JAKARTA EE

Jakarta Validation specification

Version 4.0, 2025-10-30: Draft

Table of Contents

```
Specification: Jakarta Validation

Version: 4.0

Status: Draft

Release: 2025-10-30
```

License

1. Introduction

This document is the specification of the Java API for JavaBean validation in Jakarta EE and Java SE. The technical objective of this work is to provide an object level constraint declaration and validation facility for the Java application developer, as well as a constraint metadata repository and query API.

It also offers method and constructor validation facilities to ensure constraints on their parameters and return values.

1.1. Expert group

This work has been conducted as part of the Jakarta Validation under the Jakarta EE Specification Process and formerly JSR 380, 349 and 303 under the Java Community Process Program. This specification is the result of the collaborative work of the members of the Expert Groups and the community at large.

The following persons have actively contributed to Bean Validation 2.0 as members of the JSR 380 expert group and the community at large in alphabetical order:

- Matt Benson
- Emmanuel Bernard (Red Hat, Inc.)
- Linda DeMichiel (Oracle)
- Hendrik Ebbers (Karakun AG)
- Hardy Ferentschik (Red Hat, Inc.)
- Christian Kaltepoth (ingenit GmbH & Co. KG)
- Werner Keil
- Marco Molteni (Genidea Sagl)
- Gunnar Morling (Red Hat, Inc.) - Specification Lead
- Michael Nascimento Santos
- Otavio Santana
- Guillaume Smet (Red Hat, Inc.)
- Tsuyoshi Yoshitomi (Fujitsu Limited)

The following persons have actively contributed to Bean Validation 1.1 as members of the JSR 349 expert group and the community at large in alphabetical order:

- Matt Benson
- Paul Benedict
- Emmanuel Bernard (Red Hat, Inc.) - Specification Lead
- Edward Burns (Oracle)
- Peter Davis
- Linda DeMichiel (Oracle)
- Hardy Ferentschik (Red Hat, Inc.)
- Antonio Goncalves
- Cemalettin Koç
- Rich Midwinter
- Gunnar Morling (individual then Red Hat, Inc.)
- Pete Muir (Red Hat, Inc.)
- Michael Nascimento Santos
- Gerhard Petracek
- Kevin Pollet (SERLI)
- Jagadish Prasath Ramu (Oracle)
- Bill Shannon (Oracle)
- Sebastian Thomschke

Former expert group members of JSR-303 in alphabetical order are:

- Geert Bevin
- Emmanuel Bernard (Red Hat, Inc.) - Specification Lead
- Uri Boness
- Erik Brakkee (Ericsson AB)
- Ed Burns (Sun Microsystems, Inc.)
- Jason Carreira

- Robert Clevenger (Oracle - retired)
- Linda DeMichiel (Sun Microsystems, Inc.)
- Tim Fennel
- Bharath Ganesh (Pramati Technologies)
- Romain Guy (Google Inc.)
- Robert Harrop
- Jacob J. Hookom
- Bob Lee (Google Inc.)
- Craig R. McClanahan (Sun Microsystems, Inc.)
- Niall K. Pemberton
- Steve Peterson
- Dhanji R. Prasanna (Google Inc., formerly individual)
- Gerhard Petracek
- Matt Raible
- Michael Nascimento Santos
- Sebastian Thomschke
- Jon Wetherbee (Oracle)

1.2. Specification goals

Validating data is a common task that occurs throughout an application, from the presentation layer to the persistence layer. Often the same validation logic is implemented in each layer, proving to be time consuming and error-prone. To avoid duplication of these validations in each layer, developers often bundle validation logic directly into the domain model, cluttering domain classes with validation code that is, in fact, metadata about the class itself.

This specification defines a metadata model and API for JavaBean validation. The default metadata source is annotations, with the ability to override and extend the metadata through the use of XML validation descriptors.

The validation API developed by this specification is not intended for use in any one tier or programming model. It is specifically not tied to either the web tier or the persistence tier, and is available for both server-side application programming, as well as rich client Swing application developers. This API is seen as a general extension to the JavaBeans object model, and as such is expected to be used as a core component in other specifications. Ease of use and flexibility have influenced the design of this specification.

As of version 1.1, Jakarta Validation constraints can also be applied to the parameters and return values of methods of arbitrary Java types. Thus the Jakarta Validation API can be used to describe and validate the contract (comprising pre- and postconditions) applying to a given method ("Programming by Contract", PbC). Note that it is *not* the goal of this specification to develop a fully-fledged PbC solution but rather an easy-to-use facility satisfying the most common needs related to applying constraints to method parameters and return values, based on the proven concepts of the Jakarta Validation API.

