cannot be loaded from the
 * database
 * @since 3.2
 */
boolean isInstance(@Nonnull Object entity,
                   @Nonnull Class<?> entityClass);

/**
 * Return the concrete entity class if the given entity belonging
```

```
     * to the persistence unit and to an open persistence context.
     * This method may, but is not required to, load the given entity
     * by side effect.
     * @param entity  entity instance
     * @param <T> the entity type
     * @return an entity class belonging to the persistence unit
     * @throws IllegalArgumentException if the given object is not an
     * instance of an entity class belonging to the persistence unit
     * @throws PersistenceException if the entity is not associated
     * with an open persistence context or cannot be loaded from the
     * database
     * @since 3.2
     */
    @Nonnull
    <T> Class<? extends T> getClass(@Nonnull T entity);

    /**
     * Return the id of the entity.
     * A generated id is not guaranteed to be available until after
     * the database insert has occurred.
     * Returns null if the entity does not yet have an id.
     * @param entity  entity instance
     * @return id of the entity
     * @throws IllegalArgumentException if the object is found not
     *         to be an entity
     */
    @Nullable
    Object getIdentifier(@Nonnull Object entity);

    /**
     * Return the version of the entity.
     * A generated version is not guaranteed to be available until after
     * the database insert has occurred.
     * Returns null if the entity does not yet have a version.
     * @param entity  entity instance
     * @return id of the entity
     * @throws IllegalArgumentException if the object is found not
     *         to be an entity
     * @since 3.2
     */
    @Nullable
    Object getVersion(@Nonnull Object entity);
}
```

B.26. Reference

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

import java.util.List;
import java.util.Map;

/**
 * A reference to a named statement or query declared via the
 * {@link NamedQuery} or {@link NamedNativeQuery} annotations,
 * or using {@link jakarta.persistence.query.JakartaQuery} or
 * {@link jakarta.persistence.query.NativeQuery}. An
 * instance of {@code Reference} is usually obtained from the
 * static metamodel. This interface declares operations common
 * to {@link StatementReference} and {@link TypedQueryReference}.
 *
 * @since 4.0
 */
public sealed interface Reference
        permits StatementReference, TypedQueryReference {
```

```java
/**
 * The name of the statement or query, as specified by
 * {@link NamedQuery#name} or {@link NamedNativeQuery#name},
 * or as inferred from the name of the method annotated
 * {@link jakarta.persistence.query.JakartaQuery} or
 * {@link jakarta.persistence.query.NativeQuery}.
 */
@Nullable
String getName();

/**
 * The types of the supplied
 * {@linkplain #getArguments arguments} to parameters of the
 * statement or query, or {@code null} if no arguments were
 * supplied. Arguments are present when this is a reference
 * to a statement or query declared using an annotation of a
 * method.
 * <p>
 * Any mutation of the returned list results in an
 * {@link UnsupportedOperationException}.
 *
 * @since 4.0
 */
@Nullable
List<Class<?>> getParameterTypes();

/**
 * The names assigned to the supplied
 * {@linkplain #getArguments arguments} to parameters of the
 * statement or query, or {@code null} if no arguments were
 * supplied. Arguments are present when this is a reference
 * to a statement or query declared using an annotation of a
 * method. If the query has named parameters, these are
 * interpreted as the parameter names. Otherwise, if the
 * query has positional parameters, they are ignored.
 * <p>
 * Any mutation of the returned list results in an
 * {@link UnsupportedOperationException}.
 *
 * @since 4.0
 */
@Nullable
List<String> getParameterNames();

/**
 * The arguments supplied to the parameters of the statement
 * or query, or {@code null} if no arguments were supplied.
 * Arguments are present when this is a reference to a query
 * declared using an annotation of a method.
 * <ul>
 * <li>If the query has ordinal parameters, the position of
 *     an argument in this array determines its assignment
 *     to a parameter.
 * <li>If the query has named parameters, this array is
 *     aligned elementwise with the array of
 *     {@linkplain #getParameterNames parameter names} to
 *     obtain an assignment of arguments to parameters.
 * </ul>
 * <p>
 * Any mutation of the returned list results in an
 * {@link UnsupportedOperationException}.
 *
 * @see Query#setParameter(int, Object)
 * @see Query#setParameter(String, Object)
 * @since 4.0
 */
```

```
    @Nullable
    List<Object> getArguments();

    /**
     * A map keyed by hint name of all hints specified via
     * {@link NamedQuery#hints}, {@link NamedNativeQuery#hints},
     * or {@link jakarta.persistence.query.QueryOptions#hints}.
     * <p>
     * Any attempted mutation of the returned map results in an
     * {@link UnsupportedOperationException}.
     */
    @Nonnull
    Map<String,Object> getHints();
}
```

B.27. TypedQueryReference

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

import java.util.Set;

/**
 * A reference to a typed named query declared via the
 * {@link NamedQuery} or {@link NamedNativeQuery} annotations,
 * or using {@link jakarta.persistence.query.JakartaQuery} or
 * {@link jakarta.persistence.query.NativeQuery}. An
 * instance of {@code TypedQueryReference} is usually obtained
 * from the static metamodel of the annotated type.
 *
 * <p>In this example, a method is annotated, and the name of
 * the query is determined by the name of the annotated method.
 * The annotated query is executed with the given argument to
 * the query parameter named {@code pattern} and with the
 * {@link CacheStoreMode#BYPASS} option:
 * {@snippet :
 * class Library {
 *     @Inject EntityManager entityManager;
 *
 *     @JakartaQuery("select a from Book b join b.authors a where b.title like :pattern")
 *     @QueryOptions(cacheStoreMode = CacheStoreMode.BYPASS)
 *     List<Author> findAuthorsGivenTitles(String pattern) {
 *         return entityManager.createQuery(Library_.findAuthorsGivenTitles(pattern))
 *                 .getResultList();
 *     }
 * }
 * }
 *
 * <p>In this example, it is the entity class that is annotated,
 * and the {@link NamedQuery} annotation explicitly specifies a
 * name:
 * {@snippet :
 * @NamedQuery(name = "byTitle",
 *             query = "from Book where title like ?1")
 * @Entity class Book { .. }
 * }
 * <p>In this case the {@code TypedQueryReference} obtained from
 * its static metamodel does not include arguments to the query
 * parameters, and so they must be supplied via {@code setParameter}:
 * {@snippet :
 * var books =
 *         entityManager.createQuery(Book_._byTitle_)
 *                 .setCacheStoreMode(CacheStoreMode.BYPASS)
 *                 .setParameter(1, pattern)
 *                 .getResultList();
```

```
 * }
 *
 * <p>A {@code TypedQueryReference} may include arguments to
 * parameters of the query.
 * <ul>
 * <li>A reference representing a query declared using an
 *     annotation of a type or field has no arguments, and so
 *     {@link #getArguments()}, {@link #getParameterNames()},
 *     and {@link #getParameterTypes()} all return {@code null}.
 * <li>A reference representing a query declared using an
 *     annotation of a method holds information about the
 *     arguments passed to the method, making it available via
 *     {@link #getArguments()}, {@link #getParameterNames()},
 *     and {@link #getParameterTypes()}.
 * </ul>
 *
 * <p>In the Jakarta Persistence query language, only SELECT
 * queries are typed queries, since only a SELECT query can
 * return a result. A DELETE or UPDATE statement is not a
 * typed query and is always represented by an untyped
 * instance of {@link StatementReference}. On the other hand, a
 * native SQL query is considered a typed query if it returns
 * a result set.
 *
 * @param <R> an upper bound on the result type of the query
 *
 * @see EntityHandler#createQuery(TypedQueryReference)
 *
 * @since 3.2
 */
public non-sealed interface TypedQueryReference<R> extends Reference {

    /**
     * The result type of the query, as specified by
     * {@link NamedQuery#resultClass} or
     * {@link NamedNativeQuery#resultClass}, or as inferred
     * from the declared return type of the method annotated
     * {@link jakarta.persistence.query.JakartaQuery} or
     * {@link jakarta.persistence.query.NativeQuery}.
     */
    @Nonnull
    Class<? extends R> getResultType();

    /**
     * All {@linkplain TypedQuery.Option options} specified by
     * {@link jakarta.persistence.query.QueryOptions},
     * {@link NamedQuery}, or {@link NamedNativeQuery}, or an
     * empty set if no options were specified.
     *
     * @see TypedQuery#addOption
     * @see jakarta.persistence.query.QueryOptions#cacheStoreMode
     * @see jakarta.persistence.query.QueryOptions#cacheRetrieveMode
     * @see jakarta.persistence.query.QueryOptions#lockMode
     * @see jakarta.persistence.query.QueryOptions#lockScope
     * @see jakarta.persistence.query.QueryOptions#timeout
     * @see jakarta.persistence.query.QueryOptions#flush
     *
     * @since 4.0
     */
    @Nonnull
    Set<TypedQuery.Option> getOptions();

    /**
     * The specified entity graph name, if any,
     * or {@code null} if no entity graph was
     * specified.
     *
```

```
     * @see NamedQuery#entityGraph
     * @see jakarta.persistence.query.QueryOptions#entityGraph
     * @since 4.0
     */
    @Nullable
    String getEntityGraphName();
}
```

B.28. StatementReference

```
import jakarta.annotation.Nonnull;

import java.util.Set;

/**
 * A reference to an executable named statement for an operation
 * that returns a row count. The named statement is declared via the
 * {@link NamedQuery} or {@link NamedNativeQuery} annotations,
 * or using {@link jakarta.persistence.query.JakartaQuery} or
 * {@link jakarta.persistence.query.NativeQuery}. An
 * instance of {@code StatementReference} is usually obtained
 * from the static metamodel of the annotated type.
 *
 * <p>In this example, a method is annotated, and the name of
 * the statement is determined by the name of the annotated
 * method:
 * {@snippet :
 * class Filer {
 *     @Inject EntityManager entityManager;
 *
 *     @JakartaQuery("delete from Record where temporary = true")
 *     int purgeTemporaryRecords() {
 *         return entityManager.createStatement(Filer_.purgeTemporaryRecords())
 *                 .execute();
 *     }
 * }
 * }
 *
 * <p>In this example, the entity class is annotated,
 * and the {@link NamedQuery} annotation explicitly specifies a
 * name:
 * {@snippet :
 * @NamedQuery(name = "updateSales",
 *             query = "update Book set sales = ?1 where isbn = ?2")
 * @Entity class Book { .. }
 * }
 * <p>In this case the {@code StatementReference} obtained from
 * its static metamodel does not include arguments to the query
 * parameters, and so they must be supplied via {@code setParameter}:
 * {@snippet :
 * int updated =
 *         entityManager.createStatement(Book_._updateSales_)
 *                 .setParameter(1, sales)
 *                 .setParameter(2, isbn)
 *                 .execute();
 * }
 *
 * <p>A {@code StatementReference} may include arguments to parameters
 * of the query.
 * <ul>
 * <li>A reference representing a query declared using an
 *     annotation of a type or field has no arguments, and so
 *     {@link #getArguments()}, {@link #getParameterNames()},
 *     and {@link #getParameterTypes()} all return {@code null}.
 * <li>A reference representing a query declared using an
```

```
 *     annotation of a method holds information about the
 *     arguments passed to the method, making it available via
 *     {@link #getArguments()}, {@link #getParameterNames()},
 *     and {@link #getParameterTypes()}.
 * </ul>
 *
 * <p>In the Jakarta Persistence query language, a DELETE or
 * UPDATE statement is not a typed query and is always
 * represented by an instance of {@link StatementReference}.
 * A native SQL query is represented by an instance of
 * {@link StatementReference} if it returns a row count. A
 * Jakarta Persistence SELECT query, or a native SQL query
 * that returns a result set, is usually represented as an
 * instance of {@link TypedQueryReference}.
 *
 * @see EntityHandler#createStatement(StatementReference)
 *
 * @since 4.0
 */
public non-sealed interface StatementReference extends Reference {
    /**
     * All {@linkplain Statement.Option options} specified by
     * {@link jakarta.persistence.query.QueryOptions},
     * {@link NamedQuery}, or {@link NamedNativeQuery}, or an
     * empty set if no options were specified.
     *
     * @see Statement#addOption
     * @see jakarta.persistence.query.QueryOptions#timeout
     * @see jakarta.persistence.query.QueryOptions#flush
     */
    @Nonnull
    Set<Statement.Option> getOptions();
}
```

Appendix C: Criteria API Interfaces

The following APIs are defined in the package `jakarta.persistence.criteria`.

C.1. `CriteriaBuilder`

```
//     Linda DeMichiel      - 2.1
//     Linda DeMichiel      - 2.0


package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.time.temporal.Temporal;
import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.function.Consumer;

import jakarta.persistence.AttributeConverter;
import jakarta.persistence.StatementReference;
import jakarta.persistence.Tuple;
import jakarta.persistence.TypedQueryReference;

/**
 * Used to construct criteria queries, compound selections,
 * expressions, predicates, orderings.
 *
 * <p> Note that {@link Predicate} is used instead of
 * {@code Expression<Boolean>} in this API in
 * order to work around the fact that Java generics are not
 * compatible with varags.
 *
 * @since 2.0
 */
public interface CriteriaBuilder {

    /**
     * Create a {@link CriteriaQuery} object.
     * @return criteria query object
     */
    @Nonnull
    CriteriaQuery<Object> createQuery();

    /**
     * Create a {@link CriteriaQuery} object with the given
     * result type.
     * @param resultClass  type of the query result
     * @param <T> the query result type
     * @return criteria query object
     */
    @Nonnull
    <T> CriteriaQuery<T> createQuery(@Nonnull Class<T> resultClass);

    /**
     * Create a {@link CriteriaQuery} object representing the
     * given Jakarta Persistence query language {@code SELECT}
     * query with the given result type.
     * @param resultClass type of the query result
     * @param jpql A Jakarta Persistence query language
     *             {@code SELECT} query
```

```
 * @param <T> the query result type
 * @return criteria query object
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid, or if the query result is
 *         found to not be assignable to the specified type
 *         and the specified type does not have a suitable
 *         constructor
 * @since 4.0
 */
@Nonnull
<T> CriteriaQuery<T> createQuery(@Nonnull Class<T> resultClass,
                                 @Nonnull String jpql);

/**
 * Create a {@link CriteriaQuery} object representing the
 * given Jakarta Persistence query language {@code SELECT}
 * query.
 * @param jpql A Jakarta Persistence query language
 *             {@code SELECT} query
 * @return criteria query object
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid
 * @since 4.0
 */
@Nonnull
CriteriaQuery<?> createQuery(@Nonnull String jpql);

/**
 * Create a {@link CriteriaQuery} object that returns a
 * tuple of objects as its result.
 * @return criteria query object
 */
@Nonnull
CriteriaQuery<Tuple> createTupleQuery();

// methods to construct queries for bulk updates and deletes:

/**
 * Create a {@link CriteriaUpdate} object to perform a bulk
 * update operation.
 * @param targetEntity  target type for update operation
 * @param <T> the target entity type
 * @return the criteria statement object
 * @since 2.1
 */
@Nonnull
<T> CriteriaUpdate<T> createCriteriaUpdate(@Nonnull Class<T> targetEntity);

/**
 * Create a {@link CriteriaUpdate} object representing the
 * given Jakarta Persistence query language {@code UPDATE}
 * query with the given target entity type.
 * @param jpql A Jakarta Persistence query language
 *             {@code UPDATE} query
 * @param targetEntity target type for update operation
 * @param <T> the target entity type
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid, or if the entity type updated
 *         by the query is not exactly the same as the
 *         specified target type
 * @return the criteria statement object
 * @since 4.0
 */
@Nonnull
<T> CriteriaUpdate<T> createCriteriaUpdate(@Nonnull Class<T> targetEntity,
                                           @Nonnull String jpql);
```

```java
/**
 * Create a {@link CriteriaUpdate} object representing the
 * given Jakarta Persistence query language {@code UPDATE}
 * query.
 * @param jpql A Jakarta Persistence query language
 *             {@code UPDATE} query
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid
 * @return the criteria statement object
 * @since 4.0
 */
@Nonnull
CriteriaUpdate<?> createCriteriaUpdate(@Nonnull String jpql);

/**
 * Create a {@link CriteriaDelete} object to perform a bulk
 * delete operation.
 * @param targetEntity  target type for delete operation
 * @param <T> the target entity type
 * @return the criteria statement object
 * @since 2.1
 */
@Nonnull
<T> CriteriaDelete<T> createCriteriaDelete(@Nonnull Class<T> targetEntity);

/**
 * Create a {@link CriteriaDelete} object representing the
 * given Jakarta Persistence query language {@code DELETE}
 * query with the given target entity type.
 * @param jpql A Jakarta Persistence query language
 *             {@code DELETE} query
 * @param targetEntity target type for delete operation
 * @param <T> the target entity type
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid, or if the entity type deleted
 *         by the query is not exactly the same as the
 *         specified target type
 * @return the criteria statement object
 * @since 4.0
 */
@Nonnull
<T> CriteriaDelete<T> createCriteriaDelete(@Nonnull Class<T> targetEntity,
                                           @Nonnull String jpql);

/**
 * Create a {@link CriteriaDelete} object representing the
 * given Jakarta Persistence query language {@code DELETE}
 * query.
 * @param jpql A Jakarta Persistence query language
 *             {@code DELETE} query
 * @throws IllegalArgumentException if the query string is
 *         found to be invalid
 * @return the criteria statement object
 * @since 4.0
 */
@Nonnull
CriteriaDelete<?> createCriteriaDelete(@Nonnull String jpql);

/**
 * Modify the Jakarta Persistence query language query
 * represented by the given reference, returning a
 * reference to the modified query inheriting all the
 * options of the given reference. This operation never
 * modifies the query represented by the given reference.
 * @param reference A reference to a Jakarta Persistence
 *                  query language {@code SELECT} query
 * @param augmentation a consumer that modifies the query
```

```
 * @param <T> the query result type
 * @return a reference to the modified query
 * @throws IllegalArgumentException if the given reference
 *         does not represent a named Jakarta Persistence
 *         query language {@code SELECT} query belonging
 *         to the owning factory
 * @since 4.0
 */
@Nonnull
<T> TypedQueryReference<T> augment(@Nonnull TypedQueryReference<T> reference,
                                   @Nonnull Consumer<CriteriaQuery<T>> augmentation);

/**
 * Modify the Jakarta Persistence query language statement
 * represented by the given reference, returning a reference
 * to the modified statement inheriting all the options of
 * the given reference. This operation never modifies the
 * query represented by the given reference.
 * @param reference A reference to a Jakarta Persistence
 *                  query language statement
 * @param augmentation a consumer that modifies the statement
 * @return a reference to the modified statement
 * @throws IllegalArgumentException if the given reference
 *         does not represent a named Jakarta Persistence
 *         query language statement belonging to the owning
 *         factory
 * @since 4.0
 */
@Nonnull
StatementReference augment(@Nonnull StatementReference reference,
                           @Nonnull Consumer<CriteriaStatement<?>> augmentation);


// selection construction methods:

/**
 * Create a selection item corresponding to a constructor.
 * This method is used to specify a constructor that is
 * applied to the results of the query execution. If the
 * constructor is for an entity class, the resulting entities
 * will be in the new state after the query is executed.
 * @param resultClass  class whose instance is to be constructed
 * @param selections  arguments to the constructor
 * @param <Y> the type of the constructed result
 * @return compound selection item
 * @throws IllegalArgumentException if an argument is a
 *         tuple- or array-valued selection item
 */
@Nonnull
<Y> CompoundSelection<Y> construct(@Nonnull Class<Y> resultClass,
                                   @Nonnull Selection<?>... selections);

/**
 * Create a tuple-valued selection item.
 * @param selections  selection items
 * @return tuple-valued compound selection
 * @throws IllegalArgumentException if an argument is a
 *         tuple- or array-valued selection item
 */
@Nonnull
CompoundSelection<Tuple> tuple(@Nonnull Selection<?>... selections);

/**
 * Create a tuple-valued selection item.
 * @param selections  list of selection items
 * @return tuple-valued compound selection
 * @throws IllegalArgumentException if an argument is a
```

```
 *         tuple- or array-valued selection item
 * @since 3.2
 */
@Nonnull
CompoundSelection<Tuple> tuple(@Nonnull List<Selection<?>> selections);

/**
 * Create an array-valued selection item.
 * @param selections  selection items
 * @return array-valued compound selection
 * @throws IllegalArgumentException if an argument is a
 *         tuple- or array-valued selection item
 */
@Nonnull
CompoundSelection<Object[]> array(@Nonnull Selection<?>... selections);

/**
 * Create an array-valued selection item.
 * @param selections  list of selection items
 * @return array-valued compound selection
 * @throws IllegalArgumentException if an argument is a
 *         tuple- or array-valued selection item
 * @since 3.2
 */
@Nonnull
CompoundSelection<Object[]> array(@Nonnull List<Selection<?>> selections);


//ordering:

/**
 * Create an ordering by the ascending value of the expression.
 * @param expression  expression used to define the ordering
 * @return ascending ordering corresponding to the expression
 */
@Nonnull
Order asc(@Nonnull Expression<?> expression);

/**
 * Create an ordering by the descending value of the expression.
 * @param expression  expression used to define the ordering
 * @return descending ordering corresponding to the expression
 */
@Nonnull
Order desc(@Nonnull Expression<?> expression);

/**
 * Create an ordering by the ascending value of the expression.
 * @param expression  expression used to define the ordering
 * @param nullPrecedence  the precedence of null values
 * @return ascending ordering corresponding to the expression
 * @since 3.2
 */
@Nonnull
Order asc(@Nonnull Expression<?> expression,
          @Nonnull Nulls nullPrecedence);

/**
 * Create an ordering by the descending value of the expression.
 * @param expression  expression used to define the ordering
 * @param nullPrecedence  the precedence of null values
 * @return descending ordering corresponding to the expression
 * @since 3.2
 */
@Nonnull
Order desc(@Nonnull Expression<?> expression,
           @Nonnull Nulls nullPrecedence);
```

```
//aggregate functions:

/**
 * Create an aggregate expression applying the avg operation.
 * @param x  expression representing input value to avg operation
 * @param <N> the numeric type of the expression
 * @return avg expression
 */
@Nonnull
<N extends Number> Expression<Double> avg(@Nonnull Expression<N> x);

/**
 * Create an aggregate expression applying the sum operation.
 * @param x  expression representing input value to sum operation
 * @param <N> the numeric type of the expression
 * @return sum expression
 */
@Nonnull
<N extends Number> Expression<N> sum(@Nonnull Expression<N> x);

/**
 * Create an aggregate expression applying the sum operation to an
 * Integer-valued expression, returning a Long result.
 * @param x  expression representing input value to sum operation
 * @return sum expression
 */
@Nonnull
Expression<Long> sumAsLong(@Nonnull Expression<Integer> x);

/**
 * Create an aggregate expression applying the sum operation to a
 * Float-valued expression, returning a Double result.
 * @param x  expression representing input value to sum operation
 * @return sum expression
 */
@Nonnull
Expression<Double> sumAsDouble(@Nonnull Expression<Float> x);

/**
 * Create an aggregate expression applying the numerical max
 * operation.
 * @param x  expression representing input value to max operation
 * @param <N> the numeric type of the expression
 * @return max expression
 */
@Nonnull
<N extends Number> Expression<N> max(@Nonnull Expression<N> x);

/**
 * Create an aggregate expression applying the numerical min
 * operation.
 * @param x  expression representing input value to min operation
 * @param <N> the numeric type of the expression
 * @return min expression
 */
@Nonnull
<N extends Number> Expression<N> min(@Nonnull Expression<N> x);

/**
 * Create an aggregate expression for finding the greatest of
 * the values (strings, dates, etc).
 * @param x  expression representing input value to greatest
 *           operation
 * @param <X> the type of the expression
 * @return greatest expression
```

```
 */
@Nonnull
<X extends Comparable<? super X>> Expression<X> greatest(@Nonnull Expression<X> x);

/**
 * Create an aggregate expression for finding the least of
 * the values (strings, dates, etc).
 * @param x  expression representing input value to least
 *           operation
 * @param <X> the type of the expression
 * @return least expression
 */
@Nonnull
<X extends Comparable<? super X>> Expression<X> least(@Nonnull Expression<X> x);

/**
 * Create an aggregate expression applying the count operation.
 * @param x  expression representing input value to count
 *           operation
 * @return count expression
 */
@Nonnull
Expression<Long> count(@Nonnull Expression<?> x);

/**
 * Create an aggregate expression applying the count distinct
 * operation.
 * @param x  expression representing input value to
 *        count distinct operation
 * @return count distinct expression
 */
@Nonnull
Expression<Long> countDistinct(@Nonnull Expression<?> x);



//subqueries:

/**
 * Create a predicate testing the existence of a subquery result.
 * @param subquery  subquery whose result is to be tested
 * @return exists predicate
 */
@Nonnull
Predicate exists(@Nonnull Subquery<?> subquery);

/**
 * Create an all expression over the subquery results.
 * @param subquery  subquery
 * @param <Y> the subquery result type
 * @return all expression
 */
@Nonnull
<Y> Expression<Y> all(@Nonnull Subquery<Y> subquery);

/**
 * Create a some expression over the subquery results.
 * This expression is equivalent to an {@code any} expression.
 * @param subquery  subquery
 * @param <Y> the subquery result type
 * @return some expression
 */
@Nonnull
<Y> Expression<Y> some(@Nonnull Subquery<Y> subquery);

/**
 * Create an any expression over the subquery results.
```

```
 * This expression is equivalent to a {@code some} expression.
 * @param subquery  subquery
 * @param <Y> the subquery result type
 * @return any expression
 */
@Nonnull
<Y> Expression<Y> any(@Nonnull Subquery<Y> subquery);


//boolean functions:

/**
 * Create a conjunction of the given boolean expressions.
 * @param x  boolean expression
 * @param y  boolean expression
 * @return and predicate
 */
@Nonnull
Predicate and(@Nonnull Expression<Boolean> x,
              @Nonnull Expression<Boolean> y);

/**
 * Create a conjunction of the given restriction predicates.
 * A conjunction of zero predicates is true.
 * @param restrictions  zero or more restriction predicates
 * @return and predicate
 */
@Nonnull
Predicate and(@Nonnull BooleanExpression... restrictions);

/**
 * Create a conjunction of the given restriction predicates.
 * A conjunction of zero predicates is true.
 * @param restrictions  a list of zero or more restriction predicates
 * @return and predicate
 * @since 3.2
 */
@Nonnull
Predicate and(@Nonnull List<? extends Expression<Boolean>> restrictions);

/**
 * Create a disjunction of the given boolean expressions.
 * @param x  boolean expression
 * @param y  boolean expression
 * @return or predicate
 */
@Nonnull
Predicate or(@Nonnull Expression<Boolean> x,
             @Nonnull Expression<Boolean> y);

/**
 * Create a disjunction of the given restriction predicates.
 * A disjunction of zero predicates is false.
 * @param restrictions  zero or more restriction predicates
 * @return or predicate
 */
@Nonnull
Predicate or(@Nonnull BooleanExpression... restrictions);

/**
 * Create a disjunction of the given restriction predicates.
 * A disjunction of zero predicates is false.
 * @param restrictions  a list of zero or more restriction predicates
 * @return or predicate
 * @since 3.2
 */
@Nonnull
```

```java
Predicate or(@Nonnull List<? extends Expression<Boolean>> restrictions);

/**
 * Create a negation of the given restriction.
 * @param restriction  restriction expression
 * @return not predicate
 */
@Nonnull
Predicate not(@Nonnull Expression<Boolean> restriction);

/**
 * Create a conjunction (with zero conjuncts).
 * A conjunction with zero conjuncts is true.
 * @return and predicate
 */
@Nonnull
Predicate conjunction();

/**
 * Create a disjunction (with zero disjuncts).
 * A disjunction with zero disjuncts is false.
 * @return or predicate
 */
@Nonnull
Predicate disjunction();


//turn Expression<Boolean> into a Predicate
//useful for use with varargs methods

/**
 * Create a predicate testing for a true value.
 * @param x  expression to be tested
 * @return predicate
 */
@Nonnull
Predicate isTrue(@Nonnull Expression<Boolean> x);

/**
 * Create a predicate testing for a false value.
 * @param x  expression to be tested
 * @return predicate
 */
@Nonnull
Predicate isFalse(@Nonnull Expression<Boolean> x);


//null tests:

/**
 * Create a predicate to test whether the expression is null.
 * @param x expression
 * @return is-null predicate
 */
@Nonnull
Predicate isNull(@Nonnull Expression<?> x);

/**
 * Create a predicate to test whether the expression is not null.
 * @param x expression
 * @return is-not-null predicate
 */
@Nonnull
Predicate isNotNull(@Nonnull Expression<?> x);

//equality:
```

```java
/**
 * Create a predicate for testing the arguments for equality.
 * @param x  expression
 * @param y  expression
 * @return equality predicate
 */
@Nonnull
Predicate equal(@Nonnull Expression<?> x,
                @Nonnull Expression<?> y);

/**
 * Create a predicate for testing the arguments for equality.
 * @param x  expression
 * @param y  object
 * @return equality predicate
 */
@Nonnull
Predicate equal(@Nonnull Expression<?> x,
                Object y);

/**
 * Create a predicate for testing the arguments for inequality.
 * @param x  expression
 * @param y  expression
 * @return inequality predicate
 */
@Nonnull
Predicate notEqual(@Nonnull Expression<?> x,
                   @Nonnull Expression<?> y);

/**
 * Create a predicate for testing the arguments for inequality.
 * @param x  expression
 * @param y  object
 * @return inequality predicate
 */
@Nonnull
Predicate notEqual(@Nonnull Expression<?> x,
                   Object y);


//comparisons for generic (non-numeric) operands:

/**
 * Create a predicate for testing whether the first argument is
 * greater than the second.
 * @param x  expression
 * @param y  expression
 * @param <Y> the type of the compared expressions
 * @return greater-than predicate
 * @see #gt(Expression, Expression)
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate greaterThan(@Nonnull Expression<? extends Y> x,
                                                        @Nonnull Expression<? extends Y> y);

/**
 * Create a predicate for testing whether the first argument is
 * greater than the second.
 * @param x  expression
 * @param y  value
 * @param <Y> the type of the compared values
 * @return greater-than predicate
 * @see #gt(Expression, Number)
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate greaterThan(@Nonnull Expression<? extends Y> x,
```

```
                                                        Y y);

/**
 * Create a predicate for testing whether the first argument is
 * greater than or equal to the second.
 * @param x  expression
 * @param y  expression
 * @param <Y> the type of the compared expressions
 * @return greater-than-or-equal predicate
 * @see #ge(Expression, Expression)
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate greaterThanOrEqualTo(@Nonnull Expression<? extends Y> x,
                                                                 @Nonnull Expression<? extends Y> y);

/**
 * Create a predicate for testing whether the first argument is
 * greater than or equal to the second.
 * @param x  expression
 * @param y  value
 * @param <Y> the type of the compared values
 * @return greater-than-or-equal predicate
 * @see #ge(Expression, Number)
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate greaterThanOrEqualTo(@Nonnull Expression<? extends Y> x,
                                                                 Y y);

/**
 * Create a predicate for testing whether the first argument is
 * less than the second.
 * @param x  expression
 * @param y  expression
 * @param <Y> the type of the compared expressions
 * @return less-than predicate
 * @see #lt(Expression, Expression)
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate lessThan(@Nonnull Expression<? extends Y> x,
                                                     @Nonnull Expression<? extends Y> y);

/**
 * Create a predicate for testing whether the first argument is
 * less than the second.
 * @param x  expression
 * @param y  value
 * @param <Y> the type of the compared values
 * @return less-than predicate
 * @see #lt(Expression, Number)
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate lessThan(@Nonnull Expression<? extends Y> x,
                                                     Y y);

/**
 * Create a predicate for testing whether the first argument is
 * less than or equal to the second.
 * @param x  expression
 * @param y  expression
 * @param <Y> the type of the compared expressions
 * @return less-than-or-equal predicate
 * @see #le(Expression, Expression)
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate lessThanOrEqualTo(@Nonnull Expression<? extends Y> x,
                                                              @Nonnull Expression<? extends Y> y);
```

```java
/**
 * Create a predicate for testing whether the first argument is
 * less than or equal to the second.
 * @param x  expression
 * @param y  value
 * @param <Y> the type of the compared values
 * @return less-than-or-equal predicate
 * @see #le(Expression, Number)
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate lessThanOrEqualTo(@Nonnull Expression<? extends Y> x,
                                                              Y y);

/**
 * Create a predicate for testing whether the first argument is
 * between the second and third arguments in value.
 * @param v  expression
 * @param x  expression
 * @param y  expression
 * @param <Y> the type of the compared expressions
 * @return between predicate
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate between(@Nonnull Expression<? extends Y> v,
                                                    @Nonnull Expression<? extends Y> x,
                                                    @Nonnull Expression<? extends Y> y);

/**
 * Create a predicate for testing whether the first argument is
 * between the second and third arguments in value.
 * @param v  expression
 * @param x  value
 * @param y  value
 * @param <Y> the type of the compared values
 * @return between predicate
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate between(@Nonnull Expression<? extends Y> v,
                                                    Y x, Y y);

/**
 * Create a predicate for testing whether the first argument is
 * between the second and third arguments in value.
 * @param v  value
 * @param x  expression
 * @param y  expression
 * @param <Y> the type of the compared values
 * @return between predicate
 * @since 4.0
 */
@Nonnull
<Y extends Comparable<? super Y>> Predicate between(Y v,
                                                    @Nonnull Expression<? extends Y> x,
                                                    @Nonnull Expression<? extends Y> y);

//comparisons for numeric operands:

/**
 * Create a predicate for testing whether the first argument is
 * greater than the second. This operation accepts arguments of
 * heterogeneous numeric types.
 * @param x  expression
 * @param y  expression
 * @return greater-than predicate
 */
@Nonnull
Predicate gt(@Nonnull Expression<? extends Number> x,
```

