JAKARTA EE

Jakarta Annotations


2.1, December 01, 2021: Final Release
Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, 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, Inc. [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) 2018 Eclipse Foundation. 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 THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION 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.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION 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 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. Specification Scope

Jakarta Annotations defines a collection of annotations representing common semantic concepts that enable a declarative style of programming that applies across a variety of Java technologies.
Chapter 2. Introduction

With the addition of JSR 175 (A Metadata Facility for the JavaTM Programming Language) in the Java platform we envision that various technologies will use annotations to enable a declarative style of programming. It would be unfortunate if these technologies each independently defined their own annotations for common concepts. It would be valuable to have consistency within the Jakarta EE and Java SE component technologies, but it will also be valuable to allow consistency between Jakarta EE and Java SE.

It is the intention of this specification to define a small set of common annotations that will be available for use within other specifications. It is hoped that this will help to avoid unnecessary redundancy or duplication between annotations defined in different Jakarta EE specifications. This would allow us to have the common annotations all in one place and let the technologies refer to this specification rather than have them specified in multiple specifications. This way all technologies can use the same version of the annotations and there will be consistency in the annotations used across the platforms.

2.1. Goals

Define annotations for use in Jakarta EE: This spec will define annotations for use within component technologies in Jakarta EE as well as the platform as a whole.

2.2. Non-Goals

Support for Java versions prior to J2SE 5.0

Annotations were introduced in J2SE 5.0. It is not possible to do annotation processing in versions prior to J2SE 5.0. It is not a goal of this specification to define a way of doing annotation processing of any kind for versions prior to J2SE 5.0.

2.3. Compatibility

The annotations defined in this specification may be included individually as needed in products that make use of them. Other Java specifications will require support for subsets of these annotations. Products that support these Java specifications must include the required annotations.

2.4. Conventions


Java code is formatted as shown below:
2.5. Expert Group Members

The following expert group members participated in JSR 250:

- Cedric Beust (individual)
- Bill Burke (JBoss)
- Wayne Carr (Intel)
- Robert Clevenger (Oracle)
- Evan Ireland (Sybase)
- Woo Jin Kim (Tmax Soft)
- Gavin King (JBoss)
- Rajiv Mordani (Oracle Corporation, Specification lead)
- Ted Neward (individual)
- Anurag Parashar (Pramati technologies)
- Michael Santos (individual)
- Hani Suleiman (Ironflare AB)
- Seth White (BEA)

2.6. Acknowledgements

In addition to the expert group listed above, Linda DeMichiel, Ron Monzillo, Lance Andersen and Bill Shannon all of whom work at Oracle Corporation have provided input to this specification.
Chapter 3. Annotations

This chapter describes the standard annotations, some guidelines for annotation inheritance and the usage of these annotations where possible.

3.1. General Guidelines for Inheritance of Annotations

The interplay of annotations and inheritance in the Java language is potentially a source of complexity for developers. Developers will rely on some implicit assumptions when figuring out how annotations compose with other language features. At the same time, annotation semantics are defined by individual specifications, hence the potential for inconsistencies to arise. For instance, consider the following example:

```java
public class Base {
    @TransactionAttribute(REQUIRES_NEW)
    public void foo {...}
}

@Stateless
public class Derived extends Base {
    @TransactionAttribute(NEVER)
    public void foo {...}
}
```

In keeping with the concept of method overriding, most developers will assume that in the Derived class, the effective TransactionAttribute annotation for method foo is TransactionAttribute(NEVER). On the other hand, it might have been possible for the specification governing the semantics of the TransactionAttribute annotations type to require that the effective TransactionAttribute to be the most restrictive one in the whole inheritance tree, that is, in the example above TransactionAttribute(REQUIRES_NEW). A motivation for these semantics might have been that the foo method in the Derived class may call super.foo(), resulting in the execution of some code that needs a transaction to be in place. Such a choice on the part of the specification for TransactionAttribute would have contradicted a developer’s intuition on how method overriding works.

In order to keep the resulting complexity in control, below are some guidelines recommended for how annotations defined in the different specifications should interact with inheritance:

1. Class-level annotations only affect the class they annotate and its members, that is, its methods and fields. They never affect a member declared by a superclass, even if it is not hidden or overridden by the class in question.