1.3. Required Java version

The specification uses Java 17 language features. There is no requirement that implementations be compatible with Java language versions prior to 17.

1.4. How this document is organized

This document describes each aspect of the Jakarta Validation specification in a separate chapter. One should remember that the specification is a consistent whole.

Constraint definition describes how constraints are defined.

Value extractor definition describes how extractors for the values of container types are defined.

Constraint declaration and validation process describes how a JavaBean class is decorated with annotations to describe constraints.

Validation APIs describes how to programmatically validate a JavaBean.

Constraint metadata request APIs describes how the metadata query API works.

Built-in Constraint definitions list all the built-in constraints.

XML deployment descriptor describes the XML deployment descriptors for the configuration and the mapping.

Exception model describes the exception model and hierarchy used by Jakarta Validation.

Integration describes the different integration points of Jakarta Validation with other technologies. In some cases one has to refer to the

respective specifications for the up-to-date integration rules.

In Terminology, key concepts are summarized. Some reviewers have found that reading the terminology section first helps to better understand the specification.

The changelog can be found at Changelog.

1.5. How to comment

The expert group is eager to receive feedback from readers. Feel free to contact us. You can get all the details at http://beanvalidation.org/contribute/.

2. What's new

NOTE Names used under the JCP for specifications are preserved in the What's new section for versions released prior to the move to Jakarta EE in order to preserve historical accuracy.

2.1. What's new in 4.0

Changes introduced in Jakarta Validation 4.0:

- Support for `ConstraintValidator` declaration via the service loader mechanism was added.
- The `validatedBy` attribute on the `jakarta.validation.Constraint` was made optional; an empty array being the default.
- A simpler way to get a single attribute from `ConstraintDescriptor` was introduced.
- Support for string representation of numbers by `Min` / `Max` constraints was clarified.
- Some methods of the `ConstraintViolationBuilder` were deprecated. Previously, they were only deprecated in the Javadoc comments.
- The expectations on applying `@Valid` twice were clarified.

2.2. What's new in 3.1

Two minor changes are introduced in Jakarta Validation 3.1:

- Jakarta Bean Validation has been renamed to simply Jakarta Validation.
- Support for Java records has been clarified.

2.3. What's new in 3.0

The only changes in Bean Validation 3.0 are changes to support the Jakarta EE javax to jakarta package namespace change.

The changes include:

- All `javax.validation.*` packages have moved to `jakarta.validation.*`.
- The namespace for Bean Validation XML descriptors has been changed to `https://jakarta.ee/xml/ns/validation/configuration` for `META-INF/validation.xml` and `https://jakarta.ee/xml/ns/validation/mapping` for constraint mapping files (see XML configuration: META-INF/validation.xml)

2.4. What's new in 2.0

The main contribution of Bean Validation 2.0 is leveraging the new language features and API additions of Java 8 for the purposes of validation. Java 8 or later is required to use Bean Validation 2.0.

The changes include:

- Support for validating container elements by annotating type arguments of parameterized types, e.g. `List<@Positive Integer> positiveNumbers` (see Container element constraints); this also includes:
  - More flexible cascaded validation of collection types; e.g. values *and* keys of maps can be validated now: `Map<@Valid CustomerType, @Valid Customer> customersByType`
  - Support for `java.util.Optional`
  - Support for the property types declared by JavaFX
  - Support for custom container types by plugging in additional value extractors (see Value extractor definition)
- Support for the new date/time data types for `@Past` and `@Future` (see Built-in Constraint definitions); fine-grained control over the current time and time zone used for validation (see Implementation of temporal constraint validators)
- New built-in constraints: `@Email`, `@NotEmpty`, `@NotBlank`, `@Positive`, `@PositiveOrZero`, `@Negative`, `@NegativeOrZero`, `@PastOrPresent` and `@FutureOrPresent` (see Built-in Constraint definitions)
- All built-in constraints are marked as repeatable now
- Parameter names are retrieved using reflection (see Naming parameters)
- `ConstraintValidator#initialize()` is a default method (see Constraint validation implementation)
- The namespace for Bean Validation XML descriptors has been changed to `http://xmlns.jcp.org/xml/ns/validation/configuration` for `META-INF/validation.xml` and `http://xmlns.jcp.org/xml/ns/validation/mapping` for constraint mapping files (see XML configuration: META-INF/validation.xml)

2.5. What's new in 1.1

Bean Validation 1.1 improves and builds upon Bean Validation 1.0. The expert group and the community have been working on a few specific areas.