```
            @Nonnull Expression<? extends Number> y);

/**
 * Create a predicate for testing whether the first argument is
 * greater than the second. This operation accepts arguments of
 * heterogeneous numeric types.
 * @param x  expression
 * @param y  value
 * @return greater-than predicate
 */
@Nonnull
Predicate gt(@Nonnull Expression<? extends Number> x,
             Number y);

/**
 * Create a predicate for testing whether the first argument is
 * greater than or equal to the second. This operation accepts
 * arguments of heterogeneous numeric types.
 * @param x  expression
 * @param y  expression
 * @return greater-than-or-equal predicate
 */
@Nonnull
Predicate ge(@Nonnull Expression<? extends Number> x,
             @Nonnull Expression<? extends Number> y);

/**
 * Create a predicate for testing whether the first argument is
 * greater than or equal to the second. This operation accepts
 * arguments of heterogeneous numeric types.
 * @param x  expression
 * @param y  value
 * @return greater-than-or-equal predicate
 */
@Nonnull
Predicate ge(@Nonnull Expression<? extends Number> x,
             Number y);

/**
 * Create a predicate for testing whether the first argument is
 * less than the second. This operation accepts arguments of
 * heterogeneous numeric types.
 * @param x  expression
 * @param y  expression
 * @return less-than predicate
 */
@Nonnull
Predicate lt(@Nonnull Expression<? extends Number> x,
             @Nonnull Expression<? extends Number> y);

/**
 * Create a predicate for testing whether the first argument is
 * less than the second. This operation accepts arguments of
 * heterogeneous numeric types.
 * @param x  expression
 * @param y  value
 * @return less-than predicate
 */
@Nonnull
Predicate lt(@Nonnull Expression<? extends Number> x,
             Number y);

/**
 * Create a predicate for testing whether the first argument is
 * less than or equal to the second. This operation accepts
 * arguments of heterogeneous numeric types.
 * @param x  expression
```

```
 * @param y  expression
 * @return less-than-or-equal predicate
 */
@Nonnull
Predicate le(@Nonnull Expression<? extends Number> x,
             @Nonnull Expression<? extends Number> y);

/**
 * Create a predicate for testing whether the first argument is
 * less than or equal to the second. This operation accepts
 * arguments of heterogeneous numeric types.
 * @param x  expression
 * @param y  value
 * @return less-than-or-equal predicate
 */
@Nonnull
Predicate le(@Nonnull Expression<? extends Number> x,
             Number y);


//numerical operations:

/**
 * Create an expression that returns the sign of its
 * argument, that is, {@code 1} if its argument is
 * positive, {@code -1} if its argument is negative,
 * or {@code 0} if its argument is exactly zero.
 * @param x expression
 * @return sign
 */
@Nonnull
Expression<Integer> sign(@Nonnull Expression<? extends Number> x);

/**
 * Create an expression that returns the arithmetic negation
 * of its argument.
 * @param x expression
 * @param <N> the numeric type of the expression
 * @return arithmetic negation
 */
@Nonnull
<N extends Number> Expression<N> neg(@Nonnull Expression<N> x);

/**
 * Create an expression that returns the absolute value
 * of its argument.
 * @param x expression
 * @param <N> the numeric type of the expression
 * @return absolute value
 */
@Nonnull
<N extends Number> Expression<N> abs(@Nonnull Expression<N> x);

/**
 * Create an expression that returns the ceiling of its
 * argument, that is, the smallest integer greater than
 * or equal to its argument.
 * @param x expression
 * @param <N> the numeric type of the expression
 * @return ceiling
 */
@Nonnull
<N extends Number> Expression<N> ceiling(@Nonnull Expression<N> x);

/**
 * Create an expression that returns the floor of its
 * argument, that is, the largest integer smaller than
```

```
 * or equal to its argument.
 * @param x expression
 * @param <N> the numeric type of the expression
 * @return floor
 */
@Nonnull
<N extends Number> Expression<N> floor(@Nonnull Expression<N> x);
/**
 * Create an expression that returns the sum
 * of its arguments.
 * @param x expression
 * @param y expression
 * @param <N> the numeric type of the expressions
 * @return sum
 */
@Nonnull
<N extends Number> Expression<N> sum(@Nonnull Expression<? extends N> x,
                                     @Nonnull Expression<? extends N> y);

/**
 * Create an expression that returns the sum
 * of its arguments.
 * @param x expression
 * @param y value
 * @param <N> the numeric type of the arguments
 * @return sum
 */
@Nonnull
<N extends Number> Expression<N> sum(@Nonnull Expression<? extends N> x,
                                     N y);

/**
 * Create an expression that returns the sum
 * of its arguments.
 * @param x value
 * @param y expression
 * @param <N> the numeric type of the arguments
 * @return sum
 */
@Nonnull
<N extends Number> Expression<N> sum(N x,
                                     @Nonnull Expression<? extends N> y);

/**
 * Create an expression that returns the product
 * of its arguments.
 * @param x expression
 * @param y expression
 * @param <N> the numeric type of the expressions
 * @return product
 */
@Nonnull
<N extends Number> Expression<N> prod(@Nonnull Expression<? extends N> x,
                                      @Nonnull Expression<? extends N> y);

/**
 * Create an expression that returns the product
 * of its arguments.
 * @param x expression
 * @param y value
 * @param <N> the numeric type of the arguments
 * @return product
 */
@Nonnull
<N extends Number> Expression<N> prod(@Nonnull Expression<? extends N> x,
                                      N y);
```

```
/**
 * Create an expression that returns the product
 * of its arguments.
 * @param x value
 * @param y expression
 * @param <N> the numeric type of the arguments
 * @return product
 */
@Nonnull
<N extends Number> Expression<N> prod(N x,
                                      @Nonnull Expression<? extends N> y);

/**
 * Create an expression that returns the difference
 * between its arguments.
 * @param x expression
 * @param y expression
 * @param <N> the numeric type of the expressions
 * @return difference
 */
@Nonnull
<N extends Number> Expression<N> diff(@Nonnull Expression<? extends N> x,
                                      @Nonnull Expression<? extends N> y);

/**
 * Create an expression that returns the difference
 * between its arguments.
 * @param x expression
 * @param y value
 * @param <N> the numeric type of the arguments
 * @return difference
 */
@Nonnull
<N extends Number> Expression<N> diff(@Nonnull Expression<? extends N> x,
                                      N y);

/**
 * Create an expression that returns the difference
 * between its arguments.
 * @param x value
 * @param y expression
 * @param <N> the numeric type of the arguments
 * @return difference
 */
@Nonnull
<N extends Number> Expression<N> diff(N x,
                                      @Nonnull Expression<? extends N> y);

/**
 * Create an expression that returns the quotient
 * of its arguments.
 * @param x expression
 * @param y expression
 * @return quotient
 */
@Nonnull
Expression<Number> quot(@Nonnull Expression<? extends Number> x,
                        @Nonnull Expression<? extends Number> y);

/**
 * Create an expression that returns the quotient
 * of its arguments.
 * @param x expression
 * @param y value
 * @return quotient
 */
@Nonnull
```

```
Expression<Number> quot(@Nonnull Expression<? extends Number> x,
                        Number y);

/**
 * Create an expression that returns the quotient
 * of its arguments.
 * @param x value
 * @param y expression
 * @return quotient
 */
@Nonnull
Expression<Number> quot(Number x,
                        @Nonnull Expression<? extends Number> y);

/**
 * Create an expression that returns the modulus
 * (remainder under integer division) of its
 * arguments.
 * @param x expression
 * @param y expression
 * @return modulus
 */
@Nonnull
Expression<Integer> mod(@Nonnull Expression<Integer> x,
                        @Nonnull Expression<Integer> y);

/**
 * Create an expression that returns the modulus
 * (remainder under integer division) of its
 * arguments.
 * @param x expression
 * @param y value
 * @return modulus
 */
@Nonnull
Expression<Integer> mod(@Nonnull Expression<Integer> x,
                        Integer y);

/**
 * Create an expression that returns the modulus
 * (remainder under integer division) of its
 * arguments.
 * @param x value
 * @param y expression
 * @return modulus
 */
@Nonnull
Expression<Integer> mod(Integer x,
                        @Nonnull Expression<Integer> y);

/**
 * Create an expression that returns the square root
 * of its argument.
 * @param x expression
 * @return square root
 */
@Nonnull
Expression<Double> sqrt(@Nonnull Expression<? extends Number> x);

/**
 * Create an expression that returns the exponential
 * of its argument, that is, Euler's number <i>e</i>
 * raised to the power of its argument.
 * @param x expression
 * @return exponential
 */
@Nonnull
```

```
Expression<Double> exp(@Nonnull Expression<? extends Number> x);

/**
 * Create an expression that returns the natural logarithm
 * of its argument.
 * @param x expression
 * @return natural logarithm
 */
@Nonnull
Expression<Double> ln(@Nonnull Expression<? extends Number> x);

/**
 * Create an expression that returns the first argument
 * raised to the power of its second argument.
 * @param x base
 * @param y exponent
 * @return the base raised to the power of the exponent
 */
@Nonnull
Expression<Double> power(@Nonnull Expression<? extends Number> x,
                         @Nonnull Expression<? extends Number> y);

/**
 * Create an expression that returns the first argument
 * raised to the power of its second argument.
 * @param x base
 * @param y exponent
 * @return the base raised to the power of the exponent
 */
@Nonnull
Expression<Double> power(@Nonnull Expression<? extends Number> x,
                         Number y);

/**
 * Create an expression that returns the first argument
 * rounded to the number of decimal places given by the
 * second argument.
 * @param x base
 * @param n number of decimal places
 * @param <T> the numeric type of the expression
 * @return the rounded value
 */
@Nonnull
<T extends Number> Expression<T> round(@Nonnull Expression<T> x,
                                       @Nonnull Integer n);


//typecasts:

/**
 * Typecast.
 * @param number  numeric expression
 * @return {@literal Expression<Long>}
 */
@Nonnull
Expression<Long> toLong(@Nonnull Expression<? extends Number> number);

/**
 * Typecast.
 * @param number  numeric expression
 * @return {@literal Expression<Integer>}
 */
@Nonnull
Expression<Integer> toInteger(@Nonnull Expression<? extends Number> number);

/**
 * Typecast.
```

```
 * @param number  numeric expression
 * @return {@literal Expression<Float>}
 */
@Nonnull
Expression<Float> toFloat(@Nonnull Expression<? extends Number> number);

/**
 * Typecast.
 * @param number  numeric expression
 * @return {@literal Expression<Double>}
 */
@Nonnull
Expression<Double> toDouble(@Nonnull Expression<? extends Number> number);

/**
 * Typecast.
 * @param number  numeric expression
 * @return {@literal Expression<BigDecimal>}
 */
@Nonnull
Expression<BigDecimal> toBigDecimal(@Nonnull Expression<? extends Number> number);

/**
 * Typecast.
 * @param number  numeric expression
 * @return {@literal Expression<BigInteger>}
 */
@Nonnull
Expression<BigInteger> toBigInteger(@Nonnull Expression<? extends Number> number);

/**
 * Typecast.
 * @param character expression
 * @return {@literal Expression<String>}
 */
@Nonnull
Expression<String> toString(@Nonnull Expression<Character> character);


//literals:

/**
 * Create an expression for a literal.
 * @param value  value represented by the expression
 * @param <T> the type of the literal value
 * @return expression literal
 * @throws IllegalArgumentException if value is null
 */
@Nonnull
<T> Expression<T> literal(@Nonnull T value);

/**
 * Create a number expression for a literal.
 * @param value  value represented by the expression
 * @param <N> the numeric type of the literal value
 * @return expression literal
 * @throws IllegalArgumentException if value is null
 * @since 4.0
 */
@Nonnull
<N extends Number & Comparable<N>> NumericExpression<N> numericLiteral(@Nonnull N value);

/**
 * Create a string expression for a literal.
 * @param value  value represented by the expression
 * @return expression literal
 * @throws IllegalArgumentException if value is null
```

```
 * @since 4.0
 */
@Nonnull
TextExpression stringLiteral(@Nonnull String value);

/**
 * Create a local date expression for a literal.
 * @param value  value represented by the expression
 * @param <T> the temporal type of the literal value
 * @return expression literal
 * @throws IllegalArgumentException if value is null
 * @since 4.0
 */
@Nonnull
<T extends Temporal & Comparable<? super T>> TemporalExpression<T> temporalLiteral(@Nonnull T value);

/**
 * Create a boolean expression for a literal.
 * @return expression literal
 * @since 4.0
 */
@Nonnull
BooleanExpression booleanLiteral(boolean value);

/**
 * Create an expression for a null literal with the given type.
 * @param resultClass  type of the null literal
 * @param <T> the type of the null literal
 * @return null expression literal
 */
@Nonnull
<T> Expression<T> nullLiteral(@Nonnull Class<T> resultClass);

//parameters:

/**
 * Create a parameter expression.
 * @param paramClass parameter class
 * @param <T> the parameter type
 * @return parameter expression
 */
@Nonnull
<T> ParameterExpression<T> parameter(@Nonnull Class<T> paramClass);

/**
 * Create a parameter expression with the given name.
 * @param paramClass parameter class
 * @param name  name that can be used to refer to
 *              the parameter
 * @param <T> the parameter type
 * @return parameter expression
 */
@Nonnull
<T> ParameterExpression<T> parameter(@Nonnull Class<T> paramClass,
                                     @Nonnull String name);

/**
 * Create a parameter expression whose value is bound via a
 * {@linkplain AttributeConverter converter}.
 * @param converter the class of the attribute converter
 * @param <T> the parameter type
 * @return parameter expression
 * @since 4.0
 */
@Nonnull
<T> ParameterExpression<T> convertedParameter(@Nonnull Class<? extends AttributeConverter<T,?>> converter);
```

```java
//collection operations:

/**
 * Create a predicate that tests whether a collection is empty.
 * @param collection expression
 * @param <C> the collection type
 * @return is-empty predicate
 */
@Nonnull
<C extends Collection<?>> Predicate isEmpty(@Nonnull Expression<C> collection);

/**
 * Create a predicate that tests whether a collection is
 * not empty.
 * @param collection expression
 * @param <C> the collection type
 * @return is-not-empty predicate
 */
@Nonnull
<C extends Collection<?>> Predicate isNotEmpty(@Nonnull Expression<C> collection);

/**
 * Create an expression that tests the size of a collection.
 * @param collection expression
 * @param <C> the collection type
 * @return size expression
 */
@Nonnull
<C extends Collection<?>> Expression<Integer> size(@Nonnull Expression<C> collection);

/**
 * Create an expression that tests the size of a collection.
 * @param collection collection
 * @param <C> the collection type
 * @return size expression
 */
@Nonnull
<C extends Collection<?>> Expression<Integer> size(@Nonnull C collection);

/**
 * Create a predicate that tests whether an element is
 * a member of a collection.
 * If the collection is empty, the predicate will be false.
 * @param elem element expression
 * @param collection expression
 * @param <E> the element type
 * @param <C> the collection type
 * @return is-member predicate
 */
@Nonnull
<E, C extends Collection<E>> Predicate isMember(@Nonnull Expression<E> elem,
                                                @Nonnull Expression<C> collection);

/**
 * Create a predicate that tests whether an element is
 * a member of a collection.
 * If the collection is empty, the predicate will be false.
 * @param elem element
 * @param collection expression
 * @param <E> the element type
 * @param <C> the collection type
 * @return is-member predicate
 */
@Nonnull
<E, C extends Collection<E>> Predicate isMember(E elem,
                                                @Nonnull Expression<C> collection);
```

```java
/**
 * Create a predicate that tests whether an element is
 * not a member of a collection.
 * If the collection is empty, the predicate will be true.
 * @param elem element expression
 * @param collection expression
 * @param <E> the element type
 * @param <C> the collection type
 * @return is-not-member predicate
 */
@Nonnull
<E, C extends Collection<E>> Predicate isNotMember(@Nonnull Expression<E> elem,
                                                   @Nonnull Expression<C> collection);

/**
 * Create a predicate that tests whether an element is
 * not a member of a collection.
 * If the collection is empty, the predicate will be true.
 * @param elem element
 * @param collection expression
 * @param <E> the element type
 * @param <C> the collection type
 * @return is-not-member predicate
 */
@Nonnull
<E, C extends Collection<E>> Predicate isNotMember(E elem,
                                                   @Nonnull Expression<C> collection);


//string functions:

/**
 * Create a predicate for testing whether the expression
 * satisfies the given pattern.
 * @param x  string expression
 * @param pattern  string expression
 * @return like predicate
 */
@Nonnull
Predicate like(@Nonnull Expression<String> x,
               @Nonnull Expression<String> pattern);

/**
 * Create a predicate for testing whether the expression
 * satisfies the given pattern.
 * @param x  string expression
 * @param pattern  string
 * @return like predicate
 */
@Nonnull
Predicate like(@Nonnull Expression<String> x,
               @Nonnull String pattern);

/**
 * Create a predicate for testing whether the expression
 * satisfies the given pattern.
 * @param x  string expression
 * @param pattern  string expression
 * @param escapeChar  escape character expression
 * @return like predicate
 */
@Nonnull
Predicate like(@Nonnull Expression<String> x,
               @Nonnull Expression<String> pattern,
               @Nonnull Expression<Character> escapeChar);

/**
```

```
 * Create a predicate for testing whether the expression
 * satisfies the given pattern.
 * @param x  string expression
 * @param pattern  string expression
 * @param escapeChar  escape character
 * @return like predicate
 */
@Nonnull
Predicate like(@Nonnull Expression<String> x,
               @Nonnull Expression<String> pattern,
               char escapeChar);

/**
 * Create a predicate for testing whether the expression
 * satisfies the given pattern.
 * @param x  string expression
 * @param pattern  string
 * @param escapeChar  escape character expression
 * @return like predicate
 */
@Nonnull
Predicate like(@Nonnull Expression<String> x,
               @Nonnull String pattern,
               @Nonnull Expression<Character> escapeChar);

/**
 * Create a predicate for testing whether the expression
 * satisfies the given pattern.
 * @param x  string expression
 * @param pattern  string
 * @param escapeChar  escape character
 * @return like predicate
 */
@Nonnull
Predicate like(@Nonnull Expression<String> x,
               @Nonnull String pattern,
               char escapeChar);

/**
 * Create a predicate for testing whether the expression
 * does not satisfy the given pattern.
 * @param x  string expression
 * @param pattern  string expression
 * @return not-like predicate
 */
@Nonnull
Predicate notLike(@Nonnull Expression<String> x,
                  @Nonnull Expression<String> pattern);

/**
 * Create a predicate for testing whether the expression
 * does not satisfy the given pattern.
 * @param x  string expression
 * @param pattern  string
 * @return not-like predicate
 */
@Nonnull
Predicate notLike(@Nonnull Expression<String> x,
                  @Nonnull String pattern);

/**
 * Create a predicate for testing whether the expression
 * does not satisfy the given pattern.
 * @param x  string expression
 * @param pattern  string expression
 * @param escapeChar  escape character expression
 * @return not-like predicate
```

```
 */
@Nonnull
Predicate notLike(@Nonnull Expression<String> x,
                  @Nonnull Expression<String> pattern,
                  @Nonnull Expression<Character> escapeChar);

/**
 * Create a predicate for testing whether the expression
 * does not satisfy the given pattern.
 * @param x  string expression
 * @param pattern  string expression
 * @param escapeChar  escape character
 * @return not-like predicate
 */
@Nonnull
Predicate notLike(@Nonnull Expression<String> x,
                  @Nonnull Expression<String> pattern,
                  char escapeChar);

/**
 * Create a predicate for testing whether the expression
 * does not satisfy the given pattern.
 * @param x  string expression
 * @param pattern  string
 * @param escapeChar  escape character expression
 * @return not-like predicate
 */
@Nonnull
Predicate notLike(@Nonnull Expression<String> x,
                  @Nonnull String pattern,
                  @Nonnull Expression<Character> escapeChar);

/**
 * Create a predicate for testing whether the expression
 * does not satisfy the given pattern.
 * @param x  string expression
 * @param pattern  string
 * @param escapeChar  escape character
 * @return not-like predicate
 */
@Nonnull
Predicate notLike(@Nonnull Expression<String> x,
                  @Nonnull String pattern,
                  char escapeChar);

/**
 * Create an expression for string concatenation.
 * If the given list of expressions is empty, returns
 * an expression equivalent to {@code literal("")}.
 * @param expressions  string expressions
 * @return expression corresponding to concatenation
 */
@Nonnull
Expression<String> concat(@Nonnull List<Expression<String>> expressions);

/**
 * Create an expression for string concatenation.
 * @param x  string expression
 * @param y  string expression
 * @return expression corresponding to concatenation
 */
@Nonnull
Expression<String> concat(@Nonnull Expression<String> x,
                          @Nonnull Expression<String> y);

/**
 * Create an expression for string concatenation.
```

```
 * @param x  string expression
 * @param y  string
 * @return expression corresponding to concatenation
 */
@Nonnull
Expression<String> concat(@Nonnull Expression<String> x,
                          @Nonnull String y);

/**
 * Create an expression for string concatenation.
 * @param x  string
 * @param y  string expression
 * @return expression corresponding to concatenation
 */
@Nonnull
Expression<String> concat(@Nonnull String x,
                          @Nonnull Expression<String> y);

/**
 * Create an expression for substring extraction.
 * Extracts a substring starting at the specified position
 * through to end of the string.
 * First position is 1.
 * @param x  string expression
 * @param from  start position expression
 * @return expression corresponding to substring extraction
 */
@Nonnull
Expression<String> substring(@Nonnull Expression<String> x,
                             @Nonnull Expression<Integer> from);

/**
 * Create an expression for substring extraction.
 * Extracts a substring starting at the specified position
 * through to end of the string.
 * First position is 1.
 * @param x  string expression
 * @param from  start position
 * @return expression corresponding to substring extraction
 */
@Nonnull
Expression<String> substring(@Nonnull Expression<String> x,
                             int from);

/**
 * Create an expression for substring extraction.
 * Extracts a substring of given length starting at the
 * specified position.
 * First position is 1.
 * @param x  string expression
 * @param from  start position expression
 * @param len  length expression
 * @return expression corresponding to substring extraction
 */
@Nonnull
Expression<String> substring(@Nonnull Expression<String> x,
                             @Nonnull Expression<Integer> from,
                             @Nonnull Expression<Integer> len);

/**
 * Create an expression for substring extraction.
 * Extracts a substring of given length starting at the
 * specified position.
 * First position is 1.
 * @param x  string expression
 * @param from  start position
 * @param len  length
```

```
 * @return expression corresponding to substring extraction
 */
@Nonnull
Expression<String> substring(@Nonnull Expression<String> x,
                             int from, int len);

/**
 * Used to specify how strings are trimmed.
 */
enum Trimspec {

    /**
     * Trim from leading end.
     */
    LEADING,

    /**
     * Trim from trailing end.
     */
    TRAILING,

    /**
     * Trim from both ends.
     */
    BOTH
}

/**
 * Create expression to trim blanks from both ends of
 * a string.
 * @param x  expression for string to trim
 * @return trim expression
 */
@Nonnull
Expression<String> trim(@Nonnull Expression<String> x);

/**
 * Create expression to trim blanks from a string.
 * @param ts  trim specification
 * @param x  expression for string to trim
 * @return trim expression
 */
@Nonnull
Expression<String> trim(@Nonnull Trimspec ts,
                        @Nonnull Expression<String> x);

/**
 * Create expression to trim character from both ends of
 * a string.
 * @param t  expression for character to be trimmed
 * @param x  expression for string to trim
 * @return trim expression
 */
@Nonnull
Expression<String> trim(@Nonnull Expression<Character> t,
                        @Nonnull Expression<String> x);

/**
 * Create expression to trim character from a string.
 * @param ts  trim specification
 * @param t  expression for character to be trimmed
 * @param x  expression for string to trim
 * @return trim expression
 */
@Nonnull
Expression<String> trim(@Nonnull Trimspec ts,
                        @Nonnull Expression<Character> t,
```

```
                        @Nonnull Expression<String> x);

/**
 * Create expression to trim character from both ends of
 * a string.
 * @param t  character to be trimmed
 * @param x  expression for string to trim
 * @return trim expression
 */
@Nonnull
Expression<String> trim(char t,
                        @Nonnull Expression<String> x);

/**
 * Create expression to trim character from a string.
 * @param ts  trim specification
 * @param t  character to be trimmed
 * @param x  expression for string to trim
 * @return trim expression
 */
@Nonnull
Expression<String> trim(@Nonnull Trimspec ts, char t,
                        @Nonnull Expression<String> x);

/**
 * Create expression for converting a string to lowercase.
 * @param x  string expression
 * @return expression to convert to lowercase
 */
@Nonnull
Expression<String> lower(@Nonnull Expression<String> x);

/**
 * Create expression for converting a string to uppercase.
 * @param x  string expression
 * @return expression to convert to uppercase
 */
@Nonnull
Expression<String> upper(@Nonnull Expression<String> x);

/**
 * Create expression to return length of a string.
 * @param x  string expression
 * @return length expression
 */
@Nonnull
Expression<Integer> length(@Nonnull Expression<String> x);

/**
 * Create an expression for the leftmost substring of a string,
 * @param x  string expression
 * @param len  length of the substring to return
 * @return expression for the leftmost substring
 */
@Nonnull
Expression<String> left(@Nonnull Expression<String> x,
                        int len);

/**
 * Create an expression for the rightmost substring of a string,
 * @param x  string expression
 * @param len  length of the substring to return
 * @return expression for the rightmost substring
 */
@Nonnull
Expression<String> right(@Nonnull Expression<String> x,
                         int len);
```

```java
/**
 * Create an expression for the leftmost substring of a string,
 * @param x  string expression
 * @param len  length of the substring to return
 * @return expression for the leftmost substring
 */
@Nonnull
Expression<String> left(@Nonnull Expression<String> x,
                        @Nonnull Expression<Integer> len);

/**
 * Create an expression for the rightmost substring of a string,
 * @param x  string expression
 * @param len  length of the substring to return
 * @return expression for the rightmost substring
 */
@Nonnull
Expression<String> right(@Nonnull Expression<String> x,
                         @Nonnull Expression<Integer> len);

/**
 * Create an expression replacing every occurrence of a substring
 * within a string.
 * @param x  string expression
 * @param substring  the literal substring to replace
 * @param replacement  the replacement string
 * @return expression for the resulting string
 */
@Nonnull
Expression<String> replace(@Nonnull Expression<String> x,
                           @Nonnull Expression<String> substring,
                           @Nonnull Expression<String> replacement);

/**
 * Create an expression replacing every occurrence of a substring
 * within a string.
 * @param x  string expression
 * @param substring  the literal substring to replace
 * @param replacement  the replacement string
 * @return expression for the resulting string
 */
@Nonnull
Expression<String> replace(@Nonnull Expression<String> x,
                           @Nonnull String substring,
                           @Nonnull Expression<String> replacement);

/**
 * Create an expression replacing every occurrence of a substring
 * within a string.
 * @param x  string expression
 * @param substring  the literal substring to replace
 * @param replacement  the replacement string
 * @return expression for the resulting string
 */
@Nonnull
Expression<String> replace(@Nonnull Expression<String> x,
                           @Nonnull Expression<String> substring,
                           @Nonnull String replacement);

/**
 * Create an expression replacing every occurrence of a substring
 * within a string.
 * @param x  string expression
 * @param substring  the literal substring to replace
 * @param replacement  the replacement string
 * @return expression for the resulting string
```

```
 */
@Nonnull
Expression<String> replace(@Nonnull Expression<String> x,
                           @Nonnull String substring,
                           @Nonnull String replacement);


/**
 * Create expression to locate the position of one string
 * within another, returning position of first character
 * if found.
 * The first position in a string is denoted by 1.  If the
 * string to be located is not found, 0 is returned.
 * <p><strong>Warning:</strong> the order of the parameters
 * of this method is reversed compared to the corresponding
 * function in JPQL.
 * @param x  expression for string to be searched
 * @param pattern  expression for string to be located
 * @return expression corresponding to position
 */
@Nonnull
Expression<Integer> locate(@Nonnull Expression<String> x,
                           @Nonnull Expression<String> pattern);

/**
 * Create expression to locate the position of one string
 * within another, returning position of first character
 * if found.
 * The first position in a string is denoted by 1.  If the
 * string to be located is not found, 0 is returned.
 * <p><strong>Warning:</strong> the order of the parameters
 * of this method is reversed compared to the corresponding
 * function in JPQL.
 * @param x  expression for string to be searched
 * @param pattern  string to be located
 * @return expression corresponding to position
 */
@Nonnull
Expression<Integer> locate(@Nonnull Expression<String> x,
                           @Nonnull String pattern);

/**
 * Create expression to locate the position of one string
 * within another, returning position of first character
 * if found.
 * The first position in a string is denoted by 1.  If the
 * string to be located is not found, 0 is returned.
 * <p><strong>Warning:</strong> the order of the first two
 * parameters of this method is reversed compared to the
 * corresponding function in JPQL.
 * @param x  expression for string to be searched
 * @param pattern  expression for string to be located
 * @param from  expression for position at which to start search
 * @return expression corresponding to position
 */
@Nonnull
Expression<Integer> locate(@Nonnull Expression<String> x,
                           @Nonnull Expression<String> pattern,
                           @Nonnull Expression<Integer> from);

/**
 * Create expression to locate the position of one string
 * within another, returning position of first character
 * if found.
 * The first position in a string is denoted by 1.  If the
 * string to be located is not found, 0 is returned.
 * <p><strong>Warning:</strong> the order of the first two
```

```
 * parameters of this method is reversed compared to the
 * corresponding function in JPQL.
 * @param x  expression for string to be searched
 * @param pattern  string to be located
 * @param from  position at which to start search
 * @return expression corresponding to position
 */
@Nonnull
Expression<Integer> locate(@Nonnull Expression<String> x,
                           @Nonnull String pattern,
                           int from);


// Date/time/timestamp functions:

/**
 * Create expression to return current date.
 * @return expression for current date
 */
@Nonnull
Expression<java.sql.Date> currentDate();

/**
 * Create expression to return current timestamp.
 * @return expression for current timestamp
 */
@Nonnull
Expression<java.sql.Timestamp> currentTimestamp();

/**
 * Create expression to return current time.
 * @return expression for current time
 */
@Nonnull
Expression<java.sql.Time> currentTime();

/**
 * Create expression to return current local date.
 * @return expression for current date
 */
@Nonnull
Expression<java.time.LocalDate> localDate();

/**
 * Create expression to return current local datetime.
 * @return expression for current timestamp
 */
@Nonnull
Expression<java.time.LocalDateTime> localDateTime();

/**
 * Create expression to return current local time.
 * @return expression for current time
 */
@Nonnull
Expression<java.time.LocalTime> localTime();

/**
 * Create an expression that returns the value of a
 * field extracted from a date, time, or datetime.
 * @param field a temporal field type
 * @param temporal a date, time, or datetime
 * @param <N> the type of the extracted value
 * @param <T> the temporal type
 * @return expression for the value of the extracted field
 * @since 3.2
 */
```

```
@Nonnull
<N,T extends Temporal> Expression<N> extract(@Nonnull TemporalField<N,T> field,
                                             @Nonnull Expression<T> temporal);


//in builders:

/**
 * Interface used to build in predicates.
 *
 * @param <T> the type of the tested expression
 */
interface In<T> extends Predicate {

     /**
      * Return the expression to be tested against the
      * list of values.
      * @return expression
      */
     @Nonnull
     Expression<T> getExpression();

     /**
      * Add to list of values to be tested against.
      * @param value value
      * @return in predicate
      */
     @Nonnull
     In<T> value(T value);

     /**
      * Add to list of values to be tested against.
      * @param value expression
      * @return in predicate
      */
     @Nonnull
     In<T> value(@Nonnull Expression<? extends T> value);
 }

/**
 * Create predicate to test whether given expression
 * is contained in a list of values.
 * @param  expression to be tested against list of values
 * @param <T> the type of the expression
 * @return  in predicate
 */
@Nonnull
<T> In<T> in(@Nonnull Expression<? extends T> expression);


// coalesce, nullif:

/**
 * Create an expression that returns null if all its arguments
 * evaluate to null, and the value of the first non-null argument
 * otherwise.
 * @param x expression
 * @param y expression
 * @param <Y> the type of the expression
 * @return coalesce expression
 */
@Nonnull
<Y> Expression<Y> coalesce(@Nonnull Expression<? extends Y> x,
                           @Nonnull Expression<? extends Y> y);

/**
 * Create an expression that returns null if all its arguments
```

```
 * evaluate to null, and the value of the first non-null argument
 * otherwise.
 * @param x expression
 * @param y value
 * @param <Y> the type of the expression
 * @return coalesce expression
 */
@Nonnull
<Y> Expression<Y> coalesce(@Nonnull Expression<? extends Y> x,
                           Y y);

/**
 * Create an expression that tests whether its argument are
 * equal, returning null if they are and the value of the
 * first expression if they are not.
 * @param x expression
 * @param y expression
 * @param <Y> the type of the expression
 * @return nullif expression
 */
@Nonnull
<Y> Expression<Y> nullif(@Nonnull Expression<Y> x,
                         @Nonnull Expression<?> y);

/**
 * Create an expression that tests whether its argument are
 * equal, returning null if they are and the value of the
 * first expression if they are not.
 * @param x expression
 * @param y value
 * @param <Y> the type of the expression
 * @return nullif expression
 */
@Nonnull
<Y> Expression<Y> nullif(@Nonnull Expression<Y> x,
                         Y y);


// coalesce builder:

/**
 * Interface used to build coalesce expressions.
 * <p>
 * A coalesce expression is equivalent to a case expression
 * that returns null if all its arguments evaluate to null,
 * and the value of its first non-null argument otherwise.
 *
 * @param <T> the type of the coalesce expression
 */
interface Coalesce<T> extends Expression<T> {

     /**
      * Add an argument to the coalesce expression.
      * @param value  value
      * @return coalesce expression
      */
     @Nonnull
     Coalesce<T> value(T value);