2. In addition to affecting the annotated class, class-level annotations may act as a shorthand for member-level annotations. If a member carries a specific member-level annotation, any annotations of the same type implied by a class-level annotation are ignored. In other words,
explicit member-level annotations have priority over member-level annotations implied by a class-
level annotation. For example, a TransactionAttribute(REQUIRED) annotation on a class implies
that all the public methods in the class that it is applied on are annotated with
TransactionAttribute(REQUIRED). However if there is a TransactionAttribute(NEVER) annotation
on a particular method, then the TransactionAttribute(NEVER) applies for that particular method
and not TransactionAttribute(REQUIRED).

3. The interfaces implemented by a class never contribute annotations to the class itself or any of its
members.

4. Members inherited from a superclass and which are not hidden or overridden maintain the
annotations they had in the class that declared them, including member-level annotations implied
by class-level ones.

5. Member-level annotations on a hidden or overridden member are always ignored.

This set of guidelines guarantees that the effects of an annotation are local to the class on, or inside,
which it appears. In order to find the effective annotation for a class member, a developer has to track
down its last non-hidden and non-overridden declaration and examine it. If the sought-for annotation
is not found there, then (s)he will have to examine the enclosing class declaration. If even this step fails
to provide an annotation, no other source file will be consulted.

Below are some examples that explain how the guidelines defined above will be applied to the
TransactionAttribute annotation.
The table below shows the effective `TransactionAttribute` annotation in each of the cases above by applying the guidelines specified for annotations and inheritance:

<table>
<thead>
<tr>
<th>Methods in derived classes</th>
<th>Effective TransactionAttribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo() in ABean</td>
<td><code>TransactionAttribute(REQUIRED)</code>. (Default TransactionAttribute as defined by the Jakarta Enterprise Beans specification).</td>
</tr>
</tbody>
</table>
### 3.2. jakarta.annotation.Generated

The `Generated` annotation is used to mark source code that has been generated. It can be specified on a class, method, or field. It can also be used to differentiate user-written code from generated code in a single file.

The `value` element MUST have the name of the code generator. The recommended convention is to use the fully qualified name of the code generator. For example: `com.company.package.classname`.

The `date` element is used to indicate the date the source was generated. The `date` element MUST follow the ISO 8601 standard. For example the `date` element could have the following value:

```
2001-07-04T12:08:56.235-0700
```

which represents 2001-07-04 12:08:56 local time in the U.S. Pacific time zone.

The `comments` element is a place holder for any comments that the code generator may want to include in the generated code.
package jakarta.annotation;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target({ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
@Retention(SOURCE)
public @interface Generated {
    String[] value();
    String date() default "";
    String comments() default "";
}

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Name of the code generator</td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>Date source was generated. Must follow ISO 8601 standard</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>comments</td>
<td>Placeholder for comments that the generator may want to include in the generated code</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

The following example shows the usage of the annotation defined above:

    @Generated("com.sun.xml.rpc.AProcessor")
    public interface StockQuoteService extends java.rmi.Remote {
        this.context = context;
    }

### 3.3. jakarta.annotation.Resource

The Resource annotation is used to declare a reference to a resource. It can be specified on a class, method, or field. When the annotation is applied on a field or method, the container will inject an instance of the requested resource into the application when the application is initialized. If the annotation is applied to a class, the annotation declares a resource that the application will look up at runtime. Even though this annotation is not marked Inherited, all superclasses MUST be examined to discover all uses of this annotation. All such annotation instances specify resources that are needed by the application. Note that this annotation may appear on private fields and methods of superclasses. Injection of the declared resources needs to happen in these cases as well, even if a method with such an annotation is overridden by a subclass.

The name element is the JNDI name of the resource. When the Resource annotation is applied on a
field, the default value of the *name* element is the field name qualified by the class name. When applied on a method, the default is the JavaBeans property name corresponding to the method qualified by the class name. When applied on a class, there is no default and the name MUST be specified.

The *type* element defines the Java type of the resource. When the *Resource* annotation is applied on a field, the default value of the *type* element is the type of the field. When applied on a method, the default is the type of the JavaBeans property. When applied on a class, there is no default and the type MUST be specified. When used, the type MUST be assignment compatible.