2.5.1. Openness

All of Bean Validation 1.1 work has been done in the open and in an open source way. Source code for the API, reference implementation, test compatibility kit as well as the specification and the website sources are available in the open. All discussions are done in the open in the publicly available development mailing list. Road map and proposals are also published on the website.

You can find all the details (mailing lists, source repositories etc.) at http://beanvalidation.org.

2.5.2. Dependency injection

Bean Validation uses a few components `MessageInterpolator`, `TraversableResolver`, `ParameterNameProvider`, `ConstraintValidatorFactory` and `ConstraintValidator`. Bean Validation 1.1 standardizes how these objects are managed by a container and how these objects can benefit from container services. In particular, CDI support within Java EE is being defined.

2.5.3. Method validation

Bean Validation 1.1 allows to put constraints to the parameters and return values of arbitrary methods and constructors. That way the Bean Validation API can be used to describe and validate the contract applying to a given method or constructor, that is:

- the preconditions that must be met by the caller before the method or constructor may be invoked and
- the postconditions that are guaranteed to the caller after a method or constructor invocation returns.

This enables a programming style known as "Programming by Contract" (PbC). Compared to traditional means of checking the sanity of argument and return values this approach has several advantages:

- These checks are expressed declaratively and don't have to be performed manually, which results in less code to write, read and maintain.
- The pre- and postconditions applying for a method or constructor don't have to be expressed again in the documentation, since any of its annotations will automatically be included in the generated JavaDoc. This reduces redundancies, thus avoiding efforts and inconsistencies between implementation and documentation.

2.5.4. Integration with CDI

The integration points with CDI have been increased and reworked. This opens up for a more natural and standard integration both in Java EE and Java SE and encompass dependency injection, component lifecycle management and interception for method validation.

2.5.5. Group conversion

The specification offers a way to alter the targeted group when validation cascading is happening. This feature is particularly useful to reuse a given object (graph) and to avoid leaking groups between various object subgraphs. It also makes for more readable constraints.

2.5.6. Message interpolation via the unified expression language

Constraint violation messages can now use EL expressions for a much more flexible rendering and string formatting. In particular a formatter object is injected in the EL context to convert numbers, dates etc. into the locale specific string representation. Likewise, the validated value is also available in the EL context.

2.5.7. Others

Many more minor changes have been done. Check out the change log for more details at Changelog.

3. Constraint definition

Constraints are defined by the combination of a constraint annotation and a list of constraint validation implementations. The constraint annotation is applied on types, fields, methods, constructors, parameters, container elements or other constraint annotations in case of composition.

Unless stated otherwise the default package name for the Jakarta Validation APIs is `jakarta.validation`.

3.1. Constraint annotation

A constraint on a JavaBean is expressed through one or more annotations. An annotation is considered a constraint definition if its retention policy contains `RUNTIME` and if the annotation itself is annotated with `jakarta.validation.Constraint` .

Listing 3.1: `@Constraint` *annotation*

```
/**
 * Marks an annotation as being a Jakarta Validation constraint.
 * <p>
 * A given constraint annotation must be annotated by a {@code @Constraint}
 * annotation which refers to its list of constraint validation implementations.
 * <p>
 * Each constraint annotation must host the following attributes:
 * <ul>
 *     <li>{@code String message() default [...];} which should default to an error
 *     message key made of the fully-qualified class name of the constraint followed by
 *     {@code .message}. For example {@code "{com.acme.constraints.NotSafe.message}"}</li>
 *     <li>{@code Class<?>[] groups() default {};} for user to customize the targeted
 *     groups</li>
 *     <li>{@code Class<? extends Payload>[] payload() default {};} for
 *     extensibility purposes</li>
 * </ul>
 * <p>
 * When building a constraint that is both generic and cross-parameter, the constraint
 * annotation must host the {@code validationAppliesTo()} property.
 * A constraint is generic if it targets the annotated element and is cross-parameter if
 * it targets the array of parameters of a method or constructor.
 * <pre>
 *     ConstraintTarget validationAppliesTo() default ConstraintTarget.IMPLICIT;
 * </pre>
 * This property allows the constraint user to choose whether the constraint
 * targets the return type of the executable or its array of parameters.
 *
 * A constraint is both generic and cross-parameter if
 * <ul>
 *     <li>two kinds of {@link ConstraintValidator}s are attached to the
 *     constraint, one targeting {@link ValidationTarget#ANNOTATED_ELEMENT}
 *     and one targeting {@link ValidationTarget#PARAMETERS},</li>
 *     <li>or if a {@link ConstraintValidator} targets both
 *     {@link ValidationTarget#ANNOTATED_ELEMENT} and {@link ValidationTarget#PARAMETERS}.</li>
 * </ul>
 *
 * Such dual constraints are rare. See {@link SupportedValidationTarget} for more info.
 * <p>
 * Here is an example of constraint definition:
 * <pre>
 * &#64;Documented
 * &#64;Constraint(validatedBy = OrderNumberValidator.class)
 * &#64;Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
 * &#64;Retention(RUNTIME)
 * public &#64;interface OrderNumber {
 *     String message() default "{com.acme.constraint.OrderNumber.message}";
 *     Class&lt;?&gt;[] groups() default {};
 *     Class&lt;? extends Payload&gt;[] payload() default {};
 * }
 * </pre>
 *
 * @see Payload
 * @see ConstraintTarget
```

```
 *
 * @author Emmanuel Bernard
 * @author Gavin King
 * @author Hardy Ferentschik
 */
@Documented
@Target({ ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface Constraint {

    /**
     * {@link ConstraintValidator} classes implementing the constraint. The given classes
     * must reference distinct target types for a given {@link ValidationTarget}. If two
     * {@link ConstraintValidator}s refer to the same type, an exception will occur.
     * <p>
     * At most one {@link ConstraintValidator} targeting the array of parameters of
     * methods or constructors (aka cross-parameter) is accepted. If two or more
     * are present, an exception will occur.
     *
     * @return array of {@link ConstraintValidator} classes implementing the constraint
     */
    Class<? extends ConstraintValidator<?, ?>>[] validatedBy() default {};
}
```