     /**
      * Add an argument to the coalesce expression.
      * @param value expression
      * @return coalesce expression
      */
     @Nonnull
     Coalesce<T> value(@Nonnull Expression<? extends T> value);
}
```

```
/**
 * Create a coalesce expression.
 * @param <T> the type of the coalesce expression
 * @return coalesce expression
 */
@Nonnull
<T> Coalesce<T> coalesce();


//case builders:

/**
 * Interface used to build simple case expressions.
 * Case conditions are evaluated in the order in which
 * they are specified.
 *
 * @param <C> the type of the case expression
 * @param <R> the result type of the case expression
 */
interface SimpleCase<C,R> extends Expression<R> {

    /**
     * Return the expression to be tested against the
     * conditions.
     * @return expression
     */
    @Nonnull
    Expression<C> getExpression();

    /**
     * Add a when/then clause to the case expression.
     * @param condition  "when" condition
     * @param result  "then" result value
     * @return simple case expression
     */
    @Nonnull
    SimpleCase<C, R> when(C condition,
                          R result);

    /**
     * Add a when/then clause to the case expression.
     * @param condition  "when" condition
     * @param result  "then" result expression
     * @return simple case expression
     */
    @Nonnull
    SimpleCase<C, R> when(C condition,
                          @Nonnull Expression<? extends R> result);

    /**
     * Add a when/then clause to the case expression.
     * @param condition  "when" condition
     * @param result  "then" result value
     * @return simple case expression
     */
    @Nonnull
    SimpleCase<C, R> when(@Nonnull Expression<? extends C> condition,
                          R result);

    /**
     * Add a when/then clause to the case expression.
     * @param condition  "when" condition
     * @param result  "then" result expression
     * @return simple case expression
     */
    @Nonnull
```

```
    SimpleCase<C, R> when(@Nonnull Expression<? extends C> condition,
                          @Nonnull Expression<? extends R> result);

    /**
     * Add an "else" clause to the case expression.
     * @param result  "else" result
     * @return expression
     */
    @Nonnull
    Expression<R> otherwise(R result);

    /**
     * Add an "else" clause to the case expression.
     * @param result  "else" result expression
     * @return expression
     */
    @Nonnull
    Expression<R> otherwise(@Nonnull Expression<? extends R> result);
}

/**
 * Create a simple case expression.
 * @param expression  to be tested against the case conditions
 * @param type the type of the result of the case expression
 * @return simple case expression
 */
@Nonnull
<C, R> SimpleCase<C,R> selectCase(@Nonnull Expression<? extends C> expression,
                                  @Nonnull Class<R> type);

/**
 * Create a simple case expression.
 * @param expression  to be tested against the case conditions
 * @param <C> the type of the case expression
 * @param <R> the result type of the case expression
 * @return simple case expression
 * @deprecated This operation declares an unconstrained type variable.
 *             Use {@link #selectCase(Expression,Class)} instead.
 */
@Deprecated(since = "4.0")
@Nonnull
<C, R> SimpleCase<C,R> selectCase(@Nonnull Expression<? extends C> expression);

/**
 * Interface used to build general case expressions.
 * Case conditions are evaluated in the order in which
 * they are specified.
 *
 * @param <R> the result type of the case expression
 */
interface Case<R> extends Expression<R> {

    /**
     * Add a when/then clause to the case expression.
     * @param condition  "when" condition
     * @param result  "then" result value
     * @return general case expression
     */
    @Nonnull
    Case<R> when(@Nonnull Expression<Boolean> condition,
                 R result);

    /**
     * Add a when/then clause to the case expression.
     * @param condition  "when" condition
     * @param result  "then" result expression
     * @return general case expression
```

```
     */
    @Nonnull
    Case<R> when(@Nonnull Expression<Boolean> condition,
                 @Nonnull Expression<? extends R> result);

    /**
     * Add an "else" clause to the case expression.
     * @param result  "else" result
     * @return expression
     */
    @Nonnull
    Expression<R> otherwise(R result);

    /**
     * Add an "else" clause to the case expression.
     * @param result  "else" result expression
     * @return expression
     */
    @Nonnull
    Expression<R> otherwise(@Nonnull Expression<? extends R> result);
}

/**
 * Create a general case expression.
 * @param type the type of the result of the case expression
 * @return general case expression
 */
@Nonnull
<R> Case<R> selectCase(@Nonnull Class<R> type);

/**
 * Create a general case expression.
 * @param <R> the result type of the case expression
 * @return general case expression
 * @deprecated This operation declares an unconstrained type variable.
 *             Use {@link #selectCase(Class)} instead.
 */
@Deprecated(since = "4.0")
@Nonnull
<R> Case<R> selectCase();

/**
 * Create an expression for the execution of the database
 * function with the given name.
 *
 * @apiNote This operation allows invocation of arbitrary
 * SQL functions. The persistence provider is permitted to
 * pass the given function name verbatim to the database,
 * with no additional validation or sanitization. Therefore,
 * the client must never pass unvalidated user input nor
 * any other untrusted string value to the first parameter
 * of this method.
 *
 * @param name  function name
 * @param type  expected result type
 * @param args  function arguments
 * @param <T> the function result type
 * @return expression
 */
@Nonnull
<T> Expression<T> function(@Nonnull String name, @Nonnull Class<T> type,
                           @Nonnull Expression<?>... args);


// methods for downcasting:

/**
```

```
 * Downcast Join object to the specified type.
 * @param join  Join object
 * @param type type to be downcast to
 * @param <X> the source type of the join
 * @param <T> the target type of the join
 * @param <V> the subtype to treat the join as
 * @return  Join object of the specified type
 * @since 2.1
 */
@Nonnull
<X, T, V extends T> Join<X, V> treat(@Nonnull Join<X, T> join,
                                     @Nonnull Class<V> type);

/**
 * Downcast CollectionJoin object to the specified type.
 * @param join  CollectionJoin object
 * @param type type to be downcast to
 * @param <X> the source type of the join
 * @param <T> the element type of the join
 * @param <E> the subtype to treat the join as
 * @return  CollectionJoin object of the specified type
 * @since 2.1
 */
@Nonnull
<X, T, E extends T> CollectionJoin<X, E> treat(@Nonnull CollectionJoin<X, T> join,
                                               @Nonnull Class<E> type);

/**
 * Downcast SetJoin object to the specified type.
 * @param join  SetJoin object
 * @param type type to be downcast to
 * @param <X> the source type of the join
 * @param <T> the element type of the join
 * @param <E> the subtype to treat the join as
 * @return  SetJoin object of the specified type
 * @since 2.1
 */
@Nonnull
<X, T, E extends T> SetJoin<X, E> treat(@Nonnull SetJoin<X, T> join,
                                        @Nonnull Class<E> type);

/**
 * Downcast ListJoin object to the specified type.
 * @param join  ListJoin object
 * @param type type to be downcast to
 * @param <X> the source type of the join
 * @param <T> the element type of the join
 * @param <E> the subtype to treat the join as
 * @return  ListJoin object of the specified type
 * @since 2.1
 */
@Nonnull
<X, T, E extends T> ListJoin<X, E> treat(@Nonnull ListJoin<X, T> join,
                                         @Nonnull Class<E> type);

/**
 * Downcast MapJoin object to the specified type.
 * @param join  MapJoin object
 * @param type type to be downcast to
 * @param <X> the source type of the join
 * @param <K> the key type of the join
 * @param <T> the value type of the join
 * @param <V> the subtype to treat the join as
 * @return  MapJoin object of the specified type
 * @since 2.1
 */
@Nonnull
```

```
<X, K, T, V extends T> MapJoin<X, K, V> treat(@Nonnull MapJoin<X, K, T> join,
                                              @Nonnull Class<V> type);


/**
 * Downcast Path object to the specified type.
 * @param path  path
 * @param type type to be downcast to
 * @param <X> the type of the path
 * @param <T> the subtype to treat the path as
 * @return  Path object of the specified type
 * @since 2.1
 */
@Nonnull
<X, T extends X> Path<T> treat(@Nonnull Path<X> path,
                               @Nonnull Class<T> type);

/**
 * Downcast Root or Join to the specified type.
 * @param from  root or join
 * @param type type to be downcast to
 * @param <X> the source type
 * @param <Y> the target type
 * @param <T> the subtype to treat the root or join as
 * @return  Root or Join of the specified type
 * @since 4.0
 */
@Nonnull
<X, Y, T extends Y> From<X, T> treat(@Nonnull From<X, Y> from,
                                     @Nonnull Class<T> type);

/**
 * Downcast Root object to the specified type.
 * @param root  root
 * @param type type to be downcast to
 * @param <X> the type of the root
 * @param <T> the subtype to treat the root as
 * @return  Root object of the specified type
 * @since 2.1
 */
@Nonnull
<X, T extends X> Root<T> treat(@Nonnull Root<X> root,
                               @Nonnull Class<T> type);

/**
 * Create a query which is the union of the given queries.
 * @param <T> the result type of the query
 * @return a new criteria query which returns the union of
 *         the results of the given queries
 * @since 3.2
 */
@Nonnull
<T> CriteriaSelect<T> union(@Nonnull CriteriaSelect<? extends T> left,
                            @Nonnull CriteriaSelect<? extends T> right);

/**
 * Create a query which is the union of the given queries,
 * without elimination of duplicate results.
 * @param <T> the result type of the query
 * @return a new criteria query which returns the union of
 *         the results of the given queries
 * @since 3.2
 */
@Nonnull
<T> CriteriaSelect<T> unionAll(@Nonnull CriteriaSelect<? extends T> left,
                               @Nonnull CriteriaSelect<? extends T> right);
```

```
    /**
     * Create a query which is the intersection of the given queries.
     * @param <T> the result type of the query
     * @return a new criteria query which returns the intersection of
     *         the results of the given queries
     * @since 3.2
     */
    @Nonnull
    <T> CriteriaSelect<T> intersect(@Nonnull CriteriaSelect<? super T> left,
                                    @Nonnull CriteriaSelect<? super T> right);

    /**
     * Create a query which is the intersection of the given queries,
     * without elimination of duplicate results.
     * @param <T> the result type of the query
     * @return a new criteria query which returns the intersection of
     *         the results of the given queries
     * @since 3.2
     */
    @Nonnull
    <T> CriteriaSelect<T> intersectAll(@Nonnull CriteriaSelect<? super T> left,
                                       @Nonnull CriteriaSelect<? super T> right);

    /**
     * Create a query by (setwise) subtraction of the second query
     * from the first query.
     * @param <T> the result type of the query
     * @return a new criteria query which returns the result of
     *         subtracting the results of the second query from the
     *         results of the first query
     * @since 3.2
     */
    @Nonnull
    <T> CriteriaSelect<T> except(@Nonnull CriteriaSelect<T> left,
                                 @Nonnull CriteriaSelect<?> right);

    /**
     * Create a query by (setwise) subtraction of the second query
     * from the first query, without elimination of duplicate results.
     * @param <T> the result type of the query
     * @return a new criteria query which returns the result of
     *         subtracting the results of the second query from the
     *         results of the first query
     * @since 3.2
     */
    @Nonnull
    <T> CriteriaSelect<T> exceptAll(@Nonnull CriteriaSelect<T> left,
                                    @Nonnull CriteriaSelect<?> right);
}
```

C.2. CriteriaQuery

```
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import jakarta.persistence.Tuple;

import java.util.List;

/**
 * The {@code CriteriaQuery} interface defines functionality that is
 * specific to top-level queries.
 *
 * @param <T>  the type of the defined result
 *
```

```
 * @since 2.0
 */
public interface CriteriaQuery<T> extends AbstractQuery<T>, CriteriaSelect<T> {

    /**
     * Specify the item that is to be returned in the query result.
     * Replaces the previously specified selection(s), if any.
     *
     * <p> Note: Applications using the string-based API may need to
     * specify the type of the select item when it results from
     * a get or join operation and the query result type is
     * specified.
     *
     * <p>For example:
     * {@snippet :
     * CriteriaQuery<String> q = cb.createQuery(String.class);
     * Root<Order> order = q.from(Order.class);
     * q.select(order.get("shippingAddress").<String>get("state"));
     *
     * CriteriaQuery<Product> q2 = cb.createQuery(Product.class);
     * q2.select(q2.from(Order.class)
     *             .join("items")
     *             .<Item, Product>join("product"));
     * }
     *
     * @param selection  selection specifying the item that is
     *                   to be returned in the query result
     * @return the modified query
     * @throws IllegalArgumentException if the selection is
     *         a compound selection and more than one selection
     *         item has the same assigned alias
     */
    @Nonnull
    CriteriaQuery<T> select(@Nonnull Selection<? extends T> selection);

    /**
     * Specify the selection items that are to be returned in the query result.
     * Replaces the previously specified selection(s), if any.
     *
     * <p> The type of the result of the query execution depends on the specification
     * of the type of the criteria query object created as well as the arguments
     * to the {@code multiselect} method.
     *
     * <p> An argument to the multiselect method must not be a tuple- or array-valued
     * compound selection item.
     *
     * <p>The semantics of this method are as follows:
     * <ul>
     * <li>
     * If the type of the criteria query is {@code CriteriaQuery<Tuple>}
     * (i.e., a criteria query object created by either the {@code createTupleQuery}
     * method or by passing a {@link Tuple} class argument to the {@code createQuery}
     * method), a {@link Tuple} object corresponding to the arguments of the
     * {@code multiselect} method, in the specified order, will be instantiated and
     * returned for each row that results from the query execution.
     *
     * <li> If the type of the criteria query is {@code CriteriaQuery<X>}
     * for some user-defined class X (i.e., a criteria query object created by
     * passing a X class argument to the {@code createQuery} method), the arguments
     * to the {@code multiselect} method will be passed to the X constructor and an
     * instance of type X will be returned for each row.
     *
     * <li> If the type of the criteria query is {@code CriteriaQuery<X[]>}
     * for some class X, an instance of type {@code X[]} will be returned for each row.
     * The elements of the array will correspond to the arguments of the
     * {@code multiselect} method, in the specified order.
     *
```

```
 * <li> If the type of the criteria query is {@code CriteriaQuery<Object>}
 * or if the criteria query was created without specifying a type, and only a single
 * argument is passed to the {@code multiselect} method, an instance of type
 * {@code Object} will be returned for each row.
 *
 * <li> If the type of the criteria query is {@code CriteriaQuery<Object>}
 * or if the criteria query was created without specifying a type, and more than one
 * argument is passed to the {@code multiselect} method, an instance of type
 * {@code Object[]} will be instantiated and returned for each row.  The elements of
 * the array will correspond to the arguments to the {@code multiselect} method, in
 * the specified order.
 * </ul>
 *
 * @param selections  selection items corresponding to the
 *                    results to be returned by the query
 * @return the modified query
 * @throws IllegalArgumentException if a selection item is
 *         not valid or if more than one selection item has
 *         the same assigned alias
 *
 * @deprecated Since this method is not typesafe, the use of
 * {@link CriteriaBuilder#array} or {@link CriteriaBuilder#tuple}
 * with {@link #select} is strongly preferred.
 */
@Deprecated(since = "3.2")
@Nonnull
CriteriaQuery<T> multiselect(@Nonnull Selection<?>... selections);


/**
 * Specify the selection items that are to be returned in the query result.
 * Replaces the previously specified selection(s), if any.
 *
 * <p> The type of the result of the query execution depends on the specification
 * of the type of the criteria query object created as well as the argument to the
 * {@code multiselect} method. An element of the list passed to the {@code multiselect}
 * method must not be a tuple- or array-valued compound selection item.
 *
 * <p> The semantics of this method are as follows:
 * <ul>
 * <li> If the type of the criteria query is {@code CriteriaQuery<Tuple>}
 * (i.e., a criteria query object created by either the {@code createTupleQuery}
 * method or by passing a {@link Tuple} class argument to the {@code createQuery}
 * method), a {@code Tuple} object corresponding to the elements of the list passed
 * to the {@code multiselect} method, in the specified order, will be instantiated
 * and returned for each row that results from the query execution.
 *
 * <li> If the type of the criteria query is {@code CriteriaQuery<X>}
 * for some user-defined class X (i.e., a criteria query object created by passing
 * a X class argument to the {@code createQuery} method), the elements of the list
 * passed to the {@code multiselect} method will be passed to the X constructor
 * and an instance of type X will be returned for each row.
 *
 * <li> If the type of the criteria query is {@code CriteriaQuery<X[]>}
 * for some class X, an instance of type {@code X[]} will be returned for
 * each row. The elements of the array will correspond to the elements of
 * the list passed to the {@code multiselect} method, in the specified order.
 *
 * <li> If the type of the criteria query is {@code CriteriaQuery<Object>}
 * or if the criteria query was created without specifying a type, and the list
 * passed to the {@code multiselect} method contains only a single element, an
 * instance of type {@code Object} will be returned for each row.
 *
 * <li> If the type of the criteria query is {@code CriteriaQuery<Object>}
 * or if the criteria query was created without specifying a type, and the list
 * passed to the {@code multiselect} method contains more than one element, an
 * instance of type {@code Object[]} will be instantiated and returned for each row.
```

```
 * The elements of the array will correspond to the elements of the list passed to
 * the {@code multiselect} method, in the specified order.
 * </ul>
 *
 * @param selectionList  list of selection items corresponding
 *                       to the results to be returned by the
 *                       query
 * @return the modified query
 * @throws IllegalArgumentException if a selection item is
 *         not valid or if more than one selection item has
 *         the same assigned alias
 *
 * @deprecated Since this method is not typesafe, the use of
 * {@link CriteriaBuilder#array} or {@link CriteriaBuilder#tuple}
 * with {@link #select} is strongly preferred.
 */
@Deprecated(since = "3.2")
@Nonnull
CriteriaQuery<T> multiselect(@Nonnull List<Selection<?>> selectionList);

/**
 * Modify the query to restrict the query result according
 * to the specified boolean expression.
 * Replaces the previously added restriction(s), if any.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restriction  a simple or compound boolean expression
 * @return the modified query
 */
@Nonnull
CriteriaQuery<T> where(@Nonnull Expression<Boolean> restriction);

/**
 * Modify the query to restrict the query result according
 * to the conjunction of the specified restriction predicates.
 * Replaces the previously added restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restrictions  zero or more restriction predicates
 * @return the modified query
 */
@Nonnull
CriteriaQuery<T> where(@Nonnull BooleanExpression... restrictions);

/**
 * Modify the query to restrict the query result according
 * to the conjunction of the specified restriction predicates.
 * Replaces the previously added restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restrictions  a list of zero or more restriction predicates
 * @return the modified query
 * @since 3.2
 */
@Nonnull
CriteriaQuery<T> where(@Nonnull List<? extends Expression<Boolean>> restrictions);

/**
 * Specify the expressions that are used to form groups over
 * the query results.
 * Replaces the previous specified grouping expressions, if any.
 * If no grouping expressions are specified, any previously
 * added grouping expressions are simply removed.
```

```
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param grouping  zero or more grouping expressions
 * @return the modified query
 */
@Nonnull
CriteriaQuery<T> groupBy(@Nonnull Expression<?>... grouping);

/**
 * Specify the expressions that are used to form groups over
 * the query results.
 * Replaces the previous specified grouping expressions, if any.
 * If no grouping expressions are specified, any previously
 * added grouping expressions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param grouping  list of zero or more grouping expressions
 * @return the modified query
 */
@Nonnull
CriteriaQuery<T> groupBy(@Nonnull List<Expression<?>> grouping);

/**
 * Specify a restriction over the groups of the query.
 * Replaces the previous having restriction(s), if any.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restriction  a simple or compound boolean expression
 * @return the modified query
 */
@Nonnull
CriteriaQuery<T> having(@Nonnull Expression<Boolean> restriction);

/**
 * Specify restrictions over the groups of the query
 * according the conjunction of the specified restriction
 * predicates.
 * Replaces the previously added having restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restrictions  zero or more restriction predicates
 * @return the modified query
 */
@Nonnull
CriteriaQuery<T> having(@Nonnull BooleanExpression... restrictions);

/**
 * Specify restrictions over the groups of the query
 * according the conjunction of the specified restriction
 * predicates.
 * Replaces the previously added having restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restrictions  a list of zero or more restriction predicates
 * @return the modified query
 * @since 3.2
 */
@Nonnull
CriteriaQuery<T> having(@Nonnull List<? extends Expression<Boolean>> restrictions);

/**
 * Specify the ordering expressions that are used to
 * order the query results.
```

```
     * Replaces the previous ordering expressions, if any.
     * If no ordering expressions are specified, the previous
     * ordering, if any, is simply removed, and results will
     * be returned in no particular order.
     * The left-to-right sequence of the ordering expressions
     * determines the precedence, whereby the leftmost has the
     * highest precedence.
     * @param o  zero or more ordering expressions
     * @return the modified query
     */
    @Nonnull
    CriteriaQuery<T> orderBy(@Nonnull Order... o);

    /**
     * Specify the ordering expressions that are used to
     * order the query results.
     * Replaces the previous ordering expressions, if any.
     * If no ordering expressions are specified, the previous
     * ordering, if any, is simply removed, and results will
     * be returned in no particular order.
     * The order of the ordering expressions in the list
     * determines the precedence, whereby the first element in
     * the list has the highest precedence.
     * @param o  list of zero or more ordering expressions
     * @return the modified query
     */
    @Nonnull
    CriteriaQuery<T> orderBy(@Nonnull List<Order> o);

    /**
     * Specify whether duplicate query results are eliminated.
     * A true value will cause duplicates to be eliminated.
     * A false value will cause duplicates to be retained.
     * If distinct has not been specified, duplicate results must
     * be retained.
     * This method only overrides the return type of the
     * corresponding {@code AbstractQuery} method.
     * @param distinct  boolean value specifying whether duplicate
     *        results must be eliminated from the query result or
     *        whether they must be retained
     * @return the modified query.
     */
    @Nonnull
    CriteriaQuery<T> distinct(boolean distinct);

    /**
     * Return the ordering expressions in order of precedence.
     * Returns empty list if no ordering expressions have been
     * specified.
     * Modifications to the list do not affect the query.
     * @return the list of ordering expressions
     */
    @Nonnull
    List<Order> getOrderList();
}
```

C.3. CriteriaSelect

```
/**
 * Abstracts over {@linkplain CriteriaQuery top-level queries} and
 * {@linkplain CriteriaBuilder#union unions} and
 * {@linkplain CriteriaBuilder#intersect intersections} of top-level
 * queries.
 *
 * @param <T>  the type returned by the query
```

```
 *
 * @since 3.2
 */
public interface CriteriaSelect<T> {
}
```

C.4. CriteriaStatement

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.EntityType;

/**
 * Declares operations common to {@link CriteriaUpdate} and
 * {@link CriteriaDelete}. A {@code CriteriaStatement} represents
 * a bulk operation affecting a certain entity type.
 *
 * @see CriteriaUpdate
 * @see CriteriaDelete
 * @see jakarta.persistence.EntityHandler#createStatement(CriteriaStatement)
 *
 * @param <T> The affected entity type
 * @since 4.0
 */
public interface CriteriaStatement<T> extends CommonAbstractCriteria {

    /**
     * Create and add a statement root corresponding to the entity
     * that is affected by the statement.
     * A statement has a single root, the entity that is affected.
     * @param entityClass  the entity class
     * @return query root corresponding to the given entity
     */
    @Nonnull
    Root<T> from(@Nonnull Class<T> entityClass);

    /**
     * Create and add a statement root corresponding to the entity
     * that is affected by the statement.
     * A statement has a single root, the entity that is affected.
     * @param entity  metamodel object representing the entity
     * @return query root corresponding to the given entity
     */
    @Nonnull
    Root<T> from(@Nonnull EntityType<T> entity);

    /**
     * Return the statement root.
     * @return the statement root
     */
    @Nullable
    Root<T> getRoot();

}
```

C.5. CriteriaUpdate

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.SingularAttribute;

import java.util.List;
```

```java
/**
 * The {@code CriteriaUpdate} interface defines functionality for
 * performing bulk update operations using the Criteria API.
 *
 * <p>Criteria API bulk update operations map directly to database
 * update operations, bypassing any optimistic locking checks.
 * Portable applications using bulk update operations must manually
 * update the value of the version column, if desired, and/or manually
 * validate the value of the version column. The persistence context
 * is not automatically synchronized with the result of the bulk update.
 *
 * <p> A {@code CriteriaUpdate} object must have a single root.
 *
 * @param <T>  the entity type that is the target of the update
 *
 * @since 2.1
 */
public interface CriteriaUpdate<T> extends CriteriaStatement<T> {

    /**
     * Update the value of the specified attribute.
     * @param attribute the attribute to be updated
     * @param value the new value to set
     * @param <Y> the type of the attribute
     * @param <X> the type of the value
     * @return the modified update query
     */
    @Nonnull
    <Y, X extends Y> CriteriaUpdate<T> set(@Nonnull SingularAttribute<? super T, Y> attribute,
                                           X value);

    /**
     * Update the value of the specified attribute.
     * @param attribute the attribute to be updated
     * @param value the new value to set
     * @param <Y> the type of the attribute
     * @return the modified update query
     */
    @Nonnull
    <Y> CriteriaUpdate<T> set(@Nonnull SingularAttribute<? super T, Y> attribute,
                              @Nonnull Expression<? extends Y> value);

    /**
     * Update the value of the specified attribute.
     * @param attribute the attribute to be updated
     * @param value the new value to set
     * @param <Y> the type of the attribute
     * @param <X> the type of the value
     * @return the modified update query
     */
    @Nonnull
    <Y, X extends Y> CriteriaUpdate<T> set(@Nonnull Path<Y> attribute,
                                           X value);

    /**
     * Update the value of the specified attribute.
     * @param attribute the attribute to be updated
     * @param value the new value to set
     * @param <Y> the type of the attribute
     * @return the modified update query
     */
    @Nonnull
    <Y> CriteriaUpdate<T> set(@Nonnull Path<Y> attribute,
                              @Nonnull Expression<? extends Y> value);

    /**
     * Update the value of the specified attribute.
```

```
     * @param attributeName the name of the attribute to be updated
     * @param value the new value to set
     * @return the modified update query
     * @throws IllegalArgumentException if there is no attribute
     *         with the given name
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #set(SingularAttribute,Expression)}.
     *          Use of the typesafe version is strongly preferred.
     */
    @Nonnull
    CriteriaUpdate<T> set(@Nonnull String attributeName, Object value);

    /**
     * Modify the update query to restrict the target of the
     * update according to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified update query
     */
    @Nonnull
    CriteriaUpdate<T> where(@Nonnull Expression<Boolean> restriction);

    /**
     * Modify the update query to restrict the target of the
     * update according to the conjunction of the specified
     * restriction predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified update query
     */
    @Nonnull
    CriteriaUpdate<T> where(@Nonnull BooleanExpression... restrictions);

    /**
     * Modify the update query to restrict the target of the
     * update according to the conjunction of the specified
     * restriction predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified update query
     * @since 4.0
     */
    @Nonnull
    CriteriaUpdate<T> where(@Nonnull List<? extends Expression<Boolean>> restrictions);
}
```

C.6. CriteriaDelete

```
import jakarta.annotation.Nonnull;

import java.util.List;

/**
 * The {@code CriteriaDelete} interface defines functionality for
 * performing bulk delete operations using the Criteria API
 *
 * <p>Criteria API bulk delete operations map directly to database
 * delete operations. The persistence context is not synchronized
 * with the result of the bulk delete.
```

```
 *
 * <p> A {@code CriteriaDelete} object must have a single root.
 *
 * @param <T>  the entity type that is the target of the DELETE
 *
 * @since 2.1
 */
public interface CriteriaDelete<T> extends CriteriaStatement<T> {

    /**
     * Modify the DELETE query to restrict the target of the deletion
     * according to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified delete query
     */
    @Nonnull
    CriteriaDelete<T> where(@Nonnull Expression<Boolean> restriction);

    /**
     * Modify the DELETE query to restrict the target of the deletion
     * according to the conjunction of the specified restriction
     * predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified delete query
     */
    @Nonnull
    CriteriaDelete<T> where(@Nonnull BooleanExpression... restrictions);

    /**
     * Modify the DELETE query to restrict the target of the deletion
     * according to the conjunction of the specified restriction
     * predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified delete query
     * @since 4.0
     */
    @Nonnull
    CriteriaDelete<T> where(@Nonnull List<? extends Expression<Boolean>> restrictions);
}
```

C.7. AbstractQuery

```
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import java.util.List;
import java.util.Set;
import jakarta.persistence.metamodel.EntityType;

/**
 * The {@code AbstractQuery} interface defines functionality that is common
 * to both top-level queries and subqueries.
 * It is not intended to be used directly in query construction.
 *
 * <p> All queries must have:
 *         a set of root entities (which may in turn own joins).
 * <p> All queries may have:
```

```
 *         a conjunction of restrictions.
 *
 * @param <T>  the type of the result
 *
 * @since 2.0
 */
public interface AbstractQuery<T> extends CommonAbstractCriteria {

    /**
     * Create and add a query root corresponding to the given entity,
     * forming a cartesian product with any existing roots.
     * @param entityClass  the entity class
     * @param <X> the entity type
     * @return query root corresponding to the given entity
     */
    @Nonnull
    <X> Root<X> from(@Nonnull Class<X> entityClass);

    /**
     * Create and add a query root corresponding to the given entity,
     * forming a cartesian product with any existing roots.
     * @param entity  metamodel entity representing the entity
     *                of type X
     * @param <X> the entity type
     * @return query root corresponding to the given entity
     */
    @Nonnull
    <X> Root<X> from(@Nonnull EntityType<X> entity);

    /**
     * Modify the query to restrict the query results according
     * to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified query
     */
    @Nonnull
    AbstractQuery<T> where(@Nonnull Expression<Boolean> restriction);

    /**
     * Modify the query to restrict the query results according
     * to the conjunction of the specified restriction predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  zero or more restriction predicates
     * @return the modified query
     */
    @Nonnull
    AbstractQuery<T> where(@Nonnull BooleanExpression... restrictions);

    /**
     * Modify the query to restrict the query result according
     * to the conjunction of the specified restriction predicates.
     * Replaces the previously added restriction(s), if any.
     * If no restrictions are specified, any previously added
     * restrictions are simply removed.
     * @param restrictions  a list of zero or more restriction predicates
     * @return the modified query
     * @since 3.2
     */
    @Nonnull
    AbstractQuery<T> where(@Nonnull List<? extends Expression<Boolean>> restrictions);

    /**
     * Specify the expressions that are used to form groups over
     * the query results.
```

```
 * Replaces the previous specified grouping expressions, if any.
 * If no grouping expressions are specified, any previously
 * added grouping expressions are simply removed.
 * @param grouping  zero or more grouping expressions
 * @return the modified query
 */
@Nonnull
AbstractQuery<T> groupBy(@Nonnull Expression<?>... grouping);

/**
 * Specify the expressions that are used to form groups over
 * the query results.
 * Replaces the previous specified grouping expressions, if any.
 * If no grouping expressions are specified, any previously
 * added grouping expressions are simply removed.
 * @param grouping  list of zero or more grouping expressions
 * @return the modified query
 */
@Nonnull
AbstractQuery<T> groupBy(@Nonnull List<Expression<?>> grouping);

/**
 * Specify a restriction over the groups of the query.
 * Replaces the previous having restriction(s), if any.
 * @param restriction  a simple or compound boolean expression
 * @return the modified query
 */
@Nonnull
AbstractQuery<T> having(@Nonnull Expression<Boolean> restriction);

/**
 * Specify restrictions over the groups of the query
 * according the conjunction of the specified restriction
 * predicates.
 * Replaces the previously having added restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * @param restrictions  zero or more restriction predicates
 * @return the modified query
 */
@Nonnull
AbstractQuery<T> having(@Nonnull BooleanExpression... restrictions);

/**
 * Specify restrictions over the groups of the query
 * according the conjunction of the specified restriction
 * predicates.
 * Replaces the previously added having restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * @param restrictions  a list of zero or more restriction predicates
 * @return the modified query
 * @since 3.2
 */
@Nonnull
AbstractQuery<T> having(@Nonnull List<? extends Expression<Boolean>> restrictions);

/**
 * Specify whether duplicate query results are eliminated.
 * A true value will cause duplicates to be eliminated.
 * A false value will cause duplicates to be retained.
 * If distinct has not been specified, duplicate results must
 * be retained.
 * @param distinct  boolean value specifying whether duplicate
 *        results must be eliminated from the query result or
 *        whether they must be retained
 * @return the modified query
```

```
     */
    @Nonnull
    AbstractQuery<T> distinct(boolean distinct);

    /**
     * Return the query roots. These are the roots that are
     * defined for the {@link CriteriaQuery} or {@link Subquery}
     * itself, including any subquery roots defined as a result of
     * correlation. Returns an empty set if no roots have been
     * defined. Modifications to the set do not affect the query.
     * @return the set of query roots
     */
    @Nonnull
    Set<Root<?>> getRoots();

    /**
     * Return the selection of the query, or null if no selection
     * has been set.
     * @return selection item
     */
    @Nullable
    Selection<T> getSelection();

    /**
     * Return a list of the grouping expressions.  Returns empty
     * list if no grouping expressions have been specified.
     * Modifications to the list do not affect the query.
     * @return the list of grouping expressions
     */
    @Nonnull
    List<Expression<?>> getGroupList();

    /**
     * Return the predicate that corresponds to the restriction(s)
     * over the grouping items, or null if no restrictions have
     * been specified.
     * @return having clause predicate
     */
    @Nullable
    Predicate getGroupRestriction();

    /**
     * Return whether duplicate query results must be eliminated or
     * retained.
     * @return boolean indicating whether duplicate query results
     *         must be eliminated
     */
    boolean isDistinct();