The *authenticationType* element is used to indicate the authentication type to use for the resource. It can take one of two values defined as an *Enum*: `CONTAINER` or `APPLICATION`. This element may be specified for resources representing a connection factory of any supported type and MUST NOT be specified for resources of other types.

The *shareable* element is used to indicate whether a resource can be shared between this component and other components. This element may be specified for resources representing a connection factory of any supported type or ORB object instances and MUST NOT be specified for resources of other types.

The *mappedName* element is a product-specific name that this resource should be mapped to. The *mappedName* element provides for mapping the resource reference specified by the *Resource* annotation to the name of a resource known to the application server. The mapped name could be of any form. Application servers are not required to support any particular form or type of mapped name, nor the ability to use mapped names. The mapped name is product dependent and often installation dependent. No use of mapped name is portable.

The *description* element is the description of the resource. The description is expected to be in the default language of the system on which the application is deployed. The description can be presented to help in choosing the correct resource.

The *lookup* element specifies the JNDI name of a resource that the resource being defined will be bound to. The type of the referenced resource must be compatible with that of the resource being defined.
```java
package jakarta.annotation;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Repeatable(Resources.class)
public @interface Resource {
    public enum AuthenticationType {
        CONTAINER,
        APPLICATION
    }

    String name() default "";
    Class<?> type() default Object.class;
    AuthenticationType authenticationType() default AuthenticationType.CONTAINER;
    boolean shareable() default true;
    String mappedName() default "";
    String description() default "";
    String lookup() default "";
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The JNDI name of the resource</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>type</td>
<td>The Java type of the resource</td>
<td>Object.class</td>
</tr>
<tr>
<td>authenticationType</td>
<td>The authentication type to use for the resource</td>
<td>CONTAINER</td>
</tr>
<tr>
<td>shareable</td>
<td>Indicates whether the resource can be shared.</td>
<td>true</td>
</tr>
<tr>
<td>mappedName</td>
<td>A product-specific name that the resource should map to</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>description</td>
<td>Description of the resource</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>lookup</td>
<td>the JNDI name of a resource that the resource being defined will be bound to</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>
### 3.3.1. Field based injection

To access a resource a developer declares a field and annotates it as being a resource reference. If the name and type elements are missing from the annotation they will be inferred by looking at the field declaration itself. It is an error if the type specified by the `Resource` annotation and the type of the field are incompatible.

For example:

```java
@Resource
private DataSource myDB;
```

In the example above the effective name is `com.example.class/myDB` and the effective type is `javax.sql.DataSource.class`.

```java
@Resource(name="customerDB")
private DataSource myDB;
```

In the example above the name is `customerDB` and the effective type is `javax.sql.DataSource.class`.

### 3.3.2. Setter based injection

To access a resource a developer declares a setter method and annotates it as being a resource reference. The name and type of resource may be inferred by inspecting the method declaration if necessary. The name of the resource, if not declared, is the name of the JavaBeans property as determined from the name of the setter method. The setter method MUST follow the standard JavaBeans convention—the name starts with "set"; the return type is `void`; and there is only one parameter. Additionally, the type of the parameter MUST be compatible with the `type` element of the `Resource` annotation, if specified.

For example:

```java
@Resource
private void setMyDB(DataSource ds) {
    myDB = ds;
}
```

```java
private DataSource myDB;
```

In the example above the effective name is `com.example.class/myDB` and the type is `javax.sql.DataSource.class`.
In the example above the name is `customerDB` and the type is `javax.sql.DataSource.class`.

The table below shows the mapping from Java type to the equivalent resource type in the Jakarta EE 9 (and later) deployment descriptors:

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Equivalent Resource type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>env-entry</td>
</tr>
<tr>
<td>java.lang.Character</td>
<td>env-entry</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>env-entry</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>env-entry</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>env-entry</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>env-entry</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>env-entry</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>env-entry</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>env-entry</td>
</tr>
<tr>
<td>jakarta.xml.ws.Service</td>
<td>service-ref</td>
</tr>
<tr>
<td>jakarta.jws.WebService</td>
<td>service-ref</td>
</tr>
<tr>
<td>javax.sql.DataSource</td>
<td>resource-ref</td>
</tr>
<tr>
<td>jakarta.jms.ConnectionFactory</td>
<td>resource-ref</td>
</tr>
<tr>
<td>jakarta.jms.QueueConnectionFactory</td>
<td>resource-ref</td>
</tr>
<tr>
<td>jakarta.jms.TopicConnectionFactory</td>
<td>resource-ref</td>
</tr>
<tr>
<td>jakarta.mail.Session</td>
<td>resource-ref</td>
</tr>
<tr>
<td>java.net.URL</td>
<td>resource-ref</td>
</tr>
<tr>
<td>jakarta.resource.cci.ConnectionFactory</td>
<td>resource-ref</td>
</tr>
<tr>
<td>any other connection factory defined by a resource adapter</td>
<td>resource-ref</td>
</tr>
<tr>
<td>jakarta.jms.Queue</td>
<td>message-destination-ref</td>
</tr>
<tr>
<td>jakarta.jms.Topic</td>
<td>message-destination-ref</td>
</tr>
</tbody>
</table>
### 3.4. jakarta.annotation.Resources

The *Resource* annotation is used to declare a reference to a resource. The *Resources* annotation acts as a container for multiple resource declarations.

```java
package jakarta.annotation;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target({TYPE})
@Retention(RUNTIME)
public @interface Resources {
    Resource[] value;
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Container for defining multiple resources.</td>
<td></td>
</tr>
</tbody>
</table>

The following example shows the usage of the annotation defined above:

```java
@Resources ( {
    @Resource (name="myDB", type=javax.sql.DataSource),
    @Resource (name="myMQ", type=jakarta.jmsConnectionFactory)
})

public class CalculatorBean {
    // ...
}
```

### 3.5. jakarta.annotation.PostConstruct

The *PostConstruct* annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization. This method MUST be invoked before the class is put into service. This annotation MUST be supported on all classes that support dependency injection.

```java
@PostConstruct
public void init() {
    // init method
}
```
method annotated with `PostConstruct` MUST be invoked even if the class does not request any resources to be injected. Only one method in a given class can be annotated with this annotation. The method on which the `PostConstruct` annotation is applied MUST fulfill all of the following requirements, except in cases where these requirements have been relaxed by another specification. See in particular the `Jakarta Interceptors` specification.

- The method MUST NOT have any parameters.
- The return type of the method MUST be `void`.
- The method MUST NOT throw a checked exception.
- The method on which `PostConstruct` is applied MAY be `public`, `protected`, package private or `private`.
- The method MUST NOT be static except for the application client.
- In general, the method MUST NOT be final. However, other specifications are permitted to relax this requirement on a per-component basis.
- If the method throws an unchecked exception the class MUST NOT be put into service.

```java
package jakarta.annotation;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target(METHOD)
@Retention(RUNTIME)
public @interface PostConstruct {
}
```

The following example shows the usage of the annotation defined above:

```java
@Resource
private void setMyDB(DataSource ds) {
    myDB = ds;
}

@PostConstruct
private void initialize() {
    // Initialize the connection object from the DataSource
    connection = myDB.getConnection();
}

private DataSource myDB;
private Connection connection;
```
3.6. jakarta.annotation.PreDestroy

The `PreDestroy` annotation is used on a method as a callback notification to signal that the instance is in the process of being removed by the container. The method annotated with `PreDestroy` is typically used to release resources that the instance has been holding. This annotation MUST be supported by all container managed objects that support `PostConstruct` except the application client. The method on which the `PreDestroy` annotation is applied MUST fulfill all of the following requirements, except in cases where these requirements have been relaxed by another specification. See in particular the `Jakarta Interceptors` specification.