    /**
     * Return the result type of the query or subquery. If
     * a result type was specified as an argument to the
     * {@code createQuery} or {@code subquery} method, that
     * type is returned. If the query was created using the
     * {@code createTupleQuery} method, the result type is
     * {@code Tuple}. Otherwise, the result type is
     * {@code Object}.
     * @return result type
     */
    @Nonnull
    Class<T> getResultType();
}
```

C.8. CollectionJoin

```java
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import java.util.Collection;
import java.util.List;

import jakarta.persistence.metamodel.CollectionAttribute;

/**
 * The {@code CollectionJoin} interface is the type of the result of
 * joining to a collection over an association or element
 * collection that has been specified as a {@link java.util.Collection}.
 *
 * @param <Z> the source type of the join
 * @param <E> the element type of the target {@code Collection}
 *
 * @since 2.0
 */
public interface CollectionJoin<Z, E>
        extends PluralJoin<Z, Collection<E>, E> {

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified join object
     * @since 2.1
     */
    @Override
    @Nonnull
    CollectionJoin<Z, E> on(@Nonnull Expression<Boolean> restriction);

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restrictions  zero or more restriction predicates
     * @return the modified join object
     * @since 2.1
     */
    @Override
    @Nonnull
    CollectionJoin<Z, E> on(@Nonnull BooleanExpression... restrictions);

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restrictions  zero or more restriction predicates
     * @return the modified join object
     * @since 4.0
     */
    @Override
    @Nonnull
    CollectionJoin<Z, E> on(@Nonnull List<? extends Expression<Boolean>> restrictions);

    /**
     * Return the metamodel representation for the collection
     * attribute.
     * @return metamodel type representing the {@code Collection} that is
     *         the target of the join
     */
    @Override
```

```
    @Nonnull
    CollectionAttribute<? super Z, E> getModel();

    /**
     * Downcast the element type to the given type.
     * @param type a subtype of the element type
     * @param <T> the subtype of the element type
     * @return this join downcast to the given element type
     * @since 4.0
     */
    @Override
    @Nonnull
    <T extends E> CollectionJoin<Z, T> treat(@Nonnull Class<T> type);
}
```

C.9. CommonAbstractCriteria

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.EntityType;

import java.util.Set;

/**
 * The {@code CommonAbstractCriteria} interface defines functionality
 * that is common to both top-level criteria queries and subqueries as
 * well as to update and delete criteria operations.
 * It is not intended to be used directly in query construction.
 *
 * <p> Note that criteria queries and criteria update and delete operations
 * are typed differently.
 * Criteria queries are typed according to the query result type.
 * Update and delete operations are typed according to the target of the
 * update or delete.
 *
 * @since 2.1
 */
public interface CommonAbstractCriteria {

    /**
     * Create a subquery of the query.
     * @param type  the subquery result type
     * @param <U> the subquery result type
     * @return subquery
     */
    @Nonnull
    <U> Subquery<U> subquery(@Nonnull Class<U> type);

    /**
     * Create a subquery of the query.
     * @param type  the subquery result type
     * @param <U> the subquery result type
     * @return subquery
     */
    @Nonnull
    <U> Subquery<U> subquery(@Nonnull EntityType<U> type);

    /**
     * Return the predicate that corresponds to the where clause
     * restriction(s), or null if no restrictions have been
     * specified.
     * @return where clause predicate
     */
    @Nullable
    Predicate getRestriction();
```

```
    /**
     * Return the parameters of the query.  Returns empty set if
     * there are no parameters.
     * Modifications to the set do not affect the query.
     * @return the query parameters
     */
    @Nonnull
    Set<ParameterExpression<?>> getParameters();
}
```

C.10. CompoundSelection

```
/**
 * The {@code CompoundSelection} interface defines a compound
 * selection item (a tuple, array, or result of a constructor).
 *
 * @param <X> the type of the selection item
 *
 * @since 2.0
 */
public interface CompoundSelection<X> extends Selection<X> {}
```

C.11. Expression

```
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import jakarta.persistence.criteria.CriteriaBuilder.SimpleCase;

import java.util.Collection;

/**
 * Type for query expressions.
 *
 * @param <T> the type of the expression
 *
 * @since 2.0
 */
public interface Expression<T> extends Selection<T> {

    /**
     * Create a predicate to test whether the expression is null.
     * @return predicate testing whether the expression is null
     */
    @Nonnull
    Predicate isNull();

    /**
     * Create a predicate to test whether the expression is
     * not null.
     * @return predicate testing whether the expression is not null
     */
    @Nonnull
    Predicate isNotNull();

    /**
     * Create a predicate to test whether the expression is equal to
     * the argument.
     * @param value  expression to be tested against
     * @return predicate testing for equality
     * @since 3.2
     */
```

```java
@Nonnull
Predicate equalTo(@Nonnull Expression<?> value);

/**
 * Create a predicate to test whether the expression is equal to
 * the argument.
 * @param value  value to be tested against
 * @return predicate testing for equality
 * @since 3.2
 */
@Nonnull
Predicate equalTo(Object value);

/**
 * Create a predicate to test whether the expression is unequal
 * to the argument.
 * @param value  expression to be tested against
 * @return predicate testing for inequality
 * @since 3.2
 */
@Nonnull
Predicate notEqualTo(@Nonnull Expression<?> value);

/**
 * Create a predicate to test whether the expression is unequal
 * to the argument.
 * @param value  value to be tested against
 * @return predicate testing for inequality
 * @since 3.2
 */
@Nonnull
Predicate notEqualTo(Object value);

/**
 * Create a predicate to test whether the expression is a member
 * of the argument list.
 * @param values  values to be tested against
 * @return predicate testing for membership
 */
@Nonnull
Predicate in(@Nonnull Object... values);

/**
 * Create a predicate to test whether the expression is a member
 * of the argument list.
 * @param values  expressions to be tested against
 * @return predicate testing for membership
 */
@Nonnull
Predicate in(@Nonnull Expression<?>... values);

/**
 * Create a predicate to test whether the expression is a member
 * of the collection.
 * @param values  collection of values to be tested against
 * @return predicate testing for membership
 */
@Nonnull
Predicate in(@Nonnull Collection<?> values);

/**
 * Create a predicate to test whether the expression is a member
 * of the collection.
 * @param values expression corresponding to collection to be
 *        tested against
 * @return predicate testing for membership
 */
```

```
@Nonnull
Predicate in(@Nonnull Expression<Collection<?>> values);

/**
 * Create a predicate to test whether the expression is returned
 * by the subquery.
 * @param subquery the subquery
 * @return predicate testing for membership
 * @since 4.0
 */
@Nonnull
Predicate in(@Nonnull Subquery<T> subquery);

/**
 * Perform a typecast upon the expression, returning a new
 * expression object.
 * Unlike {@link #cast(Class)}, this method does not cause
 * type conversion: the runtime type is not changed.
 * <p><em>Warning: may result in a runtime failure.</em>
 * @param type  intended type of the expression
 * @param <X> the intended type of the expression
 * @return new expression of the given type
 * @see #cast(Class)
 */
@Nonnull
<X> Expression<X> as(@Nonnull Class<X> type);

/**
 * Cast this expression to the specified type, returning a
 * new expression object.
 * Unlike {@link #as(Class)}, this method <em>does</em>
 * result in a runtime type conversion.
 * <p><em>Providers are required to support casting
 * scalar expressions to {@link String}, and
 * {@code String} expressions to {@link Integer},
 * {@link Long}, {@link Float}, and {@link Double}.
 * Support for typecasts between other basic types is
 * not required.</em>
 * @param type  a basic type
 * @param <X> the target type of the cast
 * @return a scalar expression of the given basic type
 * @since 3.2
 */
@Nonnull
<X> Expression<X> cast(@Nonnull Class<X> type);

// coalesce, nullif:

/**
 * Create an expression that returns null if this and the argument
 * evaluate to null, and the value of the first non-null expression
 * otherwise.
 * @param y expression
 * @return coalesce expression
 * @since 4.0
 */
@Nonnull
Expression<T> coalesce(@Nonnull Expression<? extends T> y);

/**
 * Create an expression that returns null if this and the argument
 * evaluate to null, and the value of the first non-null expression
 * otherwise.
 * @param y value
 * @return coalesce expression
 * @since 4.0
 */
```

```
@Nonnull
Expression<T> coalesce(T y);

/**
 * Create an expression that tests whether this expression
 * is equal to the argument, returning null if they are
 * and the value of the first expression if they are not.
 * @param y expression
 * @return nullif expression
 * @since 4.0
 */
@Nonnull
Expression<T> nullif(@Nonnull Expression<? extends T> y);

/**
 * Create an expression that tests whether this expression
 * is equal to the argument, returning null if they are
 * and the value of the first expression if they are not.
 * @param y value
 * @return nullif expression
 * @since 4.0
 */
@Nonnull
Expression<T> nullif(T y);

//case builders:

/**
 * Create a simple case expression to test against this expression.
 * @param type the type of the result of the case expression
 * @param <R> the result type of the case expression
 * @return simple case expression
 * @since 4.0
 */
@Nonnull
<R> SimpleCase<T,R> selectCase(@Nonnull Class<R> type);

//collection operations:

/**
 * Create a predicate that tests whether this expression is
 * a member of a collection.
 * If the collection is empty, the predicate will be false.
 * @param collection expression
 * @return is-member predicate
 * @since 4.0
 */
@Nonnull
Predicate isMember(@Nonnull Expression<? extends Collection<? super T>> collection);

/**
 * Create a predicate that tests whether this expression is
 * not a member of a collection.
 * If the collection is empty, the predicate will be true.
 * @param collection expression
 * @return is-not-member predicate
 * @since 4.0
 */
@Nonnull
Predicate isNotMember(@Nonnull Expression<? extends Collection<? super T>> collection);

//aggregate functions:

/**
 * Create an aggregate expression applying the count operation.
 * @return count expression
 * @since 4.0
```

```
     */
    @Nonnull
    NumericExpression<Long> count();

    /**
     * Create an aggregate expression applying the count distinct
     * operation.
     * @return count distinct expression
     * @since 4.0
     */
    @Nonnull
    NumericExpression<Long> countDistinct();

}
```

C.12. Fetch

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.Attribute;

/**
 * Represents a join-fetched association or attribute.
 *
 * @param <Z>  the source type of the fetch
 * @param <X>  the target type of the fetch
 *
 * @since 2.0
 */
public interface Fetch<Z, X> extends FetchParent<Z, X> {

    /**
     * Return the metamodel attribute corresponding to the
     * fetch join.
     * @return metamodel attribute for the join
     */
    @Nonnull
    Attribute<? super Z, ?> getAttribute();

    /**
     * Return the parent of the fetched item.
     * @return fetch parent
     */
    @Nonnull
    FetchParent<?, Z> getParent();

    /**
     * Return the join type used in the fetch join.
     * @return join type
     */
    @Nonnull
    JoinType getJoinType();
}
```

C.13. FetchParent

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.PluralAttribute;
import jakarta.persistence.metamodel.SingularAttribute;

/**
 * Represents an element of the from clause which may
 * function as the parent of Fetches.
 *
```

```
 * @param <Z>  the source type
 * @param <X>  the target type
 *
 * @since 2.0
 */
public interface FetchParent<Z, X> {

    /**
     * Return the fetch joins that have been made from this type.
     * Returns empty set if no fetch joins have been made from
     * this type.
     * Modifications to the set do not affect the query.
     * @return fetch joins made from this type
     */
    @Nonnull
    java.util.Set<Fetch<X, ?>> getFetches();

    /**
     * Create a fetch join to the specified single-valued attribute
     * using an inner join.
     * @param attribute  target of the join
     * @param <Y> the type of the fetched attribute
     * @return the resulting fetch join
     */
    @Nonnull
    <Y> Fetch<X, Y> fetch(@Nonnull SingularAttribute<? super X, Y> attribute);

    /**
     * Create a fetch join to the specified single-valued attribute
     * using the given join type.
     * @param attribute  target of the join
     * @param jt  join type
     * @param <Y> the type of the fetched attribute
     * @return the resulting fetch join
     */
    @Nonnull
    <Y> Fetch<X, Y> fetch(@Nonnull SingularAttribute<? super X, Y> attribute, @Nonnull JoinType jt);

    /**
     * Create a fetch join to the specified collection-valued
     * attribute using an inner join.
     * @param attribute  target of the join
     * @param <Y> the type of the fetched attribute
     * @return the resulting join
     */
    @Nonnull
    <Y> Fetch<X, Y> fetch(@Nonnull PluralAttribute<? super X, ?, Y> attribute);

    /**
     * Create a fetch join to the specified collection-valued
     * attribute using the given join type.
     * @param attribute  target of the join
     * @param jt  join type
     * @param <Y> the type of the fetched attribute
     * @return the resulting join
     */
    @Nonnull
    <Y> Fetch<X, Y> fetch(@Nonnull PluralAttribute<? super X, ?, Y> attribute, @Nonnull JoinType jt);


    //String-based:

    /**
     * Create a fetch join to the specified attribute using an
     * inner join.
     * @param attributeName the name of the attribute for the
     *        target of the join
```

```
     * @param <Y> the type of the fetched attribute
     * @return the resulting fetch join
     * @throws IllegalArgumentException if there is no attribute
     *         with the given name
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #fetch(SingularAttribute)}
     *          or {@link #fetch(PluralAttribute)}.
     *          Use of the typesafe version is strongly preferred.
     */
    @Nonnull
    <Y> Fetch<X, Y> fetch(@Nonnull String attributeName);

    /**
     * Create a fetch join to the specified attribute using
     * the given join type.
     * @param attributeName the name of the attribute that is the
     *        target of the join
     * @param jt the join type
     * @param <Y> the type of the fetched attribute
     * @return the resulting fetch join
     * @throws IllegalArgumentException if there is no attribute
     *         with the given name
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #fetch(SingularAttribute, JoinType)}
     *          or {@link #fetch(PluralAttribute, JoinType)}.
     *          Use of the typesafe version is strongly preferred.
     */
    @Nonnull
    <Y> Fetch<X, Y> fetch(@Nonnull String attributeName, @Nonnull JoinType jt);
}
```

C.14. AbstractQuery

```
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.EntityType;
import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.CollectionAttribute;
import jakarta.persistence.metamodel.ListAttribute;
import jakarta.persistence.metamodel.MapAttribute;
import jakarta.persistence.metamodel.SetAttribute;

import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.Set;

/**
 * Represents a bound type, usually an entity that appears in
 * the from clause, but may also be an embeddable belonging to
 * an entity in the from clause.
 * <p> Serves as a factory for {@link Join}s of associations,
 * embeddables, and collections belonging to the type, and for
 * {@link Path}s of attributes belonging to the type.
 *
 * @param <Z>  the source type
 * @param <X>  the target type
 *
 * @since 2.0
 */
```

```java
public interface From<Z, X> extends Path<X>, FetchParent<Z, X> {

    /**
     * Return the joins that have been made from this bound type.
     * Returns empty set if no joins have been made from this
     * bound type.
     * Modifications to the set do not affect the query.
     * @return joins made from this type
     */
    @Nonnull
    Set<Join<X, ?>> getJoins();

    /**
     * Whether the {@link From} object has been obtained as a result
     * of correlation (use of a {@link Subquery#correlate} method).
     * @return boolean indicating whether the object has been
     *         obtained through correlation
     */
    boolean isCorrelated();

    /**
     * Returns the parent {@link From} object from which the correlated
     * {@link From} object has been obtained through correlation (use
     * of {@link Subquery#correlate} method).
     * @return  the parent of the correlated {@code From} object
     * @throws IllegalStateException if the {@code From} object has
     *         not been obtained through correlation
     */
    @Nonnull
    From<Z, X> getCorrelationParent();

    /**
     * Create and add an inner join to the given entity.
     * @param entityClass  the target entity class
     * @param <Y> the target entity type
     * @return the resulting join
     * @since 3.2
     */
    @Nonnull
    <Y> Join<X, Y> join(@Nonnull Class<Y> entityClass);

    /**
     * Create and add a join to the given entity.
     * @param entityClass  the target entity class
     * @param joinType  join type
     * @param <Y> the target entity type
     * @return the resulting join
     * @since 3.2
     */
    @Nonnull
    <Y> Join<X, Y> join(@Nonnull Class<Y> entityClass,
                        @Nonnull JoinType joinType);

    /**
     * Create and add an inner join to the given entity.
     * @param entity  metamodel entity representing the join target
     * @param <Y> the target entity type
     * @return the resulting join
     * @since 3.2
     */
    @Nonnull
    <Y> Join<X, Y> join(@Nonnull EntityType<Y> entity);

    /**
     * Create and add a join to the given entity.
     * @param entity  metamodel entity representing the join target
     * @param joinType  join type
```

```
 * @param <Y> the target entity type
 * @return the resulting join
 * @since 3.2
 */
@Nonnull
<Y> Join<X, Y> join(@Nonnull EntityType<Y> entity,
                    @Nonnull JoinType joinType);

/**
 * Create an inner join to the specified single-valued
 * attribute.
 * @param attribute  target of the join
 * @param <Y> the type of the joined attribute
 * @return the resulting join
 */
@Nonnull
<Y> Join<X, Y> join(@Nonnull SingularAttribute<? super X, Y> attribute);

/**
 * Create a join to the specified single-valued attribute
 * using the given join type.
 * @param attribute  target of the join
 * @param jt  join type
 * @param <Y> the type of the joined attribute
 * @return the resulting join
 */
@Nonnull
<Y> Join<X, Y> join(@Nonnull SingularAttribute<? super X, Y> attribute,
                    @Nonnull JoinType jt);

/**
 * Create an inner join to the specified {@link Collection}-valued
 * attribute.
 * @param collection  target of the join
 * @param <Y> the element type of the joined collection
 * @return the resulting join
 */
@Nonnull
<Y> CollectionJoin<X, Y> join(@Nonnull CollectionAttribute<? super X, Y> collection);

/**
 * Create an inner join to the specified {@link Set}-valued
 * attribute.
 * @param set  target of the join
 * @param <Y> the element type of the joined set
 * @return the resulting join
 */
@Nonnull
<Y> SetJoin<X, Y> join(@Nonnull SetAttribute<? super X, Y> set);

/**
 * Create an inner join to the specified
 * {@link List}-valued attribute.
 * @param list  target of the join
 * @param <Y> the element type of the joined list
 * @return the resulting join
 */
@Nonnull
<Y> ListJoin<X, Y> join(@Nonnull ListAttribute<? super X, Y> list);

/**
 * Create an inner join to the specified {@link Map}-valued
 * attribute.
 * @param map  target of the join
 * @param <K> the key type of the joined map
 * @param <V> the value type of the joined map
 * @return the resulting join
```

```
 */
@Nonnull
<K, V> MapJoin<X, K, V> join(@Nonnull MapAttribute<? super X, K, V> map);

/**
 * Create a join to the specified {@link Collection}-valued
 * attribute using the given join type.
 * @param collection  target of the join
 * @param jt  join type
 * @param <Y> the element type of the joined collection
 * @return the resulting join
 */
@Nonnull
<Y> CollectionJoin<X, Y> join(@Nonnull CollectionAttribute<? super X, Y> collection,
                              @Nonnull JoinType jt);

/**
 * Create a join to the specified {@link Set}-valued attribute
 * using the given join type.
 * @param set  target of the join
 * @param jt  join type
 * @param <Y> the element type of the joined set
 * @return the resulting join
 */
@Nonnull
<Y> SetJoin<X, Y> join(@Nonnull SetAttribute<? super X, Y> set,
                       @Nonnull JoinType jt);

/**
 * Create a join to the specified {@link List}-valued attribute
 * using the given join type.
 * @param list  target of the join
 * @param jt  join type
 * @param <Y> the element type of the joined list
 * @return the resulting join
 */
@Nonnull
<Y> ListJoin<X, Y> join(@Nonnull ListAttribute<? super X, Y> list,
                        @Nonnull JoinType jt);

/**
 * Create a join to the specified {@link Map}-valued attribute
 * using the given join type.
 * @param map  target of the join
 * @param jt  join type
 * @param <K> the key type of the joined map
 * @param <V> the value type of the joined map
 * @return the resulting join
 */
@Nonnull
<K, V> MapJoin<X, K, V> join(@Nonnull MapAttribute<? super X, K, V> map,
                             @Nonnull JoinType jt);


//String-based:

/**
 * Create an inner join to the specified attribute.
 * @param attributeName the name of the attribute that is the
 *                      target of the join
 * @param <Y> the type of the joined attribute
 * @return the resulting join
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
```

```
 *          metamodel element to {@link #join(SingularAttribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<Y> Join<X, Y> join(@Nonnull String attributeName);

/**
 * Create an inner join to the specified {@link Collection}-valued
 * attribute.
 * @param attributeName the name of the attribute that is the
 *                      target of the join
 * @param <Y> the element type of the joined collection
 * @return the resulting join
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #join(CollectionAttribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<Y> CollectionJoin<X, Y> joinCollection(@Nonnull String attributeName);

/**
 * Create an inner join to the specified {@link Set}-valued
 * attribute.
 * @param attributeName the name of the attribute that is the
 *                      target of the join
 * @param <Y> the element type of the joined set
 * @return the resulting join
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #join(SetAttribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<Y> SetJoin<X, Y> joinSet(@Nonnull String attributeName);

/**
 * Create an inner join to the specified {@link List}-valued
 * attribute.
 * @param attributeName the name of the attribute that is the
 *                      target of the join
 * @param <Y> the element type of the joined list
 * @return the resulting join
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #join(ListAttribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<Y> ListJoin<X, Y> joinList(@Nonnull String attributeName);

/**
 * Create an inner join to the specified {@link Map}-valued
 * attribute.
 * @param attributeName the name of the attribute that is the
 *                      target of the join
 * @param <K> the key type of the joined map
 * @param <V> the value type of the joined map
```

```
 * @return the resulting join
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #join(MapAttribute)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<K, V> MapJoin<X, K, V> joinMap(@Nonnull String attributeName);

/**
 * Create a join to the specified attribute using the given
 * join type.
 * @param attributeName the name of the attribute that is the
 *                      target of the join
 * @param jt the join type
 * @param <Y> the type of the joined attribute
 * @return the resulting join
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #join(SingularAttribute, JoinType)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<Y> Join<X, Y> join(@Nonnull String attributeName,
                    @Nonnull JoinType jt);

/**
 * Create a join to the specified {@link Collection}-valued
 * attribute using the given join type.
 * @param attributeName the name of the attribute that is the
 *                      target of the join
 * @param jt the join type
 * @param <Y> the element type of the joined collection
 * @return the resulting join
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #join(CollectionAttribute, JoinType)}.
 *          Use of the typesafe version is strongly preferred.
 */
@Nonnull
<Y> CollectionJoin<X, Y> joinCollection(@Nonnull String attributeName,
                                        @Nonnull JoinType jt);

/**
 * Create a join to the specified {@link Set}-valued attribute
 * using the given join type.
 * @param attributeName the name of the attribute that is the
 *                      target of the join
 * @param jt the join type
 * @param <Y> the element type of the joined set
 * @return the resulting join
 * @throws IllegalArgumentException if there is no attribute
 *         with the given name
 *
 * @apiNote This method accepts a string-valued attribute name,
 *          and lacks type safety compared to passing a static
 *          metamodel element to {@link #join(SetAttribute, JoinType)}.
 *          Use of the typesafe version is strongly preferred.
```

```
     */
    @Nonnull
    <Y> SetJoin<X, Y> joinSet(@Nonnull String attributeName,
                              @Nonnull JoinType jt);

    /**
     * Create a join to the specified {@link List}-valued attribute
     * using the given join type.
     * @param attributeName the name of the attribute that is the
     *                      target of the join
     * @param jt the join type
     * @param <Y> the element type of the joined list
     * @return the resulting join
     * @throws IllegalArgumentException if there is no attribute
     *         with the given name
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #join(ListAttribute, JoinType)}.
     *          Use of the typesafe version is strongly preferred.
     */
    @Nonnull
    <Y> ListJoin<X, Y> joinList(@Nonnull String attributeName,
                                @Nonnull JoinType jt);

    /**
     * Create a join to the specified {@link Map}-valued attribute
     * using the given join type.
     * @param attributeName the name of the attribute that is the
     *                      target of the join
     * @param jt the join type
     * @return the resulting join
     * @throws IllegalArgumentException if there is no attribute
     *         with the given name
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #join(MapAttribute, JoinType)}.
     *          Use of the typesafe version is strongly preferred.
     *
     * @param <K> the key type of the joined map
     * @param <V> the value type of the joined map
     */
    @Nonnull
    <K, V> MapJoin<X, K, V> joinMap(@Nonnull String attributeName,
                                    @Nonnull JoinType jt);

    /**
     * Downcast the bound type to the given type.
     * @param type a subtype of the bound type
     * @param <T> the subtype of the bound type
     * @return this root or join downcast to the given type
     * @since 4.0
     */
    @Override
    @Nonnull
    <T extends X> From<?, T> treat(@Nonnull Class<T> type);
}
```

C.15. Join

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.metamodel.Attribute;
```

```java
import java.util.List;

/**
 * A join to an entity, embeddable, or basic type.
 *
 * @param <Z> the source type of the join
 * @param <X> the target type of the join
 *
 * @since 2.0
 */
public interface Join<Z, X> extends From<Z, X> {

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified join object
     * @since 2.1
     */
    @Nonnull
    Join<Z, X> on(@Nonnull Expression<Boolean> restriction);

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restrictions  zero or more restriction predicates
     * @return the modified join object
     * @since 2.1
     */
    @Nonnull
    Join<Z, X> on(@Nonnull BooleanExpression... restrictions);

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restrictions  zero or more restriction predicates
     * @return the modified join object
     * @since 4.0
     */
    @Nonnull
    Join<Z, X> on(@Nonnull List<? extends Expression<Boolean>> restrictions);

    /**
     * Return the predicate that corresponds to the ON
     * restriction(s) on the join, or null if no ON condition
     * has been specified.
     * @return the ON restriction predicate
     * @since 2.1
     */
    @Nullable
    Predicate getOn();

    /**
     * Return the metamodel attribute representing the join
     * target, if any, or null if the target of the join is an
     * entity type.
     * @return metamodel attribute or null
     */
    @Nullable
    Attribute<? super Z, ?> getAttribute();

    /**
     * Return the parent of the join.
     * @return join parent
```

```
     */
    @Nonnull
    From<?, Z> getParent();

    /**
     * Return the join type.
     * @return join type
     */
    @Nonnull
    JoinType getJoinType();

    /**
     * Downcast the joined type to the given type.
     * @param type a subtype of the joined type
     * @param <T> the subtype of the element type
     * @return this join downcast to the given type
     * @since 4.0
     */
    @Override
    @Nonnull
    <T extends X> Join<Z, T> treat(@Nonnull Class<T> type);
}
```

C.16. JoinType

```
/**
 * Defines the three varieties of join.
 *
 * <p>Support for {@link #RIGHT} outer joins is not required. Applications
 * which make use of right joins might not be portable between providers or
 * between SQL databases.
 *
 * @since 2.0
 */
public enum JoinType {

    /**
     * Inner join.
     */
    INNER,

    /**
     * Left outer join.
     */
    LEFT,

    /**
     * Right outer join.
     */
    RIGHT,
}
```

C.17. ListJoin

```
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import java.util.List;
import jakarta.persistence.metamodel.ListAttribute;

/**
 * The {@code ListJoin} interface is the type of the result of
 * joining to a collection over an association or element
```

```
 * collection that has been specified as a {@link java.util.List}.
 *
 * @param <Z> the source type of the join
 * @param <E> the element type of the target List
 *
 * @since 2.0
 */
public interface ListJoin<Z, E>
        extends PluralJoin<Z, List<E>, E> {

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified join object
     * @since 2.1
     */
    @Override
    @Nonnull
    ListJoin<Z, E> on(@Nonnull Expression<Boolean> restriction);

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restrictions  zero or more restriction predicates
     * @return the modified join object
     * @since 2.1
     */
    @Override
    @Nonnull
    ListJoin<Z, E> on(@Nonnull BooleanExpression... restrictions);

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restrictions  zero or more restriction predicates
     * @return the modified join object
     * @since 4.0
     */
    @Override
    @Nonnull
    ListJoin<Z, E> on(@Nonnull List<? extends Expression<Boolean>> restrictions);

    /**
     * Return the metamodel representation for the list attribute.
     * @return metamodel type representing the {@code List} that is
     *         the target of the join
     */
    @Override
    @Nonnull
    ListAttribute<? super Z, E> getModel();

    /**
     * Create an expression that corresponds to the index of
     * the object in the referenced association or element
     * collection.
     * This method must only be invoked upon an object that
     * represents an association or element collection for
     * which an order column has been defined.
     * @return expression denoting the index
     */
    @Nonnull
    Expression<Integer> index();
```

```java
    /**
     * Downcast the element type to the given type.
     * @param type a subtype of the element type
     * @param <T> the subtype of the element type
     * @return this join downcast to the given element type
     * @since 4.0
     */
    @Override
    @Nonnull
    <T extends E> ListJoin<Z, T> treat(@Nonnull Class<T> type);
}
```

C.18. LocalDateField

```java
import java.time.LocalDate;

/**
 * Each instance represents a type of field which can be
 * extracted from a {@link LocalDate}.
 *
 * @param <N> the resulting type of the extracted value
 *
 * @since 3.2
 */
public class LocalDateField<N> implements TemporalField<N, LocalDate> {

    private final String name;

    private LocalDateField(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }

    /**
     * The calendar year.
     */
    public static final LocalDateField<Integer> YEAR = new LocalDateField<>("year");
    /**
     * The calendar quarter, numbered from 1 to 4.
     */
    public static final LocalDateField<Integer> QUARTER = new LocalDateField<>("quarter");
    /**
     * The calendar month of the year, numbered from 1.
     */
    public static final LocalDateField<Integer> MONTH = new LocalDateField<>("month");
    /**
     * The ISO-8601 week number.
     */
    public static final LocalDateField<Integer> WEEK = new LocalDateField<>("week");
    /**
     * The calendar day of the month, numbered from 1.
     */
    public static final LocalDateField<Integer> DAY = new LocalDateField<>("day");
}
```

C.19. LocalDateTimeField

```java
import java.time.LocalDate;
import java.time.LocalDateTime;
```

```java
import java.time.LocalTime;

/**
 * Each instance represents a type of field which can be
 * extracted from a {@link LocalDateTime}.
 *
 * @param <N> the resulting type of the extracted value
 *
 * @since 3.2
 */
public class LocalDateTimeField<N> implements TemporalField<N, LocalDateTime> {

    private final String name;

    private LocalDateTimeField(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }

    /**
     * The calendar year.
     */
    public static final LocalDateTimeField<Integer> YEAR = new LocalDateTimeField<>("year");
    /**
     * The calendar quarter, numbered from 1 to 4.
     */
    public static final LocalDateTimeField<Integer> QUARTER = new LocalDateTimeField<>("quarter");
    /**
     * The calendar month of the year, numbered from 1.
     */
    public static final LocalDateTimeField<Integer> MONTH = new LocalDateTimeField<>("month");
    /**
     * The ISO-8601 week number.
     */
    public static final LocalDateTimeField<Integer> WEEK = new LocalDateTimeField<>("week");
    /**
     * The calendar day of the month, numbered from 1.
     */
    public static final LocalDateTimeField<Integer> DAY = new LocalDateTimeField<>("day");

    /**
     * The hour of the day in 24-hour time, numbered from 0 to 23.
     */
    public static final LocalDateTimeField<Integer> HOUR = new LocalDateTimeField<>("hour");
    /**
     * The minute of the hour, numbered from 0 to 59.
     */
    public static final LocalDateTimeField<Integer> MINUTE = new LocalDateTimeField<>("minute");
    /**
     * The second of the minute, numbered from 0 to 59, including a fractional
     * part representing fractions of a second
     */
    public static final LocalDateTimeField<Double> SECOND = new LocalDateTimeField<>("second");

    /**
     * The {@linkplain LocalDate date} part of a datetime.
     */
    public static final LocalDateTimeField<LocalDate> DATE = new LocalDateTimeField<>("date");
    /**
     * The {@linkplain LocalTime time} part of a datetime.
     */
    public static final LocalDateTimeField<LocalTime> TIME = new LocalDateTimeField<>("time");
```

```
}
```

C.20. LocalTimeField

```
import java.time.LocalTime;

/**
 * Each instance represents a type of field which can be
 * extracted from a {@link LocalTime}.
 *
 * @param <N> the resulting type of the extracted value
 *
 * @since 3.2
 */
public class LocalTimeField<N> implements TemporalField<N, LocalTime> {

    private final String name;

    private LocalTimeField(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }

    /**
     * The hour of the day in 24-hour time, numbered from 0 to 23.
     */
    public static final LocalTimeField<Integer> HOUR = new LocalTimeField<>("hour");
    /**
     * The minute of the hour, numbered from 0 to 59.
     */
    public static final LocalTimeField<Integer> MINUTE = new LocalTimeField<>("minute");
    /**
     * The second of the minute, numbered from 0 to 59, including a fractional
     * part representing fractions of a second
     */
    public static final LocalTimeField<Double> SECOND = new LocalTimeField<>("second");
}
```

C.21. MapJoin

```
import jakarta.annotation.Nonnull;

import java.util.List;
import java.util.Map;
import jakarta.persistence.metamodel.MapAttribute;

/**
 * The {@code MapJoin} interface is the type of the result of
 * joining to a collection over an association or element
 * collection that has been specified as a {@link java.util.Map}.
 *
 * @param <Z> the source type of the join
 * @param <K> the type of the target Map key
 * @param <V> the type of the target Map value
 *
 * @since 2.0
 */
public interface MapJoin<Z, K, V>
        extends PluralJoin<Z, Map<K, V>, V> {
```

```
/**
 * Modify the join to restrict the result according to the
 * specified ON condition and return the join object.
 * Replaces the previous ON condition, if any.
 * @param restriction  a simple or compound boolean expression
 * @return the modified join object
 * @since 2.1
 */
@Override
@Nonnull
MapJoin<Z, K, V> on(@Nonnull Expression<Boolean> restriction);

/**
 * Modify the join to restrict the result according to the
 * specified ON condition and return the join object.
 * Replaces the previous ON condition, if any.
 * @param restrictions  zero or more restriction predicates
 * @return the modified join object
 * @since 2.1
 */
@Override
@Nonnull
MapJoin<Z, K, V> on(@Nonnull BooleanExpression... restrictions);

/**
 * Modify the join to restrict the result according to the
 * specified ON condition and return the join object.
 * Replaces the previous ON condition, if any.
 * @param restrictions  zero or more restriction predicates
 * @return the modified join object
 * @since 4.0
 */
@Override
@Nonnull
MapJoin<Z, K, V> on(@Nonnull List<? extends Expression<Boolean>> restrictions);

/**
 * Return the metamodel representation for the map attribute.
 * @return metamodel type representing the {@code Map} that is
 *         the target of the join
 */
@Override
@Nonnull
MapAttribute<? super Z, K, V> getModel();

/**
 * Create a path expression that corresponds to the map key.
 * @return path corresponding to map key
 */
@Nonnull
Path<K> key();

/**
 * Create a path expression that corresponds to the map value.
 * This method is for stylistic use only: it just returns this.
 * @return path corresponding to the map value
 */
@Nonnull
Path<V> value();

/**
 * Create an expression that corresponds to the map entry.
 * @return expression corresponding to the map entry
 */
@Nonnull
Expression<Map.Entry<K, V>> entry();
```

```java
    /**
     * Downcast the value type to the given type.
     * @param type a subtype of the value type
     * @param <T> the subtype of the value type
     * @return this join downcast to the given value type
     * @since 4.0
     */
    @Override
    @Nonnull
    <T extends V> MapJoin<Z, K, T> treat(@Nonnull Class<T> type);
}
```

C.22. Nulls

```java
/**
 * Specifies the precedence of null values within query result sets.
 *
 * @see CriteriaBuilder#asc(Expression, Nulls)
 * @see CriteriaBuilder#desc(Expression, Nulls)
 *
 * @since 3.2
 */
public enum Nulls {
    /**
     * Null precedence not specified.
     */
    NONE,
    /**
     * Null values occur at the beginning of the result set.
     */
    FIRST,
    /**
     * Null values occur at the end of the result set.
     */
    LAST
}
```

C.23. Order

```java
import jakarta.annotation.Nonnull;

/**
 * An object that defines an ordering over the query results.
 *
 * @since 2.0
 */
public interface Order {

    /**
     * Switch the ordering.
     * @return a new {@code Order} instance with the reversed ordering
     */
    @Nonnull
    Order reverse();

    /**
     * Whether ascending ordering is in effect.
     * @return boolean indicating whether ordering is ascending
     */
    boolean isAscending();

    /**
```

```
     * Return the precedence of null values.
     * @return the {@linkplain Nulls precedence of null values}
     * @since 3.2
     */
    @Nonnull
    Nulls getNullPrecedence();

   /**
    * Return the expression that is used for ordering.
    * @return expression used for ordering
    */
    @Nonnull
    Expression<?> getExpression();
}
```

C.24. ParameterExpression

```
import jakarta.persistence.Parameter;

/**
 * Type of criteria query parameter expressions.
 *
 * @param <T> the type of the parameter expression
 *
 * @since 2.0
 */
public interface ParameterExpression<T> extends Parameter<T>, Expression<T> {}
```

C.25. Path

```
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import java.time.temporal.Temporal;
import java.util.Collection;
import java.util.Map;

import jakarta.persistence.metamodel.BooleanAttribute;
import jakarta.persistence.metamodel.ComparableAttribute;
import jakarta.persistence.metamodel.NumericAttribute;
import jakarta.persistence.metamodel.PluralAttribute;
import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.Bindable;
import jakarta.persistence.metamodel.MapAttribute;
import jakarta.persistence.metamodel.TextAttribute;
import jakarta.persistence.metamodel.TemporalAttribute;

/**
 * Represents a simple or compound attribute path from a
 * bound type or collection, and is a "primitive" expression.
 *
 * @param <X>  the type referenced by the path
 *
 * @since 2.0
 */
public interface Path<X> extends Expression<X> {

    /**
     * Return the bindable object that corresponds to the path
     * expression.
     * @return bindable object corresponding to the path
     */
```

```
    @Nonnull
    Bindable<X> getModel();

    /**
     * Return the parent "node" in the path or null if no parent.
     * @return parent
     */
    @Nullable
    Path<?> getParentPath();

    /**
     * Create a path corresponding to the referenced
     * single-valued attribute.
     * @param attribute single-valued attribute
     * @param <Y> the type of the attribute
     * @return path corresponding to the referenced attribute
     */
    @Nonnull
    <Y> Path<Y> get(@Nonnull SingularAttribute<? super X, Y> attribute);

    /**
     *  Create a path corresponding to the referenced
     *  single-valued attribute.
     *  @param attribute single-valued attribute
     *  @return path corresponding to the referenced attribute
     *  @since 4.0
     */
    @Nonnull
    BooleanExpression get(@Nonnull BooleanAttribute<? super X> attribute);

    /**
     *  Create a path corresponding to the referenced
     *  single-valued attribute.
     *  @param attribute single-valued attribute
     *  @param <C> the comparable type of the attribute
     *  @return path corresponding to the referenced attribute
     *  @since 4.0
     */
    @Nonnull
    <C extends Comparable<? super C>> ComparableExpression<C> get(@Nonnull ComparableAttribute<? super X, C>
attribute);

    /**
     *  Create a path corresponding to the referenced
     *  single-valued attribute.
     *  @param attribute single-valued attribute
     *  @param <T> the temporal type of the attribute
     *  @return path corresponding to the referenced attribute
     *  @since 4.0
     */
    @Nonnull
    <T extends Temporal & Comparable<? super T>> TemporalExpression<T> get(@Nonnull TemporalAttribute<? super X, T>
attribute);

    /**
     *  Create a path corresponding to the referenced
     *  single-valued attribute.
     *  @param attribute single-valued attribute
     *  @param <N> the numeric type of the attribute
     *  @return path corresponding to the referenced attribute
     *  @since 4.0
     */
    @Nonnull
    <N extends Number & Comparable<N>> NumericExpression<N> get(@Nonnull NumericAttribute<? super X, N> attribute);

    /**
     *  Create a path corresponding to the referenced
```

```
 *  single-valued attribute.
 *  @param attribute single-valued attribute
 *  @return path corresponding to the referenced attribute
 *  @since 4.0
 */
@Nonnull
TextExpression get(@Nonnull TextAttribute<? super X> attribute);

/**
 * Create a path corresponding to the referenced
 * collection-valued attribute.
 * @param collection collection-valued attribute
 * @param <E> the element type of the collection
 * @param <C> the collection type
 * @return expression corresponding to the referenced attribute
 */
@Nonnull
<E, C extends Collection<E>> PluralExpression<C,E> get(@Nonnull PluralAttribute<? super X, C, E> collection);

/**
 * Create a path corresponding to the referenced
 * map-valued attribute.
 * @param map map-valued attribute
 * @param <K> the key type of the map
 * @param <V> the value type of the map
 * @param <M> the map type
 * @return expression corresponding to the referenced attribute
 */
@Nonnull
<K, V, M extends Map<K, V>> PluralExpression<M,V> get(@Nonnull MapAttribute<? super X, K, V> map);

/**
 * Create an expression corresponding to the type of the path.
 * @return expression corresponding to the type of the path
 */
@Nonnull
Expression<Class<? extends X>> type();


//String-based:

/**
 * Create a path corresponding to the referenced attribute.
 *
 * <p> Note: Applications using the string-based API may need to
 * specify the type resulting from the {@link #get} operation in
 * order to avoid the use of {@code Path} variables.
 *
 * <p>For example:
 * {@snippet :
 * CriteriaQuery<Person> q = cb.createQuery(Person.class);
 * Root<Person> p = q.from(Person.class);
 * q.select(p)
 *  .where(cb.isMember("joe",
 *                     p.<Set<String>>get("nicknames")));
 * }
 * <p>rather than:
 * {@snippet :
 * CriteriaQuery<Person> q = cb.createQuery(Person.class);
 * Root<Person> p = q.from(Person.class);
 * Path<Set<String>> nicknames = p.get("nicknames");
 * q.select(p)
 *  .where(cb.isMember("joe", nicknames));
 * }
 *
 * @param attributeName the name of an attribute of the managed
 *                      type of the path
```

```
     * @return path corresponding to the referenced attribute
     * @throws IllegalStateException if invoked on a path that
     *         corresponds to a basic type
     * @throws IllegalArgumentException if there is no attribute
     *         with the given name
     *
     * @apiNote This method accepts a string-valued attribute name,
     *          and lacks type safety compared to passing a static
     *          metamodel element to {@link #get(SingularAttribute)}.
     *          Use of the typesafe version is strongly preferred.
     *
     * @param <Y> the type of the attribute
     */
    @Nonnull
    <Y> Path<Y> get(@Nonnull String attributeName);

    /**
     * Downcast the type of the path to the given type.
     * @param type a subtype of the path type
     * @param <T> the subtype of the path type
     * @return this path downcast to the given type
     * @since 4.0
     */
    @Nonnull
    <T extends X> Path<T> treat(@Nonnull Class<T> type);
}
```

C.26. PluralJoin

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.PluralAttribute;

/**
 * The {@code PluralJoin} interface defines functionality
 * that is common to joins to all collection types.  It is
 * not intended to be used directly in query construction.
 *
 * @param <Z> the source type
 * @param <C> the collection type
 * @param <E> the element type of the collection
 *
 * @since 2.0
 */
public interface PluralJoin<Z, C, E> extends Join<Z, E> {

    /**
     * Return the metamodel representation for the collection-valued
     * attribute corresponding to the join.
     * @return metamodel collection-valued attribute corresponding
     *         to the target of the join
     */
    @Override
    @Nonnull
    PluralAttribute<? super Z, C, E> getModel();

    /**
     * Downcast the element type to the given type.
     * @param type a subtype of the element type
     * @param <T> the subtype of the element type
     * @return this plural join downcast to the given element type
     * @since 4.0
     */
    @Override
    @Nonnull
    <T extends E> PluralJoin<Z, ?, T> treat(@Nonnull Class<T> type);
```

```
}
```

C.27. Predicate

```
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import java.util.List;

/**
 * The type of a simple or compound predicate: a conjunction or
 * disjunction of restrictions.
 * A simple predicate is considered to be a conjunction with a
 * single conjunct.
 *
 * @since 2.0
 */
public interface Predicate extends BooleanExpression {

    enum BooleanOperator { AND, OR }

    /**
     * Return the boolean operator for the predicate.
     * If the predicate is simple, this is {@code AND}.
     * @return boolean operator for the predicate
     */
    @Nonnull
    BooleanOperator getOperator();

    /**
     * Whether the predicate has been created from another
     * predicate by applying {@link Predicate#not()}
     * or by calling {@link CriteriaBuilder#not}.
     * @return boolean indicating if the predicate is
     *                 a negated predicate
     */
    boolean isNegated();

    /**
     * Return the top-level conjuncts or disjuncts of the
     * predicate. Returns empty list if there are no top-level
     * conjuncts or disjuncts of the predicate.
     * Modifications to the list do not affect the query.
     * @return list of boolean expressions forming the predicate
     */
    @Nonnull
    List<Expression<Boolean>> getExpressions();

    /**
     * Create a negation of the predicate.
     * @return negated predicate
     */
    @Override
    @Nonnull
    Predicate not();

}
```

C.28. Root

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.EntityType;
```

```java
/**
 * A root type in the from clause.
 * Query roots always reference entities.
 *
 * @param <X> the entity type referenced by the root
 *
 * @since 2.0
 */
public interface Root<X> extends From<X, X> {

    /**
     * Return the metamodel entity corresponding to the root.
     * @return metamodel entity corresponding to the root
     */
    @Override
    @Nonnull
    EntityType<X> getModel();

    /**
     * Downcast the root type to the given type.
     * @param type a subtype of the root type
     * @param <T> the subtype of the root type
     * @return this root downcast to the given type
     * @since 4.0
     */
    @Override
    @Nonnull
    <T extends X> Root<T> treat(@Nonnull Class<T> type);
}
```

C.29. Selection

```java
import jakarta.annotation.Nonnull;
import jakarta.persistence.TupleElement;
import java.util.List;

/**
 * The {@code Selection} interface defines an item that is to be
 * returned in a query result.
 *
 * @param <X> the type of the selection item
 *
 * @since 2.0
 */
public interface Selection<X> extends TupleElement<X> {

    /**
     * Assigns an alias to the selection item.
     * Once assigned, an alias cannot be changed or reassigned.
     * Returns the same selection item.
     * @param name  alias
     * @return selection item
     */
    @Nonnull
    Selection<X> alias(@Nonnull String name);

    /**
     * Whether the selection item is a compound selection.
     * @return boolean indicating whether the selection is a compound
     *         selection
     */
    boolean isCompoundSelection();

    /**
     * Return the selection items composing a compound selection.
```

```
     * Modifications to the list do not affect the query.
     * @return list of selection items
     * @throws IllegalStateException if selection is not a
     *         compound selection
     */
    @Nonnull
    List<Selection<?>> getCompoundSelectionItems();
}
```

C.30. SetJoin

```
import jakarta.annotation.Nonnull;

import java.util.List;
import java.util.Set;
import jakarta.persistence.metamodel.SetAttribute;

/**
 * The {@code SetJoin} interface is the type of the result of
 * joining to a collection over an association or element
 * collection that has been specified as a {@link java.util.Set}.
 *
 * @param <Z> the source type of the join
 * @param <E> the element type of the target {@code Set}
 *
 * @since 2.0
 */
public interface SetJoin<Z, E> extends PluralJoin<Z, Set<E>, E> {

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restriction  a simple or compound boolean expression
     * @return the modified join object
     * @since 2.1
     */
    @Override
    @Nonnull
    SetJoin<Z, E> on(@Nonnull Expression<Boolean> restriction);

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restrictions  zero or more restriction predicates
     * @return the modified join object
     * @since 2.1
     */
    @Override
    @Nonnull
    SetJoin<Z, E> on(@Nonnull BooleanExpression... restrictions);

    /**
     * Modify the join to restrict the result according to the
     * specified ON condition and return the join object.
     * Replaces the previous ON condition, if any.
     * @param restrictions  zero or more restriction predicates
     * @return the modified join object
     * @since 4.0
     */
    @Override
    @Nonnull
    SetJoin<Z, E> on(@Nonnull List<? extends Expression<Boolean>> restrictions);
```

```java
    /**
     * Return the metamodel representation for the set attribute.
     * @return metamodel type representing the {@code Set} that is
     *         the target of the join
     */
    @Override
    @Nonnull
    SetAttribute<? super Z, E> getModel();

    /**
     * Downcast the element type to the given type.
     * @param type a subtype of the element type
     * @param <T> the subtype of the element type
     * @return this join downcast to the given element type
     * @since 4.0
     */
    @Override
    @Nonnull
    <T extends E> SetJoin<Z, T> treat(@Nonnull Class<T> type);
}
```

C.31. Subquery

```java
package jakarta.persistence.criteria;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import java.util.List;
import java.util.Set;

/**
 * The {@code Subquery} interface defines functionality that is
 * specific to subqueries.
 *
 * <p>A subquery has an expression as its selection item.
 *
 * @param <T> the type of the selection item.
 *
 * @since 2.0
 */
public interface Subquery<T> extends AbstractQuery<T>, Expression<T> {

    /**
     * Specify the item that is to be returned as the subquery
     * result.
     * Replaces the previously specified selection, if any.
     * @param expression  expression specifying the item that
     *        is to be returned as the subquery result
     * @return the modified subquery
     */
    @Nonnull
    Subquery<T> select(@Nonnull Expression<T> expression);

    /**
     * Modify the subquery to restrict the result according
     * to the specified boolean expression.
     * Replaces the previously added restriction(s), if any.
     * This method only overrides the return type of the
     * corresponding {@code AbstractQuery} method.
     * @param restriction  a simple or compound boolean expression
     * @return the modified subquery
     */
    @Nonnull
    Subquery<T> where(@Nonnull Expression<Boolean> restriction);
```

```java
/**
 * Modify the subquery to restrict the result according
 * to the conjunction of the specified restriction predicates.
 * Replaces the previously added restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restrictions  zero or more restriction predicates
 * @return the modified subquery
 */
@Nonnull
Subquery<T> where(@Nonnull BooleanExpression... restrictions);

/**
 * Modify the query to restrict the query result according
 * to the conjunction of the specified restriction predicates.
 * Replaces the previously added restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restrictions  a list of zero or more restriction predicates
 * @return the modified query
 * @since 3.2
 */
@Nonnull
Subquery<T> where(@Nonnull List<? extends Expression<Boolean>> restrictions);

/**
 * Specify the expressions that are used to form groups over
 * the subquery results.
 * Replaces the previous specified grouping expressions, if any.
 * If no grouping expressions are specified, any previously
 * added grouping expressions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param grouping  zero or more grouping expressions
 * @return the modified subquery
 */
@Nonnull
Subquery<T> groupBy(@Nonnull Expression<?>... grouping);

/**
 * Specify the expressions that are used to form groups over
 * the subquery results.
 * Replaces the previous specified grouping expressions, if any.
 * If no grouping expressions are specified, any previously
 * added grouping expressions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param grouping  list of zero or more grouping expressions
 * @return the modified subquery
 */
@Nonnull
Subquery<T> groupBy(@Nonnull List<Expression<?>> grouping);

/**
 * Specify a restriction over the groups of the subquery.
 * Replaces the previous having restriction(s), if any.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restriction  a simple or compound boolean expression
 * @return the modified subquery
 */
@Nonnull
Subquery<T> having(@Nonnull Expression<Boolean> restriction);
```

```java
/**
 * Specify restrictions over the groups of the subquery
 * according the conjunction of the specified restriction
 * predicates.
 * Replaces the previously added having restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restrictions  zero or more restriction predicates
 * @return the modified subquery
 */
@Nonnull
Subquery<T> having(@Nonnull BooleanExpression... restrictions);

/**
 * Specify restrictions over the groups of the query
 * according the conjunction of the specified restriction
 * predicates.
 * Replaces the previously added having restriction(s), if any.
 * If no restrictions are specified, any previously added
 * restrictions are simply removed.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param restrictions  a list of zero or more restriction predicates
 * @return the modified query
 * @since 3.2
 */
@Nonnull
Subquery<T> having(@Nonnull List<? extends Expression<Boolean>> restrictions);

/**
 * Specify whether duplicate query results are eliminated.
 * A true value will cause duplicates to be eliminated.
 * A false value will cause duplicates to be retained.
 * If distinct has not been specified, duplicate results must
 * be retained.
 * This method only overrides the return type of the
 * corresponding {@code AbstractQuery} method.
 * @param distinct  boolean value specifying whether duplicate
 *        results must be eliminated from the subquery result or
 *        whether they must be retained
 * @return the modified subquery.
 */
@Nonnull
Subquery<T> distinct(boolean distinct);

/**
 * Create a subquery root or join correlated to a root or join
 * of the enclosing query.
 * @param parent  root or join of the containing query
 * @param <X> the source type
 * @param <Y> the target type
 * @return subquery from
 * @since 4.0
 */
@Nonnull
<X, Y> From<X, Y> correlate(@Nonnull From<X, Y> parent);

/**
 * Create a subquery root correlated to a root of the
 * enclosing query.
 * @param parentRoot  a root of the containing query
 * @param <Y> the entity type
 * @return subquery root
 */
```

```
@Nonnull
<Y> Root<Y> correlate(@Nonnull Root<Y> parentRoot);

/**
 * Create a subquery join object correlated to a join object
 * of the enclosing query.
 * @param parentJoin  join object of the containing query
 * @param <X> the source type
 * @param <Y> the target type
 * @return subquery join
 */
@Nonnull
<X, Y> Join<X, Y> correlate(@Nonnull Join<X, Y> parentJoin);

/**
 * Create a subquery collection join object correlated to a
 * collection join object of the enclosing query.
 * @param parentCollection  join object of the containing query
 * @param <X> the source type
 * @param <Y> the element type
 * @return subquery join
 */
@Nonnull
<X, Y> CollectionJoin<X, Y> correlate(@Nonnull CollectionJoin<X, Y> parentCollection);

/**
 * Create a subquery set join object correlated to a set join
 * object of the enclosing query.
 * @param parentSet  join object of the containing query
 * @param <X> the source type
 * @param <Y> the element type
 * @return subquery join
 */
@Nonnull
<X, Y> SetJoin<X, Y> correlate(@Nonnull SetJoin<X, Y> parentSet);

/**
 * Create a subquery list join object correlated to a list join
 * object of the enclosing query.
 * @param parentList join object of the containing query
 * @param <X> the source type
 * @param <Y> the element type
 * @return subquery join
 */
@Nonnull
<X, Y> ListJoin<X, Y> correlate(@Nonnull ListJoin<X, Y> parentList);

/**
 * Create a subquery map join object correlated to a map join
 * object of the enclosing query.
 * @param parentMap join object of the containing query
 * @param <X> the source type
 * @param <K> the key type
 * @param <V> the value type
 * @return subquery join
 */
@Nonnull
<X, K, V> MapJoin<X, K, V> correlate(@Nonnull MapJoin<X, K, V> parentMap);

/**
 * Return the query of which this is a subquery.
 * This must be a CriteriaQuery or a Subquery.
 * @return the enclosing query or subquery
 */
@Nonnull
AbstractQuery<?> getParent();
```

```
/**
 * Return the query of which this is a subquery.
 * This may be a CriteriaQuery, CriteriaUpdate, CriteriaDelete,
 * or a Subquery.
 * @return the enclosing query or subquery
 * @since 2.1
 */
@Nonnull
CommonAbstractCriteria getContainingQuery();

/**
 * Return the selection expression.
 * @return the item to be returned in the subquery result
 */
@Nullable
Expression<T> getSelection();

/**
 * Return the correlated joins of the subquery.
 * Returns empty set if the subquery has no correlated
 * joins.
 * Modifications to the set do not affect the query.
 * @return the correlated joins of the subquery
 */
@Nonnull
Set<Join<?, ?>> getCorrelatedJoins();

//subqueries:

/**
 * Create a predicate testing the existence of this subquery result.
 * @return exists predicate
 * @see CriteriaBuilder#exists(Subquery)
 * @since 4.0
 */
@Nonnull
Predicate exists();

/**
 * Create an all expression over the subquery results.
 * @return all expression
 * @see CriteriaBuilder#all(Subquery)
 * @since 4.0
 */
@Nonnull
Expression<T> all();

/**
 * Create a some expression over the subquery results.
 * This expression is equivalent to an <code>any</code> expression.
 * @return some expression
 * @see CriteriaBuilder#some(Subquery)
 * @since 4.0
 */
@Nonnull
Expression<T> some();

/**
 * Create an any expression over the subquery results.
 * This expression is equivalent to a <code>some</code> expression.
 * @return any expression
 * @see CriteriaBuilder#any(Subquery)
 * @since 4.0
 */
@Nonnull
Expression<T> any();
```

```
}
```

C.32. TemporalField

```
import java.time.temporal.Temporal;

/**
 * Each instance represents a type of field which can be
 * extracted from a date, time, or datetime.
 *
 * @param <N> the resulting type of the extracted value
 * @param <T> the temporal type (date, time, or datetime)
 *
 * @see LocalDateField
 * @see LocalTimeField
 * @see LocalDateTimeField
 * @see CriteriaBuilder#extract(TemporalField, Expression)
 *
 * @since 3.2
 */
public interface TemporalField<N,T extends Temporal> {}
```

Appendix D: Metamodel API Interfaces

The following APIs are defined in the package `jakarta.persistence.metamodel`.

D.1. `Metamodel`

```
package jakarta.persistence.metamodel;

import jakarta.annotation.Nonnull;

import java.util.Set;

/**
 * Provides access to the metamodel of persistent entities in the
 * persistence unit.
 *
 * @since 2.0
 */
public interface Metamodel {

    /**
     * Return the metamodel entity type representing the entity.
     * @param entityName  the name of the represented entity
     * @return the metamodel entity type
     * @throws IllegalArgumentException if not an entity
     * @see jakarta.persistence.Entity#name
     * @since 3.2
     */
    @Nonnull
    EntityType<?> entity(@Nonnull String entityName);

    /**
     * Return the metamodel entity type representing the entity.
     * @param cls  the type of the represented entity
     * @param <X> The type of the represented entity
     * @return the metamodel entity type
     * @throws IllegalArgumentException if not an entity
     */
    @Nonnull
    <X> EntityType<X> entity(@Nonnull Class<X> cls);

    /**
     * Return the metamodel managed type representing the
     * entity, mapped superclass, or embeddable class.
     * @param cls  the type of the represented managed class
     * @param <X> The type of the represented managed class
     * @return the metamodel managed type
     * @throws IllegalArgumentException if not a managed class
     */
    @Nonnull
    <X> ManagedType<X> managedType(@Nonnull Class<X> cls);

    /**
     * Return the metamodel embeddable type representing the
     * embeddable class.
     * @param cls  the type of the represented embeddable class
     * @param <X> The type of the represented embeddable class
     * @return the metamodel embeddable type
     * @throws IllegalArgumentException if not an embeddable class
     */
    @Nonnull
    <X> EmbeddableType<X> embeddable(@Nonnull Class<X> cls);

    /**
```

```
     * Return the metamodel managed types.
     * @return the metamodel managed types
     */
    @Nonnull
    Set<ManagedType<?>> getManagedTypes();

    /**
     * Return the metamodel entity types.
     * @return the metamodel entity types
     */
    @Nonnull
    Set<EntityType<?>> getEntities();

    /**
     * Return the metamodel embeddable types.
     * Returns am empty set if there are no embeddable types.
     * @return the metamodel embeddable types
     */
    @Nonnull
    Set<EmbeddableType<?>> getEmbeddables();
}
```

D.2. StaticMetamodel

```
package jakarta.persistence.metamodel;

import jakarta.annotation.Nonnull;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

/**
 * The {@code StaticMetamodel} annotation specifies that the class is
 * a metamodel class that represents the Java type designated by the
 * {@link #value} element. The Java type might be:
 * <ul>
 * <li>an entity class, mapped superclass, or embeddable class,
 * <li>a class with one or more {@link jakarta.persistence.NamedQuery}
 *    or {@link jakarta.persistence.NamedNativeQuery} annotations, or
 * <li>a class with one or more methods annotated
 *     {@link jakarta.persistence.query.JakartaQuery} or
 *     {@link jakarta.persistence.query.NativeQuery}.
 * </ul>
 *
 * <p>Typically, a static metamodel class is generated by an annotation
 * processor according to the specification of the Jakarta Persistence
 * <em>canonical metamodel</em>. Metamodel classes which do not
 * conform to this specification are not portable between persistence
 * providers.
 *
 * <p>The canonical metamodel for a class {@code X} belongs to the same
 * package as {@code X}, and is named {@code X_}.
 *
 * <p>Suppose the persistence unit contains the following entity class:
 * {@snippet :
 * package org.example;
 * import jakarta.persistence.*;
 *
 * @Entity class Book {
 *     @Id String isbn;
 *     ...
 * }
```

```
 * }
 * <p>Then the canonical metamodel class is {@code org.example.Book_}.
 * {@snippet :
 * import org.example.*;
 * import jakarta.persistence.*;
 * ...
 *     EntityType<Book> bookType = Book_.class_;
 *     SingularAttribute<Book,String> isbn = Book_.isbn;
 * }
 *
 * <p>Similarly, suppose the persistence unit contains the following
 * interface annotated {@code @NamedQuery}:
 * {@snippet :
 * package com.acme.orders;
 * import jakarta.persistence.*;
 *
 * @NamedQuery(name = "allOrdersBySku",
 *             query = "select o from Order o join i.items i where i.sku = ?1")
 * ...
 * interface OrderQueries { ... }
 * }
 * <p>Then the canonical metamodel class is {@code com.acme.orders.OrderQueries_}.
 * {@snippet :
 * import com.acme.orders.*;
 * import jakarta.persistence.*;
 * ...
 *     var ordersForSku =
 *             entityManager.createQuery(OrderQueries_._allOrdersBySku_)
 *                     .setParameter(1, sku)
 *                     .getResultList();
 * }
 * <p>Note that the canonical metamodel field for the named query
 * {@code allOrdersBySku} is named {@code _allOrdersBySku_} to avoid
 * naming collisions.
 *
 * @since 2.0
 */
@Target(TYPE)
@Retention(RUNTIME)
public @interface StaticMetamodel {

    /**
     * Class being modelled by the annotated class.
     */
    @Nonnull
    Class<?> value();
}
```

D.3. Attribute

```
import jakarta.annotation.Nonnull;

/**
 * Represents an attribute of a Java type.
 *
 * @param <X> The represented type that contains the attribute
 * @param <Y> The type of the represented attribute
 *
 * @since 2.0
 */
public interface Attribute<X, Y> {

    enum PersistentAttributeType {

        /** Many-to-one association */
```

```
    MANY_TO_ONE,

    /** One-to-one association */
    ONE_TO_ONE,

    /** Basic attribute */
    BASIC,

    /** Embeddable class attribute */
    EMBEDDED,

    /** Many-to-many association */
    MANY_TO_MANY,

    /** One-to-many association */
    ONE_TO_MANY,

    /** Element collection */
    ELEMENT_COLLECTION
}

/**
 * Return the name of the attribute.
 * @return name
 */
@Nonnull
String getName();

/**
 * Return the persistent attribute type for the attribute.
 * @return persistent attribute type
 */
@Nonnull
PersistentAttributeType getPersistentAttributeType();

/**
 * Return the managed type representing the type in which
 * the attribute was declared.
 * @return declaring type
 */
@Nonnull
ManagedType<X> getDeclaringType();

/**
 * Return the Java type of the represented attribute.
 * @return Java type
 */
@Nonnull
Class<Y> getJavaType();

/**
 * Return the {@link java.lang.reflect.Member} for the
 *  represented attribute.
 * @return corresponding {@link java.lang.reflect.Member}
 */
@Nonnull
java.lang.reflect.Member getJavaMember();

/**
 * Is the attribute an association.
 * @return boolean indicating whether the attribute
 *         corresponds to an association
 */
boolean isAssociation();

/**
 * Is the attribute collection-valued (represents a
```

```
     * {@code Collection}, {@code Set}, {@code List}, or
     * {@code Map}).
     * @return boolean indicating whether the attribute is
     *         collection-valued
     */
    boolean isCollection();
}
```

D.4. BasicType

```
/**
 * An instance of {@code BasicType} represents a
 * {@linkplain jakarta.persistence.Basic basic type}
 * (possibly an {@linkplain jakarta.persistence.Enumerated
 * enumerated}, {@linkplain jakarta.persistence.Lob LOB}, or
 * {@linkplain jakarta.persistence.Temporal temporal} type).
 *
 * @param <X> The represented basic type
 *
 * @since 2.0
 */
public interface BasicType<X> extends Type<X> {}
```

D.5. Bindable

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.criteria.Path;

/**
 * An instances of the type {@code Bindable} represents an object
 * or attribute type that can be bound into a {@link Path Path}.
 *
 * @param <T> The type of the represented object or attribute
 *
 * @since 2.0
 *
 */
public interface Bindable<T> {

    enum BindableType {
        /**
         * Single-valued attribute type.
         *
         * @see SingularAttribute
         */
        SINGULAR_ATTRIBUTE,

        /**
         * Multivalued attribute type, that is, a collection.
         *
         * @see PluralAttribute
         */
        PLURAL_ATTRIBUTE,

        /**
         * Entity type.
         *
         * @see EntityType
         */
        ENTITY_TYPE
    }

    /**
```

```
     * Return the bindable type of the represented object.
     * @return bindable type
     */
    @Nonnull
    BindableType getBindableType();

    /**
     * Return the Java type of the represented object.
     * If the bindable type of the object is {@code PLURAL_ATTRIBUTE},
     * the Java element type is returned. If the bindable type is
     * {@code SINGULAR_ATTRIBUTE} or {@code ENTITY_TYPE},
     * the Java type of the
     * represented entity or attribute is returned.
     * @return Java type
     */
    @Nonnull
    Class<T> getBindableJavaType();
}
```

D.6. CollectionAttribute

```
/**
 * Instances of the type {@code CollectionAttribute} represent
 * persistent {@link java.util.Collection}-valued attributes.
 *
 * @param <X> The type the represented Collection belongs to
 * @param <E> The element type of the represented Collection
 *
 * @since 2.0
 *
 */
public interface CollectionAttribute<X, E>
    extends PluralAttribute<X, java.util.Collection<E>, E> {}
```

D.7. EmbeddableType

```
/**
 * An instance of {@code EmbeddableType} represents an
 * {@linkplain jakarta.persistence.Embeddable embeddable}
 * type.
 *
 * @param <X> The represented embeddable type.
 *
 * @since 2.0
 */
public interface EmbeddableType<X> extends ManagedType<X> {}
```

D.8. EntityType

```
package jakarta.persistence.metamodel;

import jakarta.annotation.Nonnull;
import jakarta.persistence.EntityGraph;

import java.util.Map;

/**
 * An instance of {@code EntityType} represents
 * an {@linkplain jakarta.persistence.Entity entity}
 * type.
 *
```

```
 * @param <X> The represented entity type.
 *
 * @since 2.0
 */
public interface EntityType<X>
            extends IdentifiableType<X>, Bindable<X>{

    /**
     * Return the entity name.
     * @return entity name
     */
    @Nonnull
    String getName();

    /**
     * Create a new mutable {@link EntityGraph} with this given root entity type,
     * allowing programmatic definition of the graph.
     * @return a trivial entity graph with only a root node
     * @see jakarta.persistence.EntityHandler#createEntityGraph(Class)
     * @since 4.0
     */
    @Nonnull
    EntityGraph<X> createEntityGraph();

    /**
     * A map keyed by {@linkplain jakarta.persistence.NamedEntityGraph#name
     * graph name}, containing every named {@linkplain EntityGraph entity graph}
     * whose root entity type is exactly this type.
     * @return a map keyed by graph name
     * @see jakarta.persistence.EntityManagerFactory#getNamedEntityGraphs(Class)
     * @since 4.0
     */
    @Nonnull
    Map<String, EntityGraph<X>> getNamedEntityGraphs();
}
```