- The method MUST NOT have any parameters.
- The return type of the method MUST be `void`.
- The method MUST NOT throw a checked exception.
- The method on which `PreDestroy` is applied MAY be `public`, `protected`, package private or `private`.
- The method MUST NOT be static.
- In general, the method MUST NOT be final. However, other specifications are permitted to relax this requirement on a per-component basis.
- If the method throws an unchecked exception it is ignored.

```java
package jakarta.annotation;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target(METHOD)
@Retention(RUNTIME)
public @interface PreDestroy {
}
```

The following example shows the usage of the annotation defined above:
@Resource
private void setMyDB(DataSource ds) {
    myDB = ds;
}

@PostConstruct
private void initialize() {
    // Initialize the connection object from the DataSource
    connection = myDB.getConnection();
}

@PreDestroy
private void cleanup() {
    // Close the connection to the DataSource.
    connection.close();
}

private DataSource myDB;
private Connection connection;

3.7. jakarta.annotation.Priority

The Priority annotation can be applied to any program elements to indicate in what order they should be used. The effect of using the Priority annotation in any particular instance is defined by other specifications that define the use of a specific class.

For example, the Jakarta Interceptors specification defines the use of priorities on interceptors to control the order in which interceptors are called.

Priority values should generally be non-negative, with negative values reserved for special meanings such as “undefined” or “not specified”. A specification that defines use of the Priority annotation may define the range of allowed priorities and any priority values with special meaning.
3.8. jakarta.annotation.Nonnull

The *Nonnull* annotation is used to mark elements that cannot be *null*.

This information can be used for validation by IDEs, static analysis tools, and runtime.

The annotation may be present on any target. This specification defines behavior on following targets:

- Method - return type will never be *null*
- Parameter - parameter must not be *null*
- Field - field cannot be *null* after construction of the object is completed

The following example shows the usage of the annotation defined above:
3.9. jakarta.annotation.Nullable

The Nullable annotation is used to mark elements that may be null.

This information can be used for validation by IDEs, static analysis tools, and runtime.

The annotation may be present on any target. This specification defines behavior on following targets:

- Method - return type may be null
- Parameter - parameter may be null
- Field - field may be null

```java
package jakarta.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Documented
@Retention(RUNTIME)
public @interface Nullable {
}
```

The following example shows the usage of the annotation defined above:

```java
public interface StockQuoteService {
    BigDecimal quote(@NonNull String marker);
}
```

3.10. jakarta.annotation.security.RunAs

The RunAs annotation defines the security role of the application during execution in a Jakarta EE container. It can be specified on a class. This allows developers to execute an application under a particular role. The role MUST map to the user / group information in the container's security realm. The value element in the annotation is the name of a security role.
package jakarta.annotation.security;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target(TYPE)
@Retention(RUNTIME)
public @interface RunAs {
    String value();
}

The following example shows the usage of the annotation defined above:

```java
@RunAs("Admin")
public class Calculator {
    // ...
}
```

### 3.11. jakarta.annotation.security.RolesAllowed

The `RolesAllowed` annotation specifies the security roles permitted to access method(s) in an application. The value element of the `RolesAllowed` annotation is a list of security role names.

The `RolesAllowed` annotation can be specified on a class or on method(s). Specifying it at a class level means that it applies to all the methods in the class. Specifying it on a method means that it is applicable to that method only. If applied at both the class and method level, the method value overrides the class value.
package jakarta.annotation.security;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface RolesAllowed {
  String[] value();
}

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>List of roles permitted to access methods in the application</td>
<td></td>
</tr>
</tbody>
</table>

The following example shows the usage of the annotation defined above:

```java
@RolesAllowed("Users")
public class Calculator {
  @RolesAllowed("Administrator")
  public void setNewRate(int rate) {
    // ...
  }
}
```

### 3.12. `jakarta.annotation.security.PermitAll`

The `PermitAll` annotation specifies that all security roles are allowed to invoke the specified method(s), that is, that the specified method(s) are “unchecked”. It can be specified on a class or on methods. Specifying it on the class means that it applies to all methods of the class. If specified at the method level, it only affects that method.

```java
package jakarta.annotation.security;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface PermitAll {
}
```
The following example shows the usage of the annotation defined above:

```java
import jakarta.annotation.security.*;

@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        // ...
    }

    @PermitAll
    public long convertCurrency(long amount) {
        // ...
    }
}
```

### 3.13. jakarta.annotation.security.DenyAll

The `DenyAll` annotation specifies that no security roles are allowed to invoke the specified method(s), that is, that the method(s) are to be excluded from execution in the Jakarta EE container.

```java
package jakarta.annotation.security;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface DenyAll {
}
```

The following example shows the usage of the annotation defined above:
import jakarta.annotation.security.*;

@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        // ...
    }

    @DenyAll
    public long convertCurrency(long amount) {
        // ...
    }
}

3.14. PermitAll, DenyAll and RolesAllowed interactions

The PermitAll, DenyAll and RolesAllowed annotations all define which security roles are allowed to access the methods on which they are applied. This section describes how these annotations interact and which usages of these annotations are valid.