D.9. IdentifiableType

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

import java.util.Set;

/**
 * An instance of the type {@code IdentifiableType} represents an
 * entity or mapped superclass type.
 *
 * @param <X> The represented entity or mapped superclass type.
 *
 * @since 2.0
 *
 */
public interface IdentifiableType<X> extends ManagedType<X> {

    /**
     * Return the attribute that corresponds to the id attribute of
     * the entity or mapped superclass.
     * @param type  the type of the represented id attribute
     * @param <Y> The type of the represented id attribute
     * @return id attribute
     * @throws IllegalArgumentException if id attribute of the given
     *         type is not present in the identifiable type or if
     *         the identifiable type has an id class
     */
    @Nonnull
```

```
<Y> SingularAttribute<? super X, Y> getId(Class<Y> type);

/**
 * Return the attribute that corresponds to the id attribute
 * declared by the entity or mapped superclass.
 * @param type  the type of the represented declared
 *              id attribute
 * @param <Y> The type of the represented declared id attribute
 * @return declared id attribute
 * @throws IllegalArgumentException if id attribute of the given
 *         type is not declared in the identifiable type or if
 *         the identifiable type has an id class
 */
@Nonnull
<Y> SingularAttribute<X, Y> getDeclaredId(Class<Y> type);

/**
 * Return the attribute that corresponds to the version
 * attribute of the entity or mapped superclass.
 * @param type  the type of the represented version attribute
 * @param <Y> The type of the represented version attribute
 * @return version attribute
 * @throws IllegalArgumentException if version attribute of the
 *          given type is not present in the identifiable type
 */
@Nonnull
<Y> SingularAttribute<? super X, Y> getVersion(Class<Y> type);

/**
 * Return the attribute that corresponds to the version
 * attribute declared by the entity or mapped superclass.
 * @param type  the type of the represented declared version
 *              attribute
 * @param <Y> The type of the represented declared version attribute
 * @return declared version attribute
 * @throws IllegalArgumentException if version attribute of the
 *         type is not declared in the identifiable type
 */
@Nonnull
<Y> SingularAttribute<X, Y> getDeclaredVersion(Class<Y> type);

/**
 * Return the identifiable type that corresponds to the most
 * specific mapped superclass or entity extended by the entity
 * or mapped superclass.
 * @return supertype of identifiable type or null if no
 *         such supertype
 */
@Nullable
IdentifiableType<? super X> getSupertype();

/**
 * Whether the identifiable type has a single id attribute.
 * Returns true for a simple id or embedded id; returns false
 * for an idclass.
 * @return boolean indicating whether the identifiable
 *         type has a single id attribute
 */
boolean hasSingleIdAttribute();

/**
 * Whether the identifiable type has a version attribute.
 * @return boolean indicating whether the identifiable
 *         type has a version attribute
 */
boolean hasVersionAttribute();
```

```
    /**
     *  Return the attributes corresponding to the id class of the
     *  identifiable type.
     *  @return id attributes
     *  @throws IllegalArgumentException if the identifiable type
     *          does not have an id class
     */
    @Nonnull
    Set<SingularAttribute<? super X, ?>> getIdClassAttributes();

    /**
     * Return the type that represents the type of the id.
     * @return type of id
     */
    @Nonnull
    Type<?> getIdType();
}
```

D.10. ListAttribute

```
/**
 * Instances of the type {@code ListAttribute} represent persistent
 * {@link java.util.List}-valued attributes.
 *
 * @param <X> The type the represented List belongs to
 * @param <E> The element type of the represented List
 *
 * @since 2.0
 *
 */
public interface ListAttribute<X, E>
        extends PluralAttribute<X, java.util.List<E>, E> {}
```

D.11. ManagedType

```
import jakarta.annotation.Nonnull;

import java.util.Set;

/**
 *  Instances of the type {@code ManagedType} represent entity, mapped
 *  superclass, and embeddable types.
 *
 *  @param <X> The represented type.
 *
 *  @since 2.0
 *
 */
public interface ManagedType<X> extends Type<X> {

    /**
     * Return the attributes of the managed type.
     * @return attributes of the managed type
     */
    @Nonnull
    Set<Attribute<? super X, ?>> getAttributes();

    /**
     * Return the attributes declared by the managed type.
     * Returns empty set if the managed type has no declared
     * attributes.
     * @return declared attributes of the managed type
     */
```

```
@Nonnull
Set<Attribute<X, ?>> getDeclaredAttributes();

/**
 * Return the single-valued attribute of the managed
 * type that corresponds to the specified name and Java type.
 * @param name  the name of the represented attribute
 * @param type  the type of the represented attribute
 * @param <Y> The type of the represented attribute
 * @return single-valued attribute with given name and type
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not present in the managed type
 */
@Nonnull
<Y> SingularAttribute<? super X, Y> getSingularAttribute(@Nonnull String name,
                                                         @Nonnull Class<Y> type);

/**
 * Return the single-valued attribute declared by the
 * managed type that corresponds to the specified name and
 * Java type.
 * @param name  the name of the represented attribute
 * @param type  the type of the represented attribute
 * @param <Y> The type of the represented attribute
 * @return declared single-valued attribute of the given
 *         name and type
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not declared in the managed type
 */
@Nonnull
<Y> SingularAttribute<X, Y> getDeclaredSingularAttribute(@Nonnull String name,
                                                         @Nonnull Class<Y> type);

/**
 * Return the single-valued attributes of the managed type.
 * Returns empty set if the managed type has no single-valued
 * attributes.
 * @return single-valued attributes
 */
@Nonnull
Set<SingularAttribute<? super X, ?>> getSingularAttributes();

/**
 * Return the single-valued attributes declared by the managed
 * type.
 * Returns empty set if the managed type has no declared
 * single-valued attributes.
 * @return declared single-valued attributes
 */
@Nonnull
Set<SingularAttribute<X, ?>> getDeclaredSingularAttributes();

/**
 * Return the Collection-valued attribute of the managed type
 * that corresponds to the specified name and Java element type.
 * @param name  the name of the represented attribute
 * @param elementType  the element type of the represented
 *                     attribute
 * @param <E> The element type of the represented collection
 * @return CollectionAttribute of the given name and element
 *         type
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not present in the managed type
 */
@Nonnull
<E> CollectionAttribute<? super X, E> getCollection(@Nonnull String name,
                                                    @Nonnull Class<E> elementType);
```

```java
/**
 * Return the Collection-valued attribute declared by the
 * managed type that corresponds to the specified name and Java
 * element type.
 * @param name  the name of the represented attribute
 * @param elementType  the element type of the represented
 *                     attribute
 * @param <E> The element type of the represented collection
 * @return declared {@code CollectionAttribute} of the given name and
 *         element type
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not declared in the managed type
 */
@Nonnull
<E> CollectionAttribute<X, E> getDeclaredCollection(@Nonnull String name,
                                                    @Nonnull Class<E> elementType);

/**
 * Return the Set-valued attribute of the managed type that
 * corresponds to the specified name and Java element type.
 * @param name  the name of the represented attribute
 * @param elementType  the element type of the represented
 *                     attribute
 * @param <E> The element type of the represented set
 * @return SetAttribute of the given name and element type
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not present in the managed type
 */
@Nonnull
<E> SetAttribute<? super X, E> getSet(@Nonnull String name,
                                      @Nonnull Class<E> elementType);

/**
 * Return the Set-valued attribute declared by the managed type
 * that corresponds to the specified name and Java element type.
 * @param name  the name of the represented attribute
 * @param elementType  the element type of the represented
 *                     attribute
 * @param <E> The element type of the represented set
 * @return declared SetAttribute of the given name and
 *         element type
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not declared in the managed type
 */
@Nonnull
<E> SetAttribute<X, E> getDeclaredSet(@Nonnull String name,
                                      @Nonnull Class<E> elementType);

/**
 * Return the List-valued attribute of the managed type that
 * corresponds to the specified name and Java element type.
 * @param name  the name of the represented attribute
 * @param elementType  the element type of the represented
 *                     attribute
 * @param <E> The element type of the represented list
 * @return ListAttribute of the given name and element type
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not present in the managed type
 */
@Nonnull
<E> ListAttribute<? super X, E> getList(@Nonnull String name,
                                        @Nonnull Class<E> elementType);

/**
 * Return the List-valued attribute declared by the managed
 * type that corresponds to the specified name and Java
```

```
 * element type.
 * @param name  the name of the represented attribute
 * @param elementType  the element type of the represented
 *                     attribute
 * @param <E> The element type of the represented list
 * @return declared ListAttribute of the given name and
 *         element type
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not declared in the managed type
 */
@Nonnull
<E> ListAttribute<X, E> getDeclaredList(@Nonnull String name,
                                        @Nonnull Class<E> elementType);

/**
 * Return the Map-valued attribute of the managed type that
 * corresponds to the specified name and Java key and value
 * types.
 * @param name  the name of the represented attribute
 * @param keyType  the key type of the represented attribute
 * @param valueType  the value type of the represented attribute
 * @param <K> The key type of the represented map
 * @param <V> The value type of the represented map
 * @return MapAttribute of the given name and key and value
 * types
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not present in the managed type
 */
@Nonnull
<K, V> MapAttribute<? super X, K, V> getMap(@Nonnull String name,
                                            @Nonnull Class<K> keyType,
                                            @Nonnull Class<V> valueType);

/**
 * Return the Map-valued attribute declared by the managed
 * type that corresponds to the specified name and Java key
 * and value types.
 * @param name  the name of the represented attribute
 * @param keyType  the key type of the represented attribute
 * @param valueType  the value type of the represented attribute
 * @param <K> The key type of the represented map
 * @param <V> The value type of the represented map
 * @return declared MapAttribute of the given name and key
 *         and value types
 * @throws IllegalArgumentException if attribute of the given
 *         name and type is not declared in the managed type
 */
@Nonnull
<K, V> MapAttribute<X, K, V> getDeclaredMap(@Nonnull String name,
                                            @Nonnull Class<K> keyType,
                                            @Nonnull Class<V> valueType);

/**
 * Return all multi-valued attributes (Collection-, Set-,
 * List-, and Map-valued attributes) of the managed type.
 * Returns empty set if the managed type has no multi-valued
 * attributes.
 * @return Collection-, Set-, List-, and Map-valued attributes
 */
@Nonnull
Set<PluralAttribute<? super X, ?, ?>> getPluralAttributes();

/**
 * Return all multi-valued attributes (Collection-, Set-,
 * List-, and Map-valued attributes) declared by the
 * managed type.
 * Returns empty set if the managed type has no declared
```

```
     * multivalued attributes.
     * @return declared Collection-, Set-, List-, and Map-valued
     *         attributes
     */
    @Nonnull
    Set<PluralAttribute<X, ?, ?>> getDeclaredPluralAttributes();


//String-based:

    /**
     * Return the attribute of the managed
     * type that corresponds to the specified name.
     * @param name  the name of the represented attribute
     * @return attribute with given name
     * @throws IllegalArgumentException if attribute of the given
     *         name is not present in the managed type
     */
    @Nonnull
    Attribute<? super X, ?> getAttribute(@Nonnull String name);

    /**
     * Return the attribute declared by the managed
     * type that corresponds to the specified name.
     * @param name  the name of the represented attribute
     * @return attribute with given name
     * @throws IllegalArgumentException if attribute of the given
     *         name is not declared in the managed type
     */
    @Nonnull
    Attribute<X, ?> getDeclaredAttribute(@Nonnull String name);

    /**
     * Return the single-valued attribute of the managed type that
     * corresponds to the specified name.
     * @param name  the name of the represented attribute
     * @return single-valued attribute with the given name
     * @throws IllegalArgumentException if attribute of the given
     *         name is not present in the managed type
     */
    @Nonnull
    SingularAttribute<? super X, ?> getSingularAttribute(@Nonnull String name);

    /**
     * Return the single-valued attribute declared by the managed
     * type that corresponds to the specified name.
     * @param name  the name of the represented attribute
     * @return declared single-valued attribute of the given
     *         name
     * @throws IllegalArgumentException if attribute of the given
     *         name is not declared in the managed type
     */
    @Nonnull
    SingularAttribute<X, ?> getDeclaredSingularAttribute(@Nonnull String name);

    /**
     * Return the Collection-valued attribute of the managed type
     * that corresponds to the specified name.
     * @param name  the name of the represented attribute
     * @return CollectionAttribute of the given name
     * @throws IllegalArgumentException if attribute of the given
     *         name is not present in the managed type
     */
    @Nonnull
    CollectionAttribute<? super X, ?> getCollection(@Nonnull String name);

    /**
```

```
 * Return the Collection-valued attribute declared by the
 * managed type that corresponds to the specified name.
 * @param name  the name of the represented attribute
 * @return declared CollectionAttribute of the given name
 * @throws IllegalArgumentException if attribute of the given
 *         name is not declared in the managed type
 */
@Nonnull
CollectionAttribute<X, ?> getDeclaredCollection(@Nonnull String name);

/**
 * Return the Set-valued attribute of the managed type that
 * corresponds to the specified name.
 * @param name  the name of the represented attribute
 * @return SetAttribute of the given name
 * @throws IllegalArgumentException if attribute of the given
 *         name is not present in the managed type
 */
@Nonnull
SetAttribute<? super X, ?> getSet(@Nonnull String name);

/**
 * Return the Set-valued attribute declared by the managed type
 * that corresponds to the specified name.
 * @param name  the name of the represented attribute
 * @return declared SetAttribute of the given name
 * @throws IllegalArgumentException if attribute of the given
 *         name is not declared in the managed type
 */
@Nonnull
SetAttribute<X, ?> getDeclaredSet(@Nonnull String name);

/**
 * Return the List-valued attribute of the managed type that
 * corresponds to the specified name.
 * @param name  the name of the represented attribute
 * @return ListAttribute of the given name
 * @throws IllegalArgumentException if attribute of the given
 *         name is not present in the managed type
 */
@Nonnull
ListAttribute<? super X, ?> getList(@Nonnull String name);

/**
 * Return the List-valued attribute declared by the managed
 * type that corresponds to the specified name.
 * @param name  the name of the represented attribute
 * @return declared ListAttribute of the given name
 * @throws IllegalArgumentException if attribute of the given
 *         name is not declared in the managed type
 */
@Nonnull
ListAttribute<X, ?> getDeclaredList(@Nonnull String name);

/**
 * Return the Map-valued attribute of the managed type that
 * corresponds to the specified name.
 * @param name  the name of the represented attribute
 * @return MapAttribute of the given name
 * @throws IllegalArgumentException if attribute of the given
 *         name is not present in the managed type
 */
@Nonnull
MapAttribute<? super X, ?, ?> getMap(@Nonnull String name);

/**
 * Return the Map-valued attribute declared by the managed
```

```
     * type that corresponds to the specified name.
     * @param name  the name of the represented attribute
     * @return declared MapAttribute of the given name
     * @throws IllegalArgumentException if attribute of the given
     *         name is not declared in the managed type
     */
    @Nonnull
    MapAttribute<X, ?, ?> getDeclaredMap(@Nonnull String name);
}
```

D.12. MapAttribute

```
import jakarta.annotation.Nonnull;

/**
 * Instances of the type {@code MapAttribute} represent
 * persistent {@link java.util.Map}-valued attributes.
 *
 * @param <X> The type the represented Map belongs to
 * @param <K> The type of the key of the represented Map
 * @param <V> The type of the value of the represented Map
 *
 * @since 2.0
 *
 */
public interface MapAttribute<X, K, V>
    extends PluralAttribute<X, java.util.Map<K, V>, V> {

    /**
     * Return the Java type of the map key.
     * @return Java key type
     */
    @Nonnull
    Class<K> getKeyJavaType();

    /**
     * Return the type representing the key type of the map.
     * @return type representing key type
     */
    @Nonnull
    Type<K> getKeyType();
}
```

D.13. MappedSuperclassType

```
package jakarta.persistence.metamodel;

/**
 * An instance of the type {@code MappedSuperclassType} represents a
 * {@linkplain jakarta.persistence.MappedSuperclass mapped superclass}
 * type.
 *
 * @param <X> The represented mapped superclass type
 *
 * @since 2.0
 */
public interface MappedSuperclassType<X> extends IdentifiableType<X> {}
```

D.14. PluralAttribute

```
import jakarta.annotation.Nonnull;
```

```java
/**
 * Instances of the type {@code PluralAttribute} represent
 * persistent collection-valued attributes.
 *
 * @param <X> The type the represented collection belongs to
 * @param <C> The type of the represented collection
 * @param <E> The element type of the represented collection
 *
 * @since 2.0
 */
public interface PluralAttribute<X, C, E>
        extends Attribute<X, C>, Bindable<E> {

    enum CollectionType {

        /** Collection-valued attribute */
        COLLECTION,

        /** Set-valued attribute */
        SET,

        /** List-valued attribute */
        LIST,

        /** Map-valued attribute */
        MAP
    }

    /**
     * Return the collection type.
     * @return collection type
     */
    @Nonnull
    CollectionType getCollectionType();

    /**
     * Return the type representing the element type of the
     * collection.
     * @return element type
     */
    @Nonnull
    Type<E> getElementType();
}
```

D.15. SetAttribute

```java
/**
 * Instances of the type {@code SetAttribute} represent
 * persistent {@link java.util.Set}-valued attributes.
 *
 * @param <X> The type the represented Set belongs to
 * @param <E> The element type of the represented Set
 *
 * @since 2.0
 */
public interface SetAttribute<X, E>
        extends PluralAttribute<X, java.util.Set<E>, E> {}
```

D.16. SingularAttribute

```java
package jakarta.persistence.metamodel;
```

```java
import jakarta.annotation.Nonnull;

/**
 * Instances of the type {@code SingularAttribute} represents persistent
 * single-valued properties or fields.
 *
 * @param <X> The type containing the represented attribute
 * @param <T> The type of the represented attribute
 *
 * @since 2.0
 */
public interface SingularAttribute<X, T>
        extends Attribute<X, T>, Bindable<T> {

    /**
     * Is the attribute an id attribute? This method returns true
     * if the attribute is a {@linkplain jakarta.persistence.Id simple
     * id}, an {@linkplain jakarta.persistence.EmbeddedId embedded id},
     * or an attribute of an {@linkplain jakarta.persistence.IdClass
     * id class}.
     * @return boolean indicating whether the attribute is an id
     */
    boolean isId();

    /**
     * Is the attribute a {@linkplain jakarta.persistence.Version
     * version} attribute?
     * @return boolean indicating whether the attribute is a
     *         version attribute
     */
    boolean isVersion();

    /**
     * Can the attribute be null?
     * @return boolean indicating whether the attribute can
     *        be null
     * @see jakarta.persistence.Basic#optional
     */
    boolean isOptional();

    /**
     * Return the type that represents the type of the attribute.
     * @return type of attribute
     */
    @Nonnull
    Type<T> getType();
}
```

D.17. Type

```java
import jakarta.annotation.Nonnull;

/**
 * An instance of the type {@code Type} represents a persistent
 * object or attribute type.
 *
 * @param <X>  The type of the represented object or attribute
 *
 * @since 2.0
 */
public interface Type<X> {

    enum PersistenceType {

        /** Entity class */
```

```
        ENTITY,

        /** Embeddable class */
        EMBEDDABLE,

        /** Mapped superclass */
        MAPPED_SUPERCLASS,

        /** Basic type */
        BASIC
    }

    /**
     * Return the persistence type.
     * @return persistence type
     */
    @Nonnull
    PersistenceType getPersistenceType();

    /**
     * Return the represented Java type.
     * @return Java type
     */
    @Nonnull
    Class<X> getJavaType();
}
```

Appendix E: Persistence SPI Interfaces

The following APIs are defined in the package `jakarta.persistence.spi`.

E.1. `ClassTransformer`

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

import java.security.ProtectionDomain;

/**
 * A persistence provider supplies an instance of this
 * interface to the {@link PersistenceUnitInfo#addTransformer}
 * method. The supplied transformer instance will get
 * called to transform entity class files when they are
 * loaded or redefined. The transformation occurs before
 * the class is defined by the JVM.
 *
 * @apiNote This is an SPI interface forming part of the Jakarta EE
 * container / persistence provider contract. It is not intended for
 * direct use by application programs.
 *
 * @since 1.0
 */
public interface ClassTransformer {

    /**
     * Invoked when a class is being loaded or redefined.
     * The implementation of this method may transform the
     * supplied class file and return a new replacement class
     * file.
     *
     * @param loader  the defining loader of the class to be
     *        transformed, may be null if the bootstrap loader
     * @param className  the name of the class in the internal form
     *        of fully qualified class and interface names
     * @param classBeingRedefined  if this is a redefine, the
     *        class being redefined, otherwise null
     * @param protectionDomain  the protection domain of the
     *        class being defined or redefined
     * @param classfileBuffer  the input byte buffer in class
     *        file format - must not be modified
     * @return a well-formed class file buffer (the result of
     *         the transform), or null if no transform is performed
     * @throws TransformerException  if the input does
     *         not represent a well-formed class file
     */
    @Nullable
    byte[] transform(@Nullable ClassLoader loader,
                     @Nonnull String className,
                     @Nullable Class<?> classBeingRedefined,
                     @Nonnull ProtectionDomain protectionDomain,
                     @Nonnull byte[] classfileBuffer)
        throws TransformerException;
}
```

E.2. `LoadState`

```
/**
 * Load states returned by the {@link ProviderUtil} SPI methods.
 * @since 2.0
```

```
 *
 */
public enum LoadState {
    /** The state of the element is known to have been loaded. */
    LOADED,
    /** The state of the element is known not to have been loaded. */
    NOT_LOADED,
    /** The load state of the element cannot be determined. */
    UNKNOWN
}
```

E.3. PersistenceProvider

```
package jakarta.persistence.spi;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import jakarta.persistence.PersistenceConfiguration;
import jakarta.persistence.PersistenceException;
import java.util.Map;

/**
 * Interface implemented by the persistence provider.
 * <p>
 * In Java SE, the {@link Persistence} class calls:
 * <ul>
 * <li>{@link #createEntityManagerFactory(String, Map)} when the
 *     application program calls
 *     {@link Persistence#createEntityManagerFactory(String, Map)},
 * <li>{@link #createEntityManagerFactory(PersistenceConfiguration)}
 *    when the application program calls
 *    {@link PersistenceConfiguration#createEntityManagerFactory},
 * <li>{@link #generateSchema(String, Map)} when the application,
 *     program calls {@link Persistence#generateSchema(String, Map)},
 * <li>{@link #generateSchema(PersistenceConfiguration)} when the
 *     application program calls
 *     {@link PersistenceConfiguration#exportSchema()}, and
 * <li>{@link #getProviderUtil()} to obtain a {@link ProviderUtil}.
 * </ul>
 * <p>In a Jakarta EE environment, the container calls:
 * <ul>
 * <li>{@link #createContainerEntityManagerFactory(PersistenceUnitInfo, Map)}
 *     to create an {@link EntityManagerFactory} for a container-managed
 *     persistence unit.
 * <li>{@link #generateSchema(PersistenceUnitInfo, Map)} to delegate
 *     DDL schema management to the persistence provider, and
 *     {@link #getClassTransformer(PersistenceUnitInfo, Map)} to
 *     obtain a provider-supplied {@link ClassTransformer}.
 * </ul>
 *
 * @apiNote This is an SPI interface forming part of the Jakarta EE
 * container / persistence provider contract. It is not intended for
 * direct use by application programs.
 *
 * @since 1.0
 */
public interface PersistenceProvider {

    /**
     * Called by {@link Persistence} class to create a new
     * {@link EntityManagerFactory} for the persistence unit with the
     * given name, optionally specifying configuration properties which
     * override the settings specified in the {@code persistence.xml}
```

```
 * file for the given persistence unit.
 *
 * @param unitName the name of the persistence unit
 * @param properties property settings for use by the persistence
 *                   provider. These properties may be used to
 *                   specify values for properties not specified
 *                   in the {@code persistence.xml} file for the
 *                   given persistence unit or to override the
 *                   values specified by the corresponding elements
 *                   of the {@code persistence.xml} file. The map
 *                   may be null when no properties are specified.
 * @return a newly created {@link EntityManagerFactory} for the
 *         named persistence unit, or null if the persistence unit
 *         configuration specified a different persistence provider
 *
 * @see Persistence#createEntityManagerFactory(String, Map)
 */
@Nonnull
EntityManagerFactory createEntityManagerFactory(@Nonnull String unitName,
                                                @Nullable Map<?, ?> properties);

/**
 * Called by {@link Persistence} class to create a new
 * {@link EntityManagerFactory} for a persistence unit with the
 * given programmatic configuration.
 *
 * @param configuration the configuration of the persistence unit
 * @return a newly created {@link EntityManagerFactory} for the
 *         persistence unit with the given configuration, or null
 *         if the persistence unit configuration specified a
 *         different persistence provider
 * @throws IllegalStateException if any required configuration
 *                               is missing
 *
 * @see Persistence#createEntityManagerFactory(PersistenceConfiguration)
 *
 * @since 3.2
 */
@Nonnull
EntityManagerFactory createEntityManagerFactory(@Nonnull PersistenceConfiguration configuration);

/**
 * Called by the Jakarta EE container to create a new
 * {@link EntityManagerFactory} for the persistence unit with the
 * given {@linkplain PersistenceUnitInfo metadata}, optionally
 * specifying configuration properties which override the settings
 * specified in the given {@link PersistenceUnitInfo}.
 * <ul>
 * <li>If a Jakarta Validation provider is present in the classpath,
 * the container must pass the {@code ValidatorFactory} instance
 * as an entry in the given map with the key
 * {@link jakarta.persistence.Persistence.ValidationProperties#VALIDATION_FACTORY}.
 * <li>If the containing archive is a bean archive, the container
 * must pass the {@code BeanManager} instance as an entry in the
 * given map with the key
 * {@link jakarta.persistence.Persistence.BeanManagementProperties#BEAN_MANAGER}.
 * </ul>
 *
 * @param info metadata describing the persistence unit
 * @param properties integration-level property settings for use by
 *                   the persistence provider. The given properties
 *                   may include properties to control schema
 *                   generation. The map may be null when no
 *                   properties are specified.
 * @return a newly created {@link EntityManagerFactory} for the
 *         persistence unit described by the given metadata
 */
```

```
@Nonnull
EntityManagerFactory createContainerEntityManagerFactory(@Nonnull PersistenceUnitInfo info,
                                                         @Nullable Map<?, ?> properties);

/**
 * Create database schemas and/or tables and/or create DDL scripts,
 * as determined by the supplied properties.
 * <p>
 * Called by the container when schema generation is to occur as a
 * separate phase from the creation of the entity manager factory.
 *
 * @param info metadata describing the persistence unit
 * @param properties property settings for schema generation;
 *                   these may include provider-specific
 *                   properties
 * @throws PersistenceException if insufficient or inconsistent
 *         configuration information is provided or if schema
 *         generation otherwise fails
 *
 * @since 2.1
 */
void generateSchema(@Nonnull PersistenceUnitInfo info,
                    @Nullable Map<?, ?> properties);

/**
 * Create database schemas and/or tables and/or create DDL scripts,
 * as determined by the supplied properties.
 * <p>
 * Called by the {@link Persistence} class when schema generation
 * is to occur as a separate phase from creation of the entity
 * manager factory.
 *
 * @param persistenceUnitName the name of the persistence unit
 * @param properties property settings for schema generation;
 *                   these may include provider-specific
 *                   properties. The values of these properties
 *                   override any values configured elsewhere.
 * @return true if the schema was generated, otherwise false
 * @throws PersistenceException if insufficient or inconsistent
 *         configuration information is provided or if schema
 *         generation otherwise fails
 *
 * @since 2.1
 */
boolean generateSchema(@Nonnull String persistenceUnitName,
                       @Nullable Map<?, ?> properties);

/**
 * Create database schemas and/or tables and/or create DDL scripts,
 * as determined by the supplied properties.
 * <p>
 * Called by the {@link Persistence} class when schema generation
 * is to occur as a separate phase from creation of the entity
 * manager factory.
 *
 * @param configuration the configuration of the persistence unit
 * @return true if the schema was generated, otherwise false
 * @throws PersistenceException if insufficient or inconsistent
 *         configuration information is provided or if schema
 *         generation otherwise fails
 *
 * @since 4.0
 */
boolean generateSchema(@Nonnull PersistenceConfiguration configuration);

/**
 * Return the utility interface implemented by the persistence
```

```
     * provider.
     * @return an instance of {@link ProviderUtil}
     *
     * @since 2.0
     */
    @Nonnull
    ProviderUtil getProviderUtil();

    /**
     * Obtain a provider-supplied class transformer that is called
     * for every new class definition or class redefinition performed
     * by the class loader returned by
     * {@link PersistenceUnitInfo#getClassLoader}. The transformer
     * has no effect on the class loader returned by
     * {@link PersistenceUnitInfo#getNewTempClassLoader}.
     * <p>
     * A class is transformed exactly once within a given classloading
     * scope, regardless of how many persistence units it belongs to.
     * <p>The given instance of {@link PersistenceUnitInfo} may
     * return {@code null} when any of the accessor methods
     * {@link PersistenceUnitInfo#getClassLoader()},
     * {@link PersistenceUnitInfo#getJtaDataSource()}, or
     * {@link PersistenceUnitInfo#getNonJtaDataSource()} is called
     * by an implementation of this method.
     * <p>
     * If the container calls this method before invoking
     * {@link #createContainerEntityManagerFactory} to create the
     * {@link EntityManagerFactory}, then the transformer returned
     * by this method is used instead of any transformer registered
     * via {@link PersistenceUnitInfo#addTransformer}. The container
     * is not required to call this method.
     *
     * @return a provider-supplied transformer that is later invoked
     *         by the container when a managed class belonging to the
     *         persistence unit is defined or redefined
     * @param info metadata describing the persistence unit
     * @param properties integration-level property settings for use
     *                   by the persistence provider, which will not
     *                   usually contain a {@code ValidatorFactory}
     *                   or {@code BeanManager}.
     * @since 4.0
     */
    @Nonnull
    ClassTransformer getClassTransformer(@Nonnull PersistenceUnitInfo info,
                                         @Nullable Map<?, ?> properties);
}
```

E.4. PersistenceProviderResolver

```
import jakarta.annotation.Nonnull;

import java.util.List;

/**
 * Provides a list of {@linkplain PersistenceProvider persistence
 * providers} available in the runtime environment.
 *
 * <p> Implementations must be thread-safe.
 *
 * <p> Note that the {@link #getPersistenceProviders} method can
 * potentially be called many times: it is recommended that the
 * implementation of this method make use of caching.
 *
 * @see PersistenceProvider
 * @since 2.0
```

```
 */
public interface PersistenceProviderResolver {

    /**
     * Returns a list of the {@linkplain PersistenceProvider
     * persistence provider} implementations available in the
     * runtime environment.
     *
     * @return list of the persistence providers available
     *         in the environment
     */
    @Nonnull
    List<PersistenceProvider> getPersistenceProviders();

    /**
     * Clear cache of providers.
     */
    void clearCachedProviders();
}
```

E.5. PersistenceProviderResolverHolder

```
package jakarta.persistence.spi;

import java.lang.ref.ReferenceQueue;
import java.lang.ref.SoftReference;
import java.lang.ref.WeakReference;
import java.lang.System.Logger;
import java.lang.System.Logger.Level;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.ServiceConfigurationError;
import java.util.ServiceLoader;


/**
 * Holds the global {@link PersistenceProviderResolver} instance.
 * If no {@code PersistenceProviderResolver} is set by the environment,
 * the default {@code PersistenceProviderResolver} is used.
 *
 * <p>Enable {@code "jakarta.persistence.spi"} logger to show diagnostic
 * information.
 *
 * <p>Implementations must be thread-safe.
 *
 * @since 2.0
 */
public class PersistenceProviderResolverHolder {

    private static PersistenceProviderResolver singleton = new DefaultPersistenceProviderResolver();

    /**
     * Returns the current persistence provider resolver.
     *
     * @return the current persistence provider resolver
     */
    public static PersistenceProviderResolver getPersistenceProviderResolver() {
        return singleton;
    }

    /**
     * Defines the persistence provider resolver used.
     *
```

```
 * @param resolver persistence provider resolver to be used.
 */
public static void setPersistenceProviderResolver(PersistenceProviderResolver resolver) {
    singleton = resolver == null ? new DefaultPersistenceProviderResolver() : resolver;
}

/**
 * Default provider resolver class to use when none is explicitly set.
 *
 * <p>Uses service loading mechanism as described in the Jakarta Persistence
 * specification. A {@code ServiceLoader.load()} call is made with the current
 * context classloader to find the service provider files on the classpath.
 */
private static class DefaultPersistenceProviderResolver implements PersistenceProviderResolver {

    /**
     * Cached list of available providers cached by CacheKey to ensure
     * there is not potential for provider visibility issues.
     */
    private final Map<CacheKey, PersistenceProviderReference> providers = new HashMap<>();

    /**
     * Queue for reference objects referring to class loaders or persistence providers.
     */
    private static final ReferenceQueue<Object> referenceQueue = new ReferenceQueue<>();

    public List<PersistenceProvider> getPersistenceProviders() {
        // Before we do the real loading work, see whether we need to
        // do some cleanup: If references to class loaders or
        // persistence providers have been nulled out, remove all related
        // information from the cache.
        processQueue();

        final var loader = Thread.currentThread().getContextClassLoader();
        final var cacheKey = new CacheKey(loader);

        final var providerRef = providers.get(cacheKey);
        if (providerRef != null) {
            final var loadedProviders = providerRef.get();
            if (loadedProviders != null) {
                return loadedProviders;
            }
        }

        final var loadedProviders = loadPersistenceProviders(loader);
        final var newProviderRef = new PersistenceProviderReference(loadedProviders, referenceQueue, cacheKey);
        providers.put(cacheKey, newProviderRef);
        return loadedProviders;
    }

    private ArrayList<PersistenceProvider> loadPersistenceProviders(ClassLoader loader) {
        final var providers = new ArrayList<PersistenceProvider>();
        try {
            ServiceLoader.load(PersistenceProvider.class, loader).iterator()
                    .forEachRemaining( ip -> {
                        try {
                            providers.add(ip);
                        }
                        catch (ServiceConfigurationError sce) {
                            log(Level.TRACE, sce.toString());
                        }
                    } );
        }
        catch (ServiceConfigurationError sce) {
            log(Level.TRACE, sce.toString());
        }
```

```
        // If none are found we'll log the provider names for diagnostic
        // purposes.
        if (providers.isEmpty()) {
            log(Level.WARNING, "No valid providers found.");
        }

        return providers;
    }

    /**
     * Remove garbage collected cache keys and providers.
     */
    private void processQueue() {
        CacheKeyReference ref;
        while ((ref = (CacheKeyReference) referenceQueue.poll()) != null) {
            providers.remove(ref.getCacheKey());
        }
    }

    private static final String LOGGER_SUBSYSTEM = "jakarta.persistence.spi";

    private Logger logger;

    private void log(Level level, String message) {
        if (logger == null) {
            logger = System.getLogger(LOGGER_SUBSYSTEM);
        }
        logger.log(level, LOGGER_SUBSYSTEM + "::" + message);
    }

    /**
     * Clear all cached providers
     */
    public void clearCachedProviders() {
        providers.clear();
    }


    /**
     * The common interface to get a {@link CacheKey} implemented by
     * {@link LoaderReference} and {@link PersistenceProviderReference}.
     */
    private sealed interface CacheKeyReference
            permits LoaderReference, PersistenceProviderReference {
        CacheKey getCacheKey();
    }

    /**
     * Key used for cached persistence providers. The key checks
     * the class loader to determine if the persistence providers
     * is a match to the requested one. The loader may be null.
     */
    private final class CacheKey {

        /* Weak Reference to ClassLoader */
        private final LoaderReference loaderRef;

        /* Cached Hashcode */
        private int hashCodeCache;

        private CacheKey(ClassLoader loader) {
            loaderRef = loader == null ? null : new LoaderReference(loader, referenceQueue, this);
            calculateHashCode();
        }

        private ClassLoader getLoader() {
            return loaderRef != null ? loaderRef.get() : null;
```

```java
    }

    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        else if (!(other instanceof CacheKey otherEntry)) {
            return false;
        }
        else {
            try {
                // quick check to see if they are not equal
                if (hashCodeCache != otherEntry.hashCodeCache) {
                    return false;
                }
                // are refs (both non-null) or (both null)?
                else if (loaderRef == null) {
                    return otherEntry.loaderRef == null;
                }
                else {
                    var loader = loaderRef.get();
                    return otherEntry.loaderRef != null
                        // with a null reference we can no longer find
                        // out which class loader was referenced; so
                        // treat it as unequal
                        && loader != null
                        && loader == otherEntry.loaderRef.get();
                }
            }
            catch (NullPointerException | ClassCastException e) {
                return false;
            }
        }
    }

    @Override
    public int hashCode() {
        return hashCodeCache;
    }

    private void calculateHashCode() {
        var loader = getLoader();
        if (loader != null) {
            hashCodeCache = loader.hashCode();
        }
    }

    @Override
    public String toString() {
        return "CacheKey[" + getLoader() + ")]";
    }
}

/**
 * References to class loaders are weak references, so that they can be garbage
 * collected when nobody else is using them. {@link DefaultPersistenceProviderResolver}
 * has no reason to keep class loaders alive.
 */
private final class LoaderReference
        extends WeakReference<ClassLoader>
        implements CacheKeyReference {
    private final CacheKey cacheKey;

    private LoaderReference(
            ClassLoader referent,
            ReferenceQueue<? super ClassLoader> queue,
```

```
                CacheKey key) {
            super(referent, queue);
            cacheKey = key;
        }

        @Override
        public CacheKey getCacheKey() {
            return cacheKey;
        }
    }

    /**
     * References to persistence provider are soft references so that they can be garbage
     * collected when they have no hard references.
     */
    private final class PersistenceProviderReference
            extends SoftReference<List<PersistenceProvider>>
            implements CacheKeyReference {
        private final CacheKey cacheKey;

        private PersistenceProviderReference(
                List<PersistenceProvider> referent,
                ReferenceQueue<? super List<PersistenceProvider>> queue,
                CacheKey key) {
            super(referent, queue);
            cacheKey = key;
        }

        @Override
        public CacheKey getCacheKey() {
            return cacheKey;
        }
    }
  }
}
```

E.6. PersistenceUnitInfo

```
package jakarta.persistence.spi;

import javax.sql.DataSource;
import java.util.List;
import java.util.Properties;
import java.net.URL;

import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.FetchType;
import jakarta.persistence.PersistenceUnitTransactionType;
import jakarta.persistence.SharedCacheMode;
import jakarta.persistence.ValidationMode;

/**
 * Interface implemented by the container and used by the persistence
 * provider when creating an {@link EntityManagerFactory}.
 *
 * @apiNote This is an SPI interface forming part of the Jakarta EE
 * container / persistence provider contract. It is not intended for
 * direct use by application programs.
 *
 * @since 1.0
 */
public interface PersistenceUnitInfo {
```

```java
/**
 * Returns the name of the persistence unit. Corresponds to the
 * {@code name} attribute in the {@code persistence.xml} file.
 * @return the name of the persistence unit
 */
@Nonnull
String getPersistenceUnitName();

/**
 * Returns the fully qualified name of the persistence provider
 * implementation class. Corresponds to the {@code provider} element
 * in the {@code persistence.xml} file.
 * @return  the fully qualified name of the persistence provider
 * implementation class
 */
@Nullable
String getPersistenceProviderClassName();

/**
 * Returns the fully-qualified class name of an annotation annotated
 * {@code Scope} or {@code NormalScope}. Corresponds to the {@code scope}
 * element in {@code persistence.xml}.
 * @return  the fully-qualified class name of the scope annotation,
 *          or null if no scope was explicitly specified
 */
@Nullable
String getScopeAnnotationName();

/**
 * Returns the fully-qualified class names of annotations annotated
 * {@code Qualifier}. Corresponds to the {@code qualifier} element in
 * {@code persistence.xml}.
 * @return  the fully-qualified class names of the qualifier annotations,
 *          or an empty list if no qualifier annotations were explicitly
 *          specified
 */
@Nullable
List<String> getQualifierAnnotationNames();

/**
 * Returns the transaction type of the entity managers created by
 * the {@link EntityManagerFactory}. The transaction type corresponds
 * to the {@code transaction-type} attribute in the {@code persistence.xml}
 * file.
 * @return  transaction type of the entity managers created
 * by the EntityManagerFactory
 */
@Nonnull
PersistenceUnitTransactionType getTransactionType();

/**
 * Returns the JTA-enabled data source to be used by the
 * persistence provider. The data source corresponds to the
 * {@code jta-data-source} element in the {@code persistence.xml}
 * file or is provided at deployment or by the container.
 * @return the JTA-enabled data source to be used by the
 * persistence provider
 */
@Nullable
DataSource getJtaDataSource();

/**
 * Returns the non-JTA-enabled data source to be used by the
 * persistence provider for accessing data outside a JTA
 * transaction. The data source corresponds to the named
 * {@code non-jta-data-source} element in the {@code persistence.xml}
 * file or provided at deployment or by the container.
```

```
 * @return the non-JTA-enabled data source to be used by the
 * persistence provider for accessing data outside a JTA
 * transaction
 */
@Nullable
DataSource getNonJtaDataSource();

/**
 * Returns the list of the names of the mapping files that the
 * persistence provider must load to determine the mappings for
 * the entity classes. The mapping files must be in the standard
 * XML mapping format, be uniquely named and be resource-loadable
 * from the application classpath.  Each mapping file name
 * corresponds to a {@code mapping-file} element in the
 * {@code persistence.xml} file.
 * @return the list of mapping file names that the persistence
 * provider must load to determine the mappings for the entity
 * classes
 */
@Nonnull
List<String> getMappingFileNames();

/**
 * Returns a list of URLs for the jar files or exploded jar
 * file directories that the persistence provider must examine
 * for managed classes of the persistence unit. Each URL
 * corresponds to a {@code jar-file} element in the
 * {@code persistence.xml} file. A URL will either be a
 * file: URL referring to a jar file or referring to a directory
 * that contains an exploded jar file, or some other URL from
 * which an InputStream in jar format can be obtained.
 * @return a list of URL objects referring to jar files or
 * directories
 */
@Nonnull
List<URL> getJarFileUrls();

/**
 * Returns the URL for the jar file or directory that is the
 * root of the persistence unit. (If the persistence unit is
 * rooted in the WEB-INF/classes directory, this is the URL
 * of that directory.)
 * The URL will either be a file: URL referring to a jar file
 * or referring to a directory that contains an exploded jar
 * file, or some other URL from which an InputStream in jar
 * format can be obtained.
 * @return a URL referring to a jar file or directory
 */
@Nonnull
URL getPersistenceUnitRootUrl();

/**
 * Returns the class names listed by {@code class} elements
 * of the {@code persistence.xml} file.
 * @return the class names listed in {@code persistence.xml}
 */
@Nonnull
List<String> getManagedClassNames();

/**
 * Returns the names of all classes (compiled Java types,
 * module descriptors, and package descriptors) belonging
 * to the persistence unit, including classes:
 * <ul>
 * <li>named explicitly in the {@code persistence.xml}
 *     file,
 * <li>named explicitly in {@code META-INF/orm.xml} or
```

```
 *     in an XML mapping file listed in
 *     {@code persistence.xml}, or
 * <li>discovered by the container via scanning the
 *     archive containing the {@code persistence.xml}
 *     file and all archives referenced by
 *     {@code <jar-file>} elements of the persistence
 *     unit definiton.
 * </ul>
 * <p>A class is discoverable via scanning if it bears
 * a {@linkplain Discoverable discoverable annotation}.
 * Discoverable classes include entity classes, mapped
 * superclasses, embeddable classes, attribute converter
 * classes, and every type declaring a named query,
 * named statement, named stored procedure query, or
 * SQL result set mapping. Classes bearing custom
 * discoverable annotation types are also discoverable.
 * @return the list of names of all classes belonging
 * to the persistence unit, including types, module
 * descriptors, and package descriptors
 * @see Discoverable
 * @since 4.0
 */
@Nonnull
List<String> getAllClassNames();

/**
 * Returns whether classes in the root of the persistence unit
 * that have not been explicitly listed are to be included in the
 * set of managed classes. This value corresponds to the
 * {@code exclude-unlisted-classes} element in the
 * {@code persistence.xml} file.
 * @return whether classes in the root of the persistence
 * unit that have not been explicitly listed are to be
 * included in the set of managed classes
 */
boolean excludeUnlistedClasses();

/**
 * Returns the specification of how the provider must use
 * a second-level cache for the persistence unit.
 * The result of this method corresponds to the {@code shared-cache-mode}
 * element in the {@code persistence.xml} file.
 * @return the second-level cache mode that must be used by the
 * provider for the persistence unit
 *
 * @since 2.0
 */
@Nonnull
SharedCacheMode getSharedCacheMode();

/**
 * Returns the validation mode to be used by the persistence
 * provider for the persistence unit.  The validation mode
 * corresponds to the {@code validation-mode} element in the
 * {@code persistence.xml} file.
 * @return the validation mode to be used by the
 * persistence provider for the persistence unit
 *
 * @since 2.0
 */
@Nonnull
ValidationMode getValidationMode();

/**
 * Returns the {@linkplain FetchType#DEFAULT default fetch type}
 * for one-to-one and many-to-one associations. The default fetch
 * type corresponds to the {@code default-to-one-fetch-type}
```

```
 * element in the {@code persistence.xml} file.
 * @return the default fetch type for one-to-one and many-to-one
 *         associations for the persistence unit
 *
 * @since 4.0
 */
@Nonnull
FetchType getDefaultToOneFetchType();

/**
 * Returns a properties object. Each property corresponds to a
 * {@code property} element in the {@code persistence.xml} file
 * or to a property set by the container.
 * @return Properties object
 */
@Nonnull
Properties getProperties();

/**
 * Returns the schema version of the {@code persistence.xml} file.
 * @return {@code persistence.xml} schema version
 *
 * @since 2.0
 */
@Nonnull
String getPersistenceXMLSchemaVersion();

/**
 * Returns ClassLoader that the provider may use to load any
 * classes, resources, or open URLs.
 * @return ClassLoader that the provider may use to load any
 * classes, resources, or open URLs
 */
@Nonnull
ClassLoader getClassLoader();

/**
 * Add a transformer supplied by the provider that is called for
 * every new class definition or class redefinition that gets
 * loaded by the loader returned by the
 * {@link PersistenceUnitInfo#getClassLoader} method. The
 * transformer has no effect on the result returned by the
 * {@link PersistenceUnitInfo#getNewTempClassLoader} method.
 * Classes are only transformed once within the same classloading
 * scope, regardless of how many persistence units they may be
 * a part of.
 * <p>If the container previously called
 * {@link PersistenceProvider#getClassTransformer} with this
 * {@code PersistenceUnitInfo}, then this method has no effect.
 * @param transformer  provider-supplied transformer that the
 * container invokes at class-(re)definition time
 */
void addTransformer(@Nonnull ClassTransformer transformer);

/**
 * Return a new instance of a {@link ClassLoader} that the provider
 * may use to temporarily load any classes, resources, or open
 * URLs. The scope and classpath of this loader is exactly the
 * same as that of the loader returned by {@link
 * PersistenceUnitInfo#getClassLoader}. None of the classes loaded
 * by this class loader are visible to application components. The
 * provider may only use this {@code ClassLoader} within the scope
 * of the {@link PersistenceProvider#createContainerEntityManagerFactory}
 * call.
 * @return temporary {@code ClassLoader} with same visibility as
 * current loader
 */
```

```
    @Nonnull
    ClassLoader getNewTempClassLoader();
}
```

E.7. ProviderUtil

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.FetchType;

/**
 * Utility interface implemented by the persistence provider. This
 * interface is invoked by the {@link jakarta.persistence.PersistenceUtil}
 * implementation to determine the load status of an entity or entity
 * attribute.
 *
 * @apiNote This is an SPI interface forming part of the Jakarta EE
 * container / persistence provider contract. It is not intended for
 * direct use by application programs.
 *
 * @since 2.0
 */
public interface ProviderUtil {

    /**
     * If the provider determines that the entity has been provided by
     * itself and that the state of the specified attribute has been loaded,
     * this method returns {@link LoadState#LOADED}.
     * <p> If the provider determines that the entity has been provided
     * by itself and that either entity attributes with {@link FetchType#EAGER}
     * have not been loaded or that the state of the specified attribute has
     * not been loaded, this method returns {@link LoadState#NOT_LOADED}.
     * <p> If a provider cannot determine the load state, this method
     * returns {@link LoadState#UNKNOWN}.
     * <p> The provider's implementation of this method must not obtain a
     * reference to an attribute value, as this could trigger the loading
     * of entity state if the entity has been provided by a different
     * provider.
     * @param entity  entity instance
     * @param attributeName  name of attribute whose load status is
     *        to be determined
     * @return load status of the attribute
     */
    @Nonnull
    LoadState isLoadedWithoutReference(@Nonnull Object entity,
                                       @Nonnull String attributeName);

    /**
     * If the provider determines that the entity has been provided by
     * itself and that the state of the specified attribute has been loaded,
     * this method returns {@link LoadState#LOADED}.
     * <p> If a provider determines that the entity has been provided by
     * itself and that either the entity attributes with {@link FetchType#EAGER}
     * have not been loaded or that the state of the specified attribute has
     * not been loaded, this method returns {@link LoadState#NOT_LOADED}.
     * <p> If the provider cannot determine the load state, this method
     * returns {@link LoadState#UNKNOWN}.
     * <p> The provider's implementation of this method is permitted to
     * obtain a reference to the attribute value. (This access is safe
     * because providers which might trigger the loading of the attribute
     * state will have already been determined by
     * {@link #isLoadedWithoutReference}.)
     *
     * @param entity  entity instance
     * @param attributeName  name of attribute whose load status is
     *        to be determined
```

```
     * @return load status of the attribute
     */
    @Nonnull
    LoadState isLoadedWithReference(@Nonnull Object entity,
                                    @Nonnull String attributeName);

    /**
     * If the provider determines that the entity has been provided by
     * itself and that the state of all attributes for which
     * {@link FetchType#EAGER} has been specified have been loaded, this
     * method returns {@link LoadState#LOADED}.
     * <p> If the provider determines that the entity has been provided
     * by itself and that not all attributes with {@link FetchType#EAGER}
     * have been loaded, this method returns {@link LoadState#NOT_LOADED}.
     * <p> If the provider cannot determine if the entity has been
     * provided by itself, this method returns {@link LoadState#UNKNOWN}.
     * <p> The provider's implementation of this method must not obtain
     * a reference to any attribute value, as this could trigger the
     * loading of entity state if the entity has been provided by a
     * different provider.
     * @param entity whose loaded status is to be determined
     * @return load status of the entity
     */
    @Nonnull
    LoadState isLoaded(@Nonnull Object entity);
}
```

Appendix F: Static Query API

The following APIs are defined in the package `jakarta.persistence.query`.

F.1. `JakartaQuery`

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

/**
 * Specifies a query expressed in the Jakarta Persistence Query Language.
 * This annotation may be applied to:
 * <ul>
 * <li>any method of any class or interface belonging to the persistence
 *     unit, or
 * <li>a query method of a Jakarta Data repository with an implementation
 *     backed by Jakarta Persistence.
 * </ul>
 * <p> The return type of the method must agree with the type returned by
 * the query. The parameter types of the method must agree with the types
 * of the query parameters. An implementation of Jakarta Persistence or
 * of Jakarta Data may be able to determine if such agreement exists at
 * compilation time or when the persistence unit is initialized, but this
 * is not required.
 *
 * <p>
 * {@snippet :
 * interface Library {
 *
 *     @JakartaQuery("from Book where title like :title")
 *     @QueryOptions(cacheStoreMode = CacheStoreMode.BYPASS)
 *     List<Book> findBooksByTitle(String title);
 *
 *     @JakartaQuery("from Book where isbn = :isbn")
 *     Book getBookWithIsbn(String isbn);
 *
 *     record Summary(String title, String isbn, LocalDate date) {}
 *     @JakartaQuery("""
 *         select title, isbn, pubDate
 *         from Book
 *         where title like = ?1 and pubDate > ?2
 *     """)
 *     List<Summary> retrieveSummaries(String title, LocalDate fromDate);
 *
 *     @JakartaQuery("delete from Trash")
 *     @QueryOptions(timeout = 30_000)
 *     int emptyTrash();
 * }
 *}
 *
 * <p> A method return type <em>agrees</em> with the type returned by a
 * {@code SELECT} query if either:
 * <ul>
 * <li>it is exactly {@code R}, {@link java.util.List List&lt;R&gt;}, or
 *     {@link java.util.stream.Stream Stream&lt;R&gt;} for some result
 *     type {@code R} which is compatible with the query, according to the
 *     rules specified below, or
 * <li>the method is a Jakarta Data repository method, and its return type
 *     is a legal query method return type for the given query, as specified
 *     by Jakarta Data.
 * </ul>
```

```
* <p>A result type {@code R} is considered <em>compatible</em> with a query
* if either:
* <ol>
* <li>the select list of the query contains only a single item, and the type
*     of the item is assignable to the given result class, or
* <li>the result class is a non-abstract class or record type that has a
*     constructor with the same number of parameters as the query has items
*     in its select list, and the constructor parameter types exactly match
*     the types of the corresponding items in the select list.
* </ol>
* <p>In the first case, each query result is returned directly to the caller.
* In the second case, each query result is automatically packaged in a new
* instance of the result class by calling the matching constructor.
*
* <p> A method return type <em>agrees</em> with the type returned by an
* {@code UPDATE} or {@code DELETE} statement if the return type is
* {@code void}, {@code int}, or {@code long}.
*
* <p> A method parameter type <em>agrees</em> with a query parameter type if
* it is a type that could be assigned to the corresponding query parameter
* by passing an instance of the type to {@code setParameter()}. In making
* this determination, method parameters are correlated with query parameters
* by position or by name, depending on whether the query uses positional or
* named parameters.
*
* <p> The parameter types of the annotated method must be basic types.
* If this annotation occurs on a generic method, on a method with a
* parameter type which is not a basic type, or on a method with a
* return type or parameter type involving a type variable, the behavior
* is undefined.
*
* <p> Any implementation of Jakarta Data backed by Jakarta Persistence
* must support the use of this annotation as a Jakarta Data query
* annotation. Thus, a repository method annotated {@code @JakartaQuery}
* is treated by the Jakarta Data provider as a query method, and an
* implementation of the method is supplied by the Jakarta Data provider.
*
* <p> When this annotation is applied to a method of a class or interface
* belonging to the persistence unit, a reference to a query declared using
* this annotation may be obtained from the static metamodel class of the
* annotated class or interface.
* {@snippet :
* List<Book> books =
*         em.createQuery(Library_.findBooksByTitle("%Jakarta%"))
*           .getResultList();
* }
* {@snippet :
* Book book =
*         em.createQuery(Library_.getBookWithIsbn(isbn))
*           .getSingleResult();
* }
* {@snippet :
* int deleted =
*         em.createQuery(Library_.emptyTrash())
*           .executeUpdate();
* }
*
* <p> In addition, the query is treated as a named query, where the query
* name agrees with the convention for Javadoc links, that is: the
* concatenation of the fully qualified name of the type, with the string
* {@code #}, the name of the annotated member, and the raw types of the
* method parameters, enclosed in parentheses and separated by commas, for
* example, {@code "org.example.Library#findBooksByTitle(java.lang.String)"}.
*
* <p> The persistence provider must treat the annotation
* {@code jakarta.data.repository.Query} as a synonym for this annotation.
*
```

```
 * @since 4.0
 */
@Target(METHOD)
@Retention(RUNTIME)
public @interface JakartaQuery {
    /**
     * The query string in the Jakarta Persistence Query Language.
     */
    String value();
}
```

F.2. NativeQuery

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

/**
 * Specifies a query expressed in the native SQL dialect of the database.
 * This annotation may be applied to:
 * <ul>
 * <li>any method of any class or interface belonging to the persistence
 *     unit, or
 * <li>a query method of a Jakarta Data repository with
 *     an implementation backed by Jakarta Persistence.
 * </ul>
 * <p> The return type of the method must agree with the type returned by
 * the query. The parameter types of the method must agree with the types
 * of the query parameters. It is not usually possible to determine if
 * such agreement exists without executing the native SQL query.
 *
 * <p>
 * {@snippet :
 * interface Library {
 *
 *     @NativeQuery("select * from books where title like ?")
 *     @QueryOptions(cacheStoreMode = CacheStoreMode.BYPASS)
 *     List<Book> findBooksByTitle(String title);
 *
 *     @NativeQuery("select * from books where isbn = ?")
 *     Book getBookWithIsbn(String isbn);
 *
 *     @NativeQuery("delete from documents"
 *             + " where id in (select document_id from trash)")
 *     @QueryOptions(timeout = 30_000)
 *     int emptyTrash();
 * }
 *}
 * <p> A method return type <em>agrees</em> with the type returned by a
 * SQL statement that returns a result set if either:
 * <ul>
 * <li>it is exactly {@code R}, {@link java.util.List List&lt;R&gt;}, or
 *     {@link java.util.stream.Stream Stream&lt;R&gt;} where the query
 *     return type is assignable to {@code R}, or
 * <li>the method is a Jakarta Data repository method, and its return type
 *     is a legal query method return type for the given query, as specified
 *     by Jakarta Data.
 * </ul>
 *
 * <p> A method return type <em>agrees</em> with the type returned by a
 * SQL statement that returns a row count if the return type is
 * {@code void}, {@code int}, or {@code long}.
 *
```

```
* <p> A method parameter type agrees with a query parameter type if it is
* a type that could be assigned to the corresponding query parameter by
* passing an instance of the type to {@code setParameter()}. In making
* this determination, method parameters are correlated with query parameters
* by position or by name, depending on whether the query uses positional or
* named parameters.
*
* <p>The parameter types of the annotated method must be basic types.
* If this annotation occurs on a generic method, on a method with a
* parameter type which is not a basic type, or on a method with a
* return type or parameter type involving a type variable, the behavior
* is undefined.
*
* <p> Any implementation of Jakarta Data backed by Jakarta Persistence
* must support the use of this annotation as a Jakarta Data query
* annotation. Thus, a repository method annotated {@code @NativeQuery}
* is treated by the Jakarta Data provider as a query method, and an
* implementation of the method is supplied by the Jakarta Data provider.
*
* <p> When this annotation is applied to a method of a class or interface
* belonging to the persistence unit, a reference to a query declared using
* this annotation may be obtained from the static metamodel class of the
* annotated class or interface.
* {@snippet :
* List<Book> books =
*        em.createQuery(Library_.findBooksByTitle("%Jakarta%"))
*          .getResultList();
* }
* {@snippet :
* Book book =
*        em.createQuery(Library_.getBookWithIsbn(isbn))
*          .getSingleResult();
* }
* {@snippet :
* int deleted =
*        em.createQuery(Library_.emptyTrash())
*          .executeUpdate();
* }
*
* <p> In addition, the query is treated as a named query, where the query
* name agrees with the convention for Javadoc links, that is: the
* concatenation of the fully qualified name of the type, with the string
* {@code #}, the name of the annotated member, and the raw types of the
* method parameters, enclosed in parentheses and separated by commas, for
* example, {@code "org.example.Library#getBookWithIsbn(java.lang.String)"}.
*
* <p> A {@linkplain jakarta.persistence.SqlResultSetMapping result set
* mapping} may be specified by annotating the method with one or more
* {@link jakarta.persistence.EntityResult},
* {@link jakarta.persistence.ColumnResult}, or
* {@link jakarta.persistence.ConstructorResult} annotations.
* {@snippet :
* @NativeQuery("select * from books where isbn = ?")
* @EntityResult(entityClass = Book.class,
*     fields = {@FieldResult(name = Book_.ISBN, column = "isbn_13"),
*               @FieldResult(name = Book_.TITLE, column = "title_en")})
* Book getBookWithIsbn(String isbn);
* }
*
* <p>When more than one result mapping annotation is specified, each
* row of the mapped query result set is represented as an instance of
* {@code Object[]} whose elements are, in order:
* <ol>
* <li>{@linkplain jakarta.persistence.EntityResult entity results},
*     in the order in which they are declared;
* <li>{@linkplain jakarta.persistence.ConstructorResult constructor
*     results}, in the order in which they are declared; and then
```

```
 * <li>{@linkplain jakarta.persistence.ColumnResult scalar results},
 *     in the order in which they are declared.
 * </ol>
 *
 * @since 4.0
 */
@Target(METHOD)
@Retention(RUNTIME)
public @interface NativeQuery {
    /**
     * The query string in the native SQL dialect of the database.
     */
    String value();
}
```

F.3. QueryOptions

```
import jakarta.persistence.CacheRetrieveMode;
import jakarta.persistence.CacheStoreMode;
import jakarta.persistence.LockModeType;
import jakarta.persistence.PessimisticLockScope;
import jakarta.persistence.Query;
import jakarta.persistence.QueryFlushMode;
import jakarta.persistence.QueryHint;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

/**
 * <p>Configures options that affect the execution of a query.
 * This annotation may be applied to:
 * <ul>
 * <li>a method with a {@link JakartaQuery} or Jakarta Data
 *     {@code jakarta.data.repository.Query} annotation,
 * <li>a method with a {@link NativeQuery} annotation,
 * <li>a Jakarta Data repository method annotated
 *     {@code jakarta.data.repository.Delete} that uses the
 *     parameter-based automatic query pattern,
 *     or
 * <li>a Jakarta Data repository method annotated
 *     {@code jakarta.data.repository.Find}.
 * </ul>
 *
 * <p>This annotation must be respected by an implementation
 * of Jakarta Data backed by Jakarta Persistence.
 *
 * @since 4.0
 */
@Target(METHOD)
@Retention(RUNTIME)
public @interface QueryOptions {
    /**
     * The {@linkplain CacheStoreMode cache store mode} to use.
     * The presence of this annotation overrides the default
     * cache store mode of the persistence context.
     * <p>
     * This option applies when the query is a JPQL {@code select}
     * query or a native query that returns a result set.
     *
     * @see jakarta.persistence.TypedQuery#setCacheStoreMode
     */
    CacheStoreMode cacheStoreMode() default CacheStoreMode.USE;
```

```java
/**
 * The {@linkplain CacheRetrieveMode cache retrieve mode}
 * to use. The presence of this annotation overrides the
 * default cache retrieve mode of the persistence context.
 * <p>
 * This option applies when the query is a JPQL {@code select}
 * query or a native query that returns a result set.
 *
 * @see jakarta.persistence.TypedQuery#setCacheRetrieveMode
 */
CacheRetrieveMode cacheRetrieveMode() default CacheRetrieveMode.USE;

/**
 * A query timeout in milliseconds.
 * By default, there is no timeout.
 * @see jakarta.persistence.Query#setTimeout
 */
int timeout() default -1;

/**
 * Query properties and hints.
 * May include vendor-specific query hints.
 * @see jakarta.persistence.Query#setHint(String, Object)
 */
QueryHint[] hints() default {};

/**
 * The lock mode type to use in query execution.
 * If a {@code lockMode} other than {@link LockModeType#NONE}
 * is specified, the query must be executed in a transaction
 * and the persistence context joined to the transaction.
 * <p> If a lock mode is explicitly specified for a
 * {@linkplain NativeQuery native query}, the behavior
 * is undefined and unportable between persistence providers.
 * <p>
 * This option applies when the query is a JPQL {@code select}
 * query or a native query that returns a result set.
 *
 * @see jakarta.persistence.TypedQuery#setLockMode
 */
LockModeType lockMode() default LockModeType.NONE;

/**
 * The pessimistic lock scope to use in query execution if a
 * pessimistic lock mode is specified via {@link #lockMode}.
 * <p>
 * This option applies when the query is a JPQL {@code select}
 * query or a native query that returns a result set.
 */
PessimisticLockScope lockScope() default PessimisticLockScope.NORMAL;

/**
 * The name of a {@link jakarta.persistence.NamedEntityGraph}
 * applied to the entity returned by the query. By default,
 * no entity graph is applied. The named entity graph must be
 * an entity graph compatible with the entity type returned
 * by the query. If the result type of the query is not an
 * entity type, the behavior is undefined. The entity graph
 * is interpreted as a load graph. The entity graph specified
 * here may be overridden by calling {@code setEntityGraph()}.
 * <p>
 * This option applies when the query is a JPQL {@code select}.
 *
 *
 * @see jakarta.persistence.NamedEntityGraph#name
 */
```

```
    String entityGraph() default "";

    /**
     * The {@linkplain QueryFlushMode query flush mode} to use when
     * executing the query.
     * @see Query#setQueryFlushMode(QueryFlushMode)
     */
    QueryFlushMode flush() default QueryFlushMode.DEFAULT;
}
```

F.4. StaticStatementReference

```
import jakarta.persistence.Statement;
import jakarta.persistence.StatementReference;
import jakarta.persistence.Timeout;
import jakarta.persistence.TypedQuery;

import java.util.*;

import static java.util.Collections.emptyMap;

/**
 * A reference to a statement declared using a {@link JakartaQuery}
 * or {@link NativeQuery} annotation. An instance of this
 * class is usually instantiated by a generated method of the
 * {@linkplain jakarta.persistence.metamodel.StaticMetamodel
 * static metamodel} for the annotated program element.
 *
 * @since 4.0
 */
public class StaticStatementReference
        implements StatementReference {

    private final Class<?> annotatedClass;
    private final String annotatedMemberName;
    private final String name;
    private final List<Class<?>> parameterTypes;
    private final List<String> parameterNames;
    private final List<Object> arguments;
    private final Map<String, Object> hints;
    private final Set<Statement.Option> options;

    /**
     * Intended for use in code generated by an annotation processor.
     * This constructor overload is adapted to the case where the
     * program element is annotated with {@link QueryOptions}.
     *
     * @param queryName The name of the query, the concatenation of
     *                  the unqualified name of the annotated type,
     *                  with the string {@code "."}, and the name of
     *                  the annotated member
     * @param annotatedClass The annotated class
     * @param annotatedMemberName The name of the annotated member
     * @param parameterTypes The types of the parameters of the
     *                       annotated member
     * @param parameterNames The names of the parameters of the
     *                       annotated member
     * @param arguments The arguments supplied to the parameters
     *                  at runtime
     * @param hints See {@link QueryOptions#hints}
     * @param options All specified {@linkplain Statement.Option options}
     */
    public StaticStatementReference(
            String queryName,
            Class<?> annotatedClass,
```

```
        String annotatedMemberName,
        List<Class<?>> parameterTypes,
        List<String> parameterNames,
        List<Object> arguments,
        Map<String, Object> hints,
        Statement.Option... options) {
    this.name = queryName;
    this.annotatedClass = annotatedClass;
    this.annotatedMemberName = annotatedMemberName;
    this.parameterTypes = parameterTypes;
    this.parameterNames = parameterNames;
    this.arguments = arguments;
    this.hints = hints;
    this.options = Set.of(options);
}