If the PermitAll, DenyAll and RolesAllowed annotations are applied on methods of a class, then the method level annotations take precedence (at the corresponding methods) over any class level annotations of type PermitAll, DenyAll and RolesAllowed.

3.15. jakarta.annotation.security.DeclareRoles

The DeclareRoles annotation is used to specify security roles used by the application. It can be specified on a class. It typically would be used to define roles that could be tested (i.e., by calling isUserInRole) from within the methods of the annotated class. It could also be used to declare roles that are not implicitly declared as the result of their use in a RolesAllowed annotation on the class or a method of the class.
The following example shows the usage of the annotation defined above:

```java
@DeclareRoles("BusinessAdmin")
public class Calculator {
    public void convertCurrency() {
        if (x.isUserInRole("BusinessAdmin")) {
            // ...
        }
    }
    // ...
}
```

### 3.16. jakarta.annotation.sql.DataSourceDefinition

The `DataSourceDefinition` annotation is used to define a container `DataSource` to be registered with JNDI. The `DataSource` may be configured by setting the annotation elements for commonly-used `DataSource` properties. Additional standard and vendor-specific properties may be specified using the `properties` element. The data source will be registered under the name specified in the `name` element. It may be defined to be in any valid Jakarta EE namespace, which will determine the accessibility of the data source from other components. A JDBC driver implementation class of the appropriate type, either `DataSource`, `ConnectionPoolDataSource`, or `XADataSource`, must be indicated by the `className` element. The driver class is not required to be available at deployment but must be available at runtime prior to any attempt to access the `DataSource`.

_DataSource_ properties should not be specified more than once. If the _url_ annotation element contains a _DataSource_ property that was also specified using the corresponding annotation element or was specified in the _properties_ annotation element, the precedence order is undefined and implementation specific.

Vendors are not required to support `properties` that do not normally apply to a specific data source type. For example, specifying the `transactional` property to be `true` but supplying a value for `className` that implements a data source class other than `XADataSource` may not be supported.

Vendor-specific properties may be combined with or used to override standard data source properties defined using this annotation.

`DataSource` properties that are specified and are not supported in a given configuration or cannot be
mapped to a vendor-specific configuration property may be ignored.

Although the annotation allows you to specify a password, it is recommended not to embed passwords in production code. The `password` element in the annotation is provided as a convenience for ease of development.

```java
package jakarta.annotation.sql;

import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(DataSourceDefinitions.class)
public @interface DataSourceDefinition {
  String name();
  String className();
  String description() default "";
  String url() default "";
  String user() default "";
  String password() default "";
  String databaseName() default "";
  int portNumber() default -1;
  String serverName() default "localhost";
  int isolationLevel() default -1;
  boolean transactional() default true;
  int initialPoolSize() default -1;
  int maxPoolSize() default -1;
  int minPoolSize() default -1;
  int maxIdleTime() default -1;
  int maxStatements() default -1;
  String[] properties() default ";{};
  int loginTimeout() default 0;
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>JNDI name by which the data source will be registered</td>
<td></td>
</tr>
<tr>
<td><code>className</code></td>
<td>DataSource implementation class name</td>
<td></td>
</tr>
<tr>
<td><code>description</code></td>
<td>Description of the data source</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Element</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>url</td>
<td>A JDBC URL. If the url annotation element contains a DataSource property that was also specified using the corresponding annotation element, the precedence order is undefined and implementation specific.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>user</td>
<td>User name for connection authentications</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>password</td>
<td>Password for connection authentications</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>databaseName</td>
<td>Name of a database on a server</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>portNumber</td>
<td>Port number where a server is listening for requests</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>serverName</td>
<td>Database server name</td>
<td>&quot;localhost&quot;</td>
</tr>
<tr>
<td>isolationLevel</td>
<td>Isolation level for connections.</td>
<td>-1 (vendor specific)</td>
</tr>
<tr>
<td>transactional</td>
<td>Indicates whether a connection is transactional or not</td>
<td>true</td>
</tr>
<tr>
<td>initialPoolSize</td>
<td>Number of connections that should be created when a connection pool is initialized</td>
<td>-1 (vendor specific)</td>
</tr>
<tr>
<td>maxPoolSize</td>
<td>Maximum number of connections that should be concurrently allocated for a connection pool</td>
<td>-1 (vendor specific)</td>
</tr>
<tr>
<td>minPoolSize</td>
<td>Minimum number of connections that should be allocated for a connection pool</td>
<td>-1 (vendor specific)</td>
</tr>
<tr>
<td>maxIdleTime</td>
<td>The number of seconds that a physical connection should remain unused in the pool before the connection is closed for a connection pool</td>
<td>-1 (vendor specific)</td>
</tr>
<tr>
<td>Element</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>maxStatements</td>
<td>The total number of statements that a connection pool should keep open. A value of 0 indicates that the caching of statements is disabled for a connection pool</td>
<td>-1 (vendor specific)</td>
</tr>
<tr>
<td>properties</td>
<td>Used to specify vendor-specific properties and less commonly used DataSource properties. If a DataSource property is specified in the properties element and the annotation element for the property is also specified, the annotation element value takes precedence.</td>
<td>{}</td>
</tr>
<tr>
<td>loginTimeout</td>
<td>The maximum time in seconds that this data source will wait while attempting to connect to a database. A value of 0 specifies that the timeout is the default system timeout if there is one, otherwise it specifies that there is no timeout</td>
<td>0</td>
</tr>
</tbody>
</table>

Examples:

```java
@DataSourceDefinition(
    name="java:global/MyApp/MyDataSource",
    className="com.foobar.MyDataSource",
    portNumber=6689,
    serverName="myserver.com",
    user="lance",
    password="secret")
```

Using a URL:

```java
@DataSourceDefinition(
    name="java:global/MyApp/MyDataSource",
    className="org.apache.derby.jdbc.ClientDataSource",
    url="jdbc:derby://localhost:1527/myDB",
    user="lance",
    password="secret")
```
The `DataSourceDefinition` annotation is used to declare a container `DataSource`. The `DataSourceDefinitions` annotation acts as a container for multiple data source declarations.

```java
package jakarta.annotation.sql;

import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DataSourceDefinitions {
    DataSourceDefinition[] value();
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Container for defining multiple data sources.</td>
<td></td>
</tr>
</tbody>
</table>

The following example shows the usage of the annotation defined above:

```java
@DataSourceDefinitions(
    @DataSourceDefinition(name="java:global/MyApp/MyDataSource",
                          className="com.foobar.MyDataSource",
                          portNumber=6689,
                          serverName="myserver.com",
                          user="lance",
                          password="secret"),
    @DataSourceDefinition(name="java:global/MyApp/MyDataSource",
                          className="org.apache.derby.jdbc.ClientDataSource",
                          url="jdbc:derby://localhost:1527/myDB",
                          user="lance",
                          password="secret")
)
public class CalculatorBean {
    // ...
}
```
3.18. jakarta.annotation.ManagedBean

The `ManagedBean` annotation is used to declare a Jakarta Managed Bean as specified in the *Jakarta Managed Beans* specification. Jakarta Managed Beans are container-managed objects that support a small set of basic services such as resource injection, lifecycle callbacks and interceptors. A Jakarta Managed Bean may optionally have a name, a `String` specified via the `value` element.

```java
package jakarta.annotation;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

@Target(TYPE)
@Retention(RUNTIME)
public@interface ManagedBean {
    boolean value() default "";
}
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Name of the Jakarta Managed Bean</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

Examples:

```java
@ManagedBean("cart")
public class ShoppingCart {
    // ...
}
```
Chapter 4. References


Jakarta Enterprise Beans, v4.0. https://jakarta.ee/specifications/enterprise-beans/4.0

Jakarta Interceptors, 2.0. https://jakarta.ee/specifications/interceptors/2.0/

Jakarta Managed Beans. https://jakarta.ee/specifications/managedbeans/2.0/