/**
 * Intended for use in code generated by an annotation processor.
 * This constructor overload is adapted to the case where the
 * program element is not annotated with {@link QueryOptions}.
 *
 * @param queryName The name of the query, the concatenation of
 *                  the unqualified name of the annotated type,
 *                  with the string {@code "."}, and the name of
 *                  the annotated member
 * @param annotatedClass The annotated class
 * @param annotatedMemberName The name of the annotated member
 * @param parameterTypes The types of the parameters of the
 *                       annotated member
 * @param parameterNames The names of the parameters of the
 *                       annotated member
 * @param arguments The arguments supplied to the parameters
 *                  at runtime
 */
public StaticStatementReference(
        String queryName,
        Class<?> annotatedClass,
        String annotatedMemberName,
        List<Class<?>> parameterTypes,
        List<String> parameterNames,
        List<Object> arguments) {
    this(queryName, annotatedClass, annotatedMemberName,
            parameterTypes, parameterNames, arguments,
            emptyMap());
}

@Override
public String getName() {
    return name;
}

@Override
public Map<String, Object> getHints() {
    return hints;
}

@Override
public List<Class<?>> getParameterTypes() {
    return parameterTypes;
}

@Override
public List<String> getParameterNames() {
    return parameterNames;
}

@Override
```

```java
    public List<Object> getArguments() {
        return arguments;
    }

    public Class<?> getAnnotatedClass() {
        return annotatedClass;
    }

    public String getAnnotatedMemberName() {
        return annotatedMemberName;
    }

    @Override
    public Set<Statement.Option> getOptions() {
        return new HashSet<>(options);
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        else if ((!(obj instanceof StaticStatementReference that))) {
            return false;
        }
        else {
            return Objects.equals(this.name, that.name)
                && Objects.equals(this.parameterTypes, that.parameterTypes)
                && Objects.equals(this.parameterNames, that.parameterNames)
                && Objects.equals(this.arguments, that.arguments)
                && Objects.equals(this.options, that.options)
                && Objects.equals(this.hints, that.hints);
        }
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, parameterTypes, parameterNames, arguments);
    }

    @Override
    public String toString() {
        return "StaticStatementReference["
                + "name=" + name + ", "
                + "parameterNames=" + parameterNames + ", "
                + "arguments=" + arguments + ", "
                + "options=" + options + ", "
                + "hints=" + hints
                + ']';
    }
}
```

F.5. StaticTypedQueryReference

```java
import jakarta.persistence.*;

import java.util.*;

import static java.util.Collections.emptyMap;

/**
 * A reference to a query declared using a {@link JakartaQuery}
 * or {@link NativeQuery} annotation. An instance of this
 * class is usually instantiated by a generated method of the
 * {@linkplain jakarta.persistence.metamodel.StaticMetamodel
```

```
 * static metamodel} for the annotated program element.
 *
 * @param <R> The result type of the query
 *
 * @since 4.0
 */
public class StaticTypedQueryReference<R>
        implements TypedQueryReference<R> {

    private final Class<?> annotatedClass;
    private final String annotatedMemberName;
    private final Class<R> resultType;
    private final String name;
    private final List<Class<?>> parameterTypes;
    private final List<String> parameterNames;
    private final List<Object> arguments;
    private final String entityGraphName;
    private final Map<String, Object> hints;
    private final Set<TypedQuery.Option> options;

    /**
     * Intended for use in code generated by an annotation processor.
     * This constructor overload is adapted to the case where the
     * program element is annotated with {@link QueryOptions}.
     *
     * @param queryName The name of the query, the concatenation of
     *                  the unqualified name of the annotated type,
     *                  with the string {@code "."}, and the name of
     *                  the annotated member
     * @param annotatedClass The annotated class
     * @param annotatedMemberName The name of the annotated member
     * @param resultType The result type of the query, the type of
     *                   the annotated member
     * @param parameterTypes The types of the parameters of the
     *                       annotated member
     * @param parameterNames The names of the parameters of the
     *                       annotated member
     * @param arguments The arguments supplied to the parameters
     *                  at runtime
     * @param entityGraphName See {@link QueryOptions#entityGraph}
     * @param options All specified {@linkplain TypedQuery.Option options}
     * @param hints See {@link QueryOptions#hints}
     */
    public StaticTypedQueryReference(
            String queryName,
            Class<?> annotatedClass,
            String annotatedMemberName,
            Class<R> resultType,
            List<Class<?>> parameterTypes,
            List<String> parameterNames,
            List<Object> arguments,
            String entityGraphName,
            Map<String, Object> hints,
            TypedQuery.Option... options) {
        this.name = queryName;
        this.annotatedClass = annotatedClass;
        this.annotatedMemberName = annotatedMemberName;
        this.resultType = resultType;
        this.parameterTypes = parameterTypes;
        this.parameterNames = parameterNames;
        this.arguments = arguments;
        this.entityGraphName = entityGraphName;
        this.hints = hints;
        this.options = Set.of(options);
    }

    /**
```

```
 * Intended for use in code generated by an annotation processor.
 * This constructor overload is adapted to the case where the
 * program element is not annotated with {@link QueryOptions}.
 *
 * @param queryName The name of the query, the concatenation of
 *                  the unqualified name of the annotated type,
 *                  with the string {@code "."}, and the name of
 *                  the annotated member
 * @param annotatedClass The annotated class
 * @param annotatedMemberName The name of the annotated member
 * @param resultType The result type of the query, the type of
 *                   the annotated member
 * @param parameterTypes The types of the parameters of the
 *                       annotated member
 * @param parameterNames The names of the parameters of the
 *                       annotated member
 * @param arguments The arguments supplied to the parameters
 *                  at runtime
 */
public StaticTypedQueryReference(
        String queryName,
        Class<?> annotatedClass,
        String annotatedMemberName,
        Class<R> resultType,
        List<Class<?>> parameterTypes,
        List<String> parameterNames,
        List<Object> arguments) {
    this(queryName, annotatedClass, annotatedMemberName, resultType,
            parameterTypes, parameterNames, arguments,
            null, emptyMap());
}

@Override
public String getName() {
    return name;
}

@Override
public Class<? extends R> getResultType() {
    return resultType;
}

@Override
public Map<String, Object> getHints() {
    return hints;
}

@Override
public List<Class<?>> getParameterTypes() {
    return parameterTypes;
}

@Override
public List<String> getParameterNames() {
    return parameterNames;
}

@Override
public List<Object> getArguments() {
    return arguments;
}

public Class<?> getAnnotatedClass() {
    return annotatedClass;
}

public String getAnnotatedMemberName() {
```

```
        return annotatedMemberName;
    }

    @Override
    public String getEntityGraphName() {
        return entityGraphName;
    }

    @Override
    public Set<TypedQuery.Option> getOptions() {
        return new HashSet<>(options);
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        else if ((!(obj instanceof StaticTypedQueryReference<?> that))) {
            return false;
        }
        else {
            return Objects.equals(this.resultType, that.resultType)
                && Objects.equals(this.name, that.name)
                && Objects.equals(this.parameterTypes, that.parameterTypes)
                && Objects.equals(this.parameterNames, that.parameterNames)
                && Objects.equals(this.arguments, that.arguments)
                && Objects.equals(this.options, that.options)
                && Objects.equals(this.hints, that.hints);
        }
    }

    @Override
    public int hashCode() {
        return Objects.hash(resultType, name, parameterTypes, parameterNames, arguments);
    }

    @Override
    public String toString() {
        return "StaticTypedQueryReference["
                + "resultType=" + resultType + ", "
                + "name=" + name + ", "
                + "parameterNames=" + parameterNames + ", "
                + "arguments=" + arguments + ", "
                + "options=" + options
                + "hints=" + hints
                + ']';
    }
}
```

Appendix G: Result Set Mapping API

The following APIs are defined in the package `jakarta.persistence.sql`.

G.1. `ResultSetMapping`

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.SingularAttribute;

/**
 * Specifies a mapping of the columns of a result set of a SQL query or stored procedure
 * to {@linkplain EntityMapping entities}, {@linkplain ColumnMapping scalar values}, and
 * {@linkplain ConstructorMapping Java class constructors}.
 *
 * <p>A {@link ResultSetMapping} may be instantiated programmatically, for example:
 * {@snippet :
 * import static jakarta.persistence.sql.ResultSetMapping.*;
 *
 * ...
 *
 * var entityMapping =
 *         entity(Author.class,
 *                 field(Author_.ssn, "auth_ssn"),
 *                 embedded(Author_.name,
 *                         field(Name_.first, "auth_first_name"),
 *                         field(Name_.last, "auth_last_name")));
 *
 * var constructorMapping =
 *         constructor(Summary.class,
 *                 column("isbn", String.class),
 *                 column("title", String.class),
 *                 column("author", String.class));
 *
 * var compoundMapping =
 *         compound(
 *                 entity(Author.class),
 *                 entity(Book.class, field(Book_.isbn, "isbn")),
 *                 column("sales", BigDecimal.class),
 *                 constructor(Summary.class, column("isbn"), column("title"))
 *         );
 *}
 *
 * <p>Alternatively, an instance representing a
 * {@linkplain jakarta.persistence.SqlResultSetMapping result set mapping defined using annotations}
 * may be obtained via {@link jakarta.persistence.EntityManagerFactory#getResultSetMappings}.
 *
 * <p>A {@code ResultSetMapping} may be used to
 * {@linkplain jakarta.persistence.EntityHandler#createNativeQuery(String, ResultSetMapping) obtain}
 * and execute a {@link jakarta.persistence.TypedQuery TypedQuery}.
 *
 * @see jakarta.persistence.SqlResultSetMapping
 * @see jakarta.persistence.EntityManagerFactory#getResultSetMappings(Class)
 * @see jakarta.persistence.EntityHandler#createNativeQuery(String, ResultSetMapping)
 * @see jakarta.persistence.StoredProcedureQuery#getResultList(ResultSetMapping)
 * @see jakarta.persistence.StoredProcedureQuery#getSingleResult(ResultSetMapping)
 * @see jakarta.persistence.StoredProcedureQuery#getSingleResultOrNull(ResultSetMapping)
 *
 * @param <T> The result type of the mapping
 *
 * @since 4.0
 */
public sealed interface ResultSetMapping<T>
        permits CompoundMapping, TupleMapping, EntityMapping, ConstructorMapping, ColumnMapping {
    /**
```

```
 * The result type of the mapping.
 */
@Nonnull
Class<T> type();

/**
 * Construct a mapping for a single column to a scalar value.
 *
 * @param columnName The colum name
 * @param type The Java type of the scalar value
 * @param <T> The type of the scalar value
 *
 * @see jakarta.persistence.ColumnResult
 */
@Nonnull
static <T> ColumnMapping<T> column(@Nonnull String columnName,
                                   @Nonnull Class<T> type) {
    return ColumnMapping.of(columnName, type);
}

/**
 * Construct a mapping for a single column to a scalar value.
 *
 * @param columnName The colum name
 *
 * @see jakarta.persistence.ColumnResult
 */
@Nonnull
static ColumnMapping<Object> column(@Nonnull String columnName) {
    return ColumnMapping.of(columnName);
}

/**
 * Construct a mapping to a constructor of a Java class.
 *
 * @param targetClass The Java class which declares the constructor
 * @param arguments Mappings for the constructor parameters, in order
 * @param <T> The type of the Java class
 *
 * @see jakarta.persistence.ConstructorResult
 */
@Nonnull
static <T> ConstructorMapping<T> constructor(@Nonnull Class<T> targetClass,
                                             @Nonnull MappingElement<?>... arguments) {
    return ConstructorMapping.of(targetClass, arguments);
}

/**
 * Construct a mapping which packages a tuple of values as a Java array.
 *
 * @param elements Mappings for elements of the type
 */
@Nonnull
static CompoundMapping compound(@Nonnull MappingElement<?>... elements) {
    return CompoundMapping.of(elements);
}

/**
 * Construct a mapping which packages a tuple of values as an instance of
 * {@link jakarta.persistence.Tuple}.
 *
 * @param elements Mappings for elements of the type
 */
@Nonnull
static TupleMapping tuple(@Nonnull MappingElement<?>... elements) {
    return TupleMapping.of(elements);
}
```

```java
/**
 * Construct a mapping for an entity class.
 *
 * @param entityClass The Java class of the entity
 * @param fields Mappings for fields or properties of the entity
 * @param <T> The entity type
 *
 * @see jakarta.persistence.EntityResult
 */
@Nonnull
@SafeVarargs
static <T> EntityMapping<T> entity(@Nonnull Class<T> entityClass,
                                   @Nonnull MemberMapping<T>... fields) {
    return EntityMapping.of(entityClass, fields);
}

/**
 * Construct a mapping for an entity class.
 *
 * @param entityClass The Java class of the entity
 * @param discriminatorColumn The name of the column holding the discriminator;
 *        an empty string indicates that there is no discriminator column
 * @param fields Mappings for fields or properties of the entity
 *               and of its entity subclasses
 * @param <T> The entity type
 *
 * @see jakarta.persistence.EntityResult
 */
@Nonnull
@SafeVarargs
static <T> EntityMapping<T> entity(@Nonnull Class<T> entityClass,
                                   @Nonnull String discriminatorColumn,
                                   @Nonnull MemberMapping<? extends T>... fields) {
    return EntityMapping.of(entityClass, discriminatorColumn, fields);
}

/**
 * Construct a mapping for an embedded object.
 *
 * @param container The Java class which declares the field holding the embedded object
 * @param embeddableClass The Java class of the embedded object
 * @param name The name of the field holding the embedded object
 * @param fields Mappings for fields or properties of the entity
 * @param <C> The container type
 * @param <T> The embeddable type
 */
@Nonnull
@SafeVarargs
static <C,T> EmbeddedMapping<C,T> embedded(@Nonnull Class<? super C> container,
                                           @Nonnull Class<T> embeddableClass,
                                           @Nonnull String name, MemberMapping<T>... fields) {
    return EmbeddedMapping.of(container, embeddableClass, name, fields);
}

/**
 * Construct a mapping for an embedded object.
 *
 * @param embedded The metamodel attribute representing the field or property holding the embedded object
 * @param fields Mappings for fields or properties of the entity
 * @param <C> The container type
 * @param <T> The embeddable type
 */
@Nonnull
@SafeVarargs
static <C,T> EmbeddedMapping<C,T> embedded(@Nonnull SingularAttribute<? super C,T> embedded,
                                           @Nonnull MemberMapping<T>... fields) {
```

```
        return EmbeddedMapping.of(embedded, fields);
    }

    /**
     * Construct a mapping for a field or property of an entity or embeddable type.
     *
     * @param container The Java class which declares the field or property
     * @param type The type of the field or property
     * @param name The name of the field or property
     * @param columnName The name of the mapped column
     * @param <C> The type of the entity or embeddable type
     * @param <T> The type of the field or property
     *
     * @see jakarta.persistence.FieldResult
     */
    @Nonnull
    static <C,T> FieldMapping<C,T> field(@Nonnull Class<? super C> container,
                                         @Nonnull Class<T> type,
                                         @Nonnull String name,
                                         @Nonnull String columnName) {
        return FieldMapping.of(container, type, name, columnName);
    }

    /**
     * Construct a mapping for a field or property of an entity or embeddable type.
     *
     * @param attribute The metamodel attribute representing the field or property
     * @param columnName The name of the mapped column
     * @param <C> The type of the entity or embeddable type
     * @param <T> The type of the field or property
     *
     * @see jakarta.persistence.FieldResult
     */
    static <C,T> FieldMapping<C,T> field(@Nonnull SingularAttribute<? super C,T> attribute,
                                         @Nonnull String columnName) {
        return FieldMapping.of(attribute, columnName);
    }
}
```

G.2. MappingElement

```
import jakarta.annotation.Nonnull;
import jakarta.persistence.TupleElement;

/**
 * Supertype of objects which map a column or columns of a
 * JDBC {@link java.sql.ResultSet} to a Java type returned
 * by the query. In particular, a {@link CompoundMapping}
 * or {@link TupleMapping} packages a list of
 * {@code MappingElement}s.
 *
 * @param <T> The type returned
 *
 * @since 4.0
 */
public sealed interface MappingElement<T> extends TupleElement<T>
        permits EntityMapping, ColumnMapping, ConstructorMapping  {
    /**
     * Specify an alias for this element of the result set.
     * @param alias The alias
     */
    @Nonnull
    MappingElement<T> withAlias(@Nonnull String alias);
}
```

G.3. MemberMapping

```
/**
 * Supertype of objects which map a column or columns of
 * a JDBC {@link java.sql.ResultSet} to a member of an
 * entity or embeddable type. A {@link EntityMapping} or
 * {@link EmbeddedMapping} packages a list of
 * {@code MemberMapping}s.
 *
 * @param <T> The entity or embeddable type
 *
 * @since 4.0
 */
public sealed interface MemberMapping<T>
        permits FieldMapping, EmbeddedMapping {
}
```

G.4. ColumnMapping

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

import static java.util.Objects.requireNonNull;

/**
 * Maps a column of a JDBC {@link java.sql.ResultSet} to a scalar
 * value in the result returned by the query.
 *
 * @param columnName The name of the mapped column of the result set
 * @param type The Java type of the resulting scalar value
 * @param alias An optional alias
 * @param <T> The type of the resulting scalar value
 *
 * @see jakarta.persistence.ColumnResult
 *
 * @since 4.0
 */
public record ColumnMapping<T>(@Nonnull String columnName,
                               @Nonnull Class<T> type,
                               @Nullable String alias)
        implements MappingElement<T>, ResultSetMapping<T> {

    public ColumnMapping {
        requireNonNull(columnName, "columnName is required");
        requireNonNull(type, "type is required");
    }

    /**
     * The Java type of the scalar value.
     */
    @Override
    @Nonnull
    public Class<T> getJavaType() {
        return type;
    }

    /**
     * The colum name.
     */
    @Override
    @Nullable
    public String getAlias() {
        return alias;
    }
```

```java
    /**
     * Specify an alias for this column in the result set.
     * @param alias The alias
     */
    @Override
    @Nonnull
    public ColumnMapping<T> withAlias(@Nonnull String alias) {
        return new ColumnMapping<>(columnName, type, alias);
    }

    /**
     * Construct a new instance.
     * @param columnName The name of the mapped column of the result set
     */
    @Nonnull
    public static ColumnMapping<Object> of(@Nonnull String columnName) {
        return new ColumnMapping<>(columnName, Object.class, columnName);
    }

    /**
     * Construct a new instance.
     * @param columnName The name of the mapped column of the result set
     * @param type The Java type of the resulting scalar value
     * @param <T> The type of the resulting scalar value
     */
    @Nonnull
    public static <T> ColumnMapping<T> of(@Nonnull String columnName,
                                          @Nonnull Class<T> type) {
        return new ColumnMapping<>(columnName, type, columnName);
    }
}
```

G.5. FieldMapping

```java
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.SingularAttribute;

import static java.util.Objects.requireNonNull;

/**
 * Maps a column of a JDBC {@link java.sql.ResultSet} to a given
 * field or column of an entity or embeddable type.
 *
 * @param container The Java class which declares the field
 * @param type The Java class of the field
 * @param name The name of the field
 * @param columnName The name of the mapped column of the result set
 * @param <C> The type of the entity or embeddable type
 * @param <T> The type of the field
 *
 * @see jakarta.persistence.FieldResult
 *
 * @since 4.0
 */
public record FieldMapping<C,T>(@Nonnull Class<? super C> container,
                                @Nonnull Class<T> type,
                                @Nonnull String name,
                                @Nonnull String columnName)
        implements MemberMapping<C> {

    public FieldMapping {
        requireNonNull(container, "container is required");
        requireNonNull(type, "type is required");
        requireNonNull(name, "name is required");
```

```
        requireNonNull(columnName, "columnName is required");
    }

    /**
     * Construct a new instance.
     * @param container The Java class which declares the field
     * @param type The Java class of the field
     * @param name The name of the field
     * @param columnName The name of the mapped column of the result set
     * @param <C> The type of the entity or embeddable type
     * @param <T> The type of the field
     */
    @Nonnull
    public static <C,T> FieldMapping<C,T> of(@Nonnull Class<? super C> container,
                                             @Nonnull Class<T> type,
                                             @Nonnull String name,
                                             @Nonnull String columnName) {
        return new FieldMapping<>(container, type, name, columnName);
    }

    /**
     * Construct a new instance.
     * @param attribute The metamodel object representing the field
     * @param columnName The name of the mapped column of the result set
     * @param <C> The type of the entity or embeddable type
     * @param <T> The type of the field
     */
    @Nonnull
    public static <C,T> FieldMapping<C,T> of(@Nonnull SingularAttribute<? super C,T> attribute,
                                             @Nonnull String columnName) {
        return new FieldMapping<>(attribute.getDeclaringType().getJavaType(),
                attribute.getJavaType(), attribute.getName(), columnName);
    }
}
```

G.6. EntityMapping

```
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;
import jakarta.persistence.LockModeType;

import static java.util.Objects.requireNonNull;

/**
 * Maps columns of a JDBC {@link java.sql.ResultSet} to a given
 * {@linkplain jakarta.persistence.Entity entity} class.
 *
 * @param entityClass The entity class
 * @param lockMode The lock mode acquired by the SQL query
 * @param discriminatorColumn The name of the column holding the
 *        {@linkplain jakarta.persistence.DiscriminatorColumn
 *        discriminator}; a {@code null} value indicates that
 *        there is no discriminator column.
 * @param fields Mappings for fields or properties of the entity
 *               and of its entity subclasses
 * @param alias An optional alias
 * @param <T> The entity type
 *
 * @see jakarta.persistence.EntityResult
 *
 * @since 4.0
 */
public record EntityMapping<T>(@Nonnull Class<T> entityClass,
                               @Nonnull LockModeType lockMode,
                               @Nullable String discriminatorColumn,
```

```
                            @Nonnull MemberMapping<? extends T>[] fields,
                            @Nullable String alias)
    implements MappingElement<T>, ResultSetMapping<T> {

public EntityMapping(@Nonnull Class<T> entityClass,
                     @Nonnull LockModeType lockMode,
                     @Nullable String discriminatorColumn,
                     @Nonnull MemberMapping<? extends T>[] fields,
                     @Nullable String alias) {
    requireNonNull(entityClass, "entityClass is required");
    requireNonNull(lockMode, "lockMode is required");
    if (discriminatorColumn != null && discriminatorColumn.isBlank()) {
        throw new IllegalArgumentException("discriminatorColumn may not be blank");
    }
    requireNonNull(fields, "fields are required");
    for (var field : fields) {
        requireNonNull(field, "field is required");
    }
    this.entityClass = entityClass;
    this.lockMode = lockMode;
    this.discriminatorColumn = discriminatorColumn;
    this.fields = fields.clone();
    this.alias = alias;
}

@Override
@Nonnull
public MemberMapping<? extends T>[] fields() {
    return fields.clone();
}

/**
 * The entity class.
 */
@Override
@Nonnull
public Class<T> getJavaType() {
    return entityClass;
}

/**
 * Return the alias specified via {@link #withAlias},
 * which may be used to retrieve an entity instance using
 * {@link jakarta.persistence.Tuple#get(String, Class)}
 * @return the explicitly specified alias or {@code null}
 */
@Override
@Nullable
public String getAlias() {
    return alias;
}

/**
 * Specify an alias for this entity in the result set.
 * @param alias The alias
 */
@Override
@Nonnull
public EntityMapping<T> withAlias(@Nonnull String alias) {
    return new EntityMapping<>(entityClass, lockMode, discriminatorColumn, fields, alias);
}

/**
 * Specify the lock mode obtained on this entity.
 * @param lockMode The lock mode
 */
@Nonnull
```

```java
    public EntityMapping<T> withLockMode(@Nonnull LockModeType lockMode) {
        return new EntityMapping<>(entityClass, lockMode, discriminatorColumn, fields, alias);
    }

    /**
     * Construct a new instance.
     * @param entityClass The entity class
     * @param fields Mappings for fields or properties of the entity
     * @param <T> The entity type
     */
    @Nonnull
    @SafeVarargs
    public static <T> EntityMapping<T> of(@Nonnull Class<T> entityClass,
                                          @Nonnull MemberMapping<T>... fields) {
        return new EntityMapping<>(entityClass, LockModeType.NONE, null, fields, null);
    }

    /**
     * Construct a new instance.
     * @param entityClass The entity class
     * @param discriminatorColumn The name of the column holding the
     *        {@linkplain jakarta.persistence.DiscriminatorColumn
     *        discriminator}; a {@code null} value indicates that
     *        there is no discriminator column.
     * @param fields Mappings for fields or properties of the entity
     *               and of its entity subclasses
     * @param <T> The entity type
     */
    @Nonnull
    @SafeVarargs
    public static <T> EntityMapping<T> of(@Nonnull Class<T> entityClass,
                                          @Nonnull String discriminatorColumn,
                                          @Nonnull MemberMapping<? extends T>... fields) {
        return new EntityMapping<>(entityClass, LockModeType.NONE, discriminatorColumn, fields, null);
    }

    /**
     * The entity class.
     */
    @Override
    @Nonnull
    public Class<T> type() {
        return entityClass;
    }
}
```

G.7. EmbeddedMapping

```java
import jakarta.annotation.Nonnull;
import jakarta.persistence.metamodel.SingularAttribute;

import static java.util.Objects.requireNonNull;

/**
 * Maps columns of a JDBC {@link java.sql.ResultSet} to a given
 * {@linkplain jakarta.persistence.Embedded embedded object}.
 *
 * @param container The Java class which declares the field
 *                  holding the embedded object
 * @param embeddableClass The embeddable class
 * @param name The name of the field holding the embedded object
 * @param fields Mappings for fields or properties of the entity
 * @param <T> The embeddable type
 * @param <C> The container type
 *
```

```java
 * @since 4.0
 */
public record EmbeddedMapping<C,T>(@Nonnull Class<? super C> container,
                                   @Nonnull Class<T> embeddableClass,
                                   @Nonnull String name,
                                   @Nonnull MemberMapping<T>[] fields)
        implements MemberMapping<C> {

    public EmbeddedMapping(@Nonnull Class<? super C> container,
                           @Nonnull Class<T> embeddableClass,
                           @Nonnull String name,
                           @Nonnull MemberMapping<T>[] fields) {
        requireNonNull(container, "container is required");
        requireNonNull(embeddableClass, "embeddableClass is required");
        requireNonNull(name, "name is required");
        requireNonNull(fields, "fields are required");
        for (var field : fields) {
            requireNonNull(field, "field is required");
        }
        this.container = container;
        this.embeddableClass = embeddableClass;
        this.name = name;
        this.fields = fields.clone();
    }

    @Override
    @Nonnull
    public MemberMapping<T>[] fields() {
        return fields.clone();
    }

    /**
     * Construct a new instance.
     * @param container The Java class which declares the field
     *                  holding the embedded object
     * @param embeddableClass The embeddable class
     * @param name The name of the field holding the embedded object
     * @param fields Mappings for fields or properties of the entity
     * @param <T> The embeddable type
     * @param <C> The container type
     */
    @SafeVarargs
    @Nonnull
    public static <C,T> EmbeddedMapping<C,T> of(@Nonnull Class<? super C> container,
                                                @Nonnull Class<T> embeddableClass,
                                                @Nonnull String name,
                                                @Nonnull MemberMapping<T>... fields) {
        return new EmbeddedMapping<>(container, embeddableClass, name, fields);
    }

    /**
     * Construct a new instance.
     * @param embedded The metamodel object representing the embedded object
     * @param fields Mappings for fields or properties of the entity
     * @param <T> The embeddable type
     * @param <C> The container type
     */
    @SafeVarargs
    public static <C,T> EmbeddedMapping<C,T> of(@Nonnull SingularAttribute<? super C,T> embedded,
                                                @Nonnull MemberMapping<T>... fields) {
        return new EmbeddedMapping<>(embedded.getDeclaringType().getJavaType(),
                embedded.getJavaType(), embedded.getName(), fields);
    }
}
```

G.8. CompoundMapping

```java
import jakarta.annotation.Nonnull;

import static java.util.Objects.requireNonNull;

/**
 * Maps a JDBC {@link java.sql.ResultSet} to a tuple of values
 * packaged as an object array.
 *
 * @param elements Mappings for the elements of the tuple
 *
 * @see TupleMapping
 * @see jakarta.persistence.SqlResultSetMapping
 *
 * @since 4.0
 */
public record CompoundMapping(@Nonnull MappingElement<?>[] elements)
        implements ResultSetMapping<Object[]> {

    public CompoundMapping(@Nonnull MappingElement<?>[] elements) {
        requireNonNull(elements, "elements are required");
        if (elements.length == 0) {
            throw new IllegalArgumentException("at least one element is required");
        }
        for (var element : elements) {
            requireNonNull(element, "element is required");
        }
        this.elements = elements.clone();
    }

    @Override
    @Nonnull
    public MappingElement<?>[] elements() {
        return elements.clone();
    }

    /**
     * Construct a new instance.
     * @param elements Mappings for the elements of the tuple
     */
    @Nonnull
    public static CompoundMapping of(@Nonnull MappingElement<?>... elements) {
        return new CompoundMapping(elements);
    }

    /**
     * Always returns {@code Object[].class}.
     */
    @Override
    @Nonnull
    public Class<Object[]> type() {
        return Object[].class;
    }
}
```

G.9. ConstructorMapping

```java
import jakarta.annotation.Nonnull;
import jakarta.annotation.Nullable;

import static java.util.Objects.requireNonNull;

/**
```

```
 * Maps columns of a JDBC {@link java.sql.ResultSet} to parameters
 * of the constructor of a Java class.
 *
 * @param targetClass The Java class which declares the constructor
 * @param arguments Mappings assigned to the parameters of the constructor
 * @param alias An optional alias
 * @param <T> The type of the Java class
 *
 * @see jakarta.persistence.ConstructorResult
 *
 * @since 4.0
 */

public record ConstructorMapping<T>(@Nonnull Class<T> targetClass,
                                    @Nonnull MappingElement<?>[] arguments,
                                    @Nullable String alias)
        implements MappingElement<T>, ResultSetMapping<T> {

    public ConstructorMapping(@Nonnull Class<T> targetClass,
                              @Nonnull MappingElement<?>[] arguments,
                              @Nullable String alias) {
        requireNonNull(targetClass, "targetClass is required");
        requireNonNull(arguments, "arguments are required");
        if (arguments.length == 0) {
            throw new IllegalArgumentException("at least one argument is required");
        }
        for (var element : arguments) {
            requireNonNull(element, "argument is required");
        }
        this.targetClass = targetClass;
        this.arguments = arguments.clone();
        this.alias = alias;
    }

    @Override
    @Nonnull
    public MappingElement<?>[] arguments() {
        return arguments.clone();
    }

    /**
     * The Java class which declares the constructor.
     */
    @Override
    @Nonnull
    public Class<T> getJavaType() {
        return targetClass;
    }

    /**
     * Return the alias specified via {@link #withAlias},
     * which may be used to retrieve a constructed value using
     * {@link jakarta.persistence.Tuple#get(String, Class)}
     * @return the explicitly specified alias or {@code null}
     */
    @Override
    @Nullable
    public String getAlias() {
        return alias;
    }

    /**
     * Specify an alias for this constructor result in the result set.
     * @param alias The alias
     */
    @Override
    @Nonnull
```

```java
    public ConstructorMapping<T> withAlias(@Nonnull String alias) {
        return new ConstructorMapping<>(targetClass, arguments, alias);
    }

    /**
     * Construct a new instance.
     * @param targetClass The Java class which declares the constructor
     * @param arguments Mappings assigned to the parameters of the constructor
     * @param <T> The type of the Java class
     */
    @Nonnull
    public static <T> ConstructorMapping<T> of(@Nonnull Class<T> targetClass,
                                               @Nonnull MappingElement<?>... arguments) {
        return new ConstructorMapping<>(targetClass, arguments, null);
    }

    /**
     * The Java class which declares the constructor.
     */
    @Override
    @Nonnull
    public Class<T> type() {
        return targetClass;
    }
}
```

G.10. TupleMapping

```java
import jakarta.annotation.Nonnull;
import jakarta.persistence.Tuple;

import static java.util.Objects.requireNonNull;

/**
 * Map columns of a JDBC {@link java.sql.ResultSet} to entries
 * in a tuple of values packaged as an instance of {@link Tuple}.
 *
 * <p>{@snippet :
 * import static jakarta.persistence.sql.ResultSetMapping.*;
 *
 * ...
 *
 * var isbn = column("isbn", String.class);
 * var title = column("title", String.class);
 * var pages = column("pages", int.class);
 *
 * entityManager.createNativeQuery("select isbn, title, pages from Book",
 *                                 tuple(isbn, title, pages))
 *     .getResultList().forEach(tuple -> {
 *         String bookIsbn = tuple.get(isbn);
 *         String bookTitle = tuple.get(title);
 *         int bookPages = tuple.get(pages);
 *         ...
 *     });
 * }
 *
 * <p>Each result {@link Tuple} contains an entry for each
 * {@linkplain #elements element} of the {@code TupleMapping}.
 * The index and alias of a tuple entry are determined by the
 * position of the element in {@code TupleMapping} and the
 * {@linkplain MappingElement#getAlias alias} of the element,
 * respectively.
 *
 *
 * @param elements Mappings for the elements of the tuple
```

```
 *
 * @see CompoundMapping
 * @see jakarta.persistence.SqlResultSetMapping
 *
 * @since 4.0
 */
public record TupleMapping(@Nonnull MappingElement<?>[] elements)
        implements ResultSetMapping<Tuple> {

    public TupleMapping(@Nonnull MappingElement<?>[] elements) {
        requireNonNull(elements, "elements are required");
        if (elements.length == 0) {
            throw new IllegalArgumentException("at least one element is required");
        }
        for (var element : elements) {
            requireNonNull(element, "element is required");
        }
        this.elements = elements.clone();
    }

    @Override
    @Nonnull
    public MappingElement<?>[] elements() {
        return elements.clone();
    }

    /**
     * Construct a new instance.
     * @param elements Mappings for the elements of the tuple
     */
    @Nonnull
    public static TupleMapping of(@Nonnull MappingElement<?>... elements) {
        return new TupleMapping(elements);
    }

    /**
     * Always returns {@link Tuple Tuple.class}.
     */
    @Override
    @Nonnull
    public Class<Tuple> type() {
        return Tuple.class;
    }
}
```