A constraint is said to be generic if it has at least one constraint validator targeting the element annotated i.e. targeting the (returned) element annotated by the constraint (a bean, a field, a getter, a method/constructor return value or a method/constructor parameter). A constraint is said to be cross-parameter if it has one constraint validator targeting the array of parameters of a method or constructor (to validate the consistency of several method/constructor parameters). A Jakarta Validation constraint is most of the time either a generic constraint or a cross-parameter constraint. In rare situations, a constraint can be both.

Generic constraint annotations can target any of the following `ElementType`s:

- `FIELD` for constrained attributes
- `METHOD` for constrained getters and constrained method return values
- `CONSTRUCTOR` for constrained constructor return values
- `PARAMETER` for constrained method and constructor parameters
- `TYPE` for constrained beans
- `ANNOTATION_TYPE` for constraints composing other constraints
- `TYPE_USE` for container element constraints

Cross-parameter constraint annotations can target any of the following `ElementType`s:

- `METHOD`
- `CONSTRUCTOR`
- `ANNOTATION_TYPE` for cross-parameter constraints composing other cross-parameter constraints

A constraint annotation that is both can target the union of the generic and cross-parameter constraint annotations targets.

While other `ElementType`s are not forbidden, the provider does not have to recognize and process constraints placed on such types.

Since a given constraint definition applies to one or more specific Java types, the JavaDoc for the constraint annotation should clearly state which types are supported. Applying a constraint annotation to an incompatible type will raise an `UnexpectedTypeException`. Care should be taken on defining the list of `ConstraintValidator`s. The type resolution algorithm (see ConstraintValidator resolution algorithm) could lead to exceptions if the `ConstraintValidator` list leads to ambiguities.

At most one `ConstraintValidator` supporting cross-parameter validation must be present for a given constraint. A `ConstraintDefinitionException` is raised otherwise. The JavaDoc should clearly state if the constraint is a generic and / or a cross-parameter constraint.

If a constraint definition is not valid, a `ConstraintDefinitionException` is raised either at validation time or when the metadata is requested. Invalid constraint definitions causes are multiple but include missing or illegal `message` or `groups` elements (see Constraint definition properties).

NOTE

Jakarta Validation defines rules for applying constraint annotations in inheritance hierarchies, described in Inheritance (interface and superclass) and Method constraints in inheritance hierarchies. It is therefore not recommended to specify the meta annotation `java.lang.annotation.Inherited` at constraint annotation types, as it is not relevant in the context of Jakarta Validation and would conflict with the proposed rules.

3.1.1. Constraint definition properties

A constraint definition may have attributes that are specified at the time the constraint is applied to a JavaBean. The properties are mapped as annotation elements. The annotation element names `message`, `groups`, `validationAppliesTo` and `payload` are considered reserved names; annotation elements starting with `valid` are not allowed ; a constraint may use any other element name for its attributes.

3.1.1.1. message

Every constraint annotation must define a `message` element of type `String`.

```
String message() default "{com.acme.constraint.MyConstraint.message}";
```

The `message` element value is used to create the error message. See Message interpolation for a detailed explanation. It is recommended to default `message` values to resource bundle keys to enable internationalization. It is also recommended to use the following convention: the resource bundle key should be the fully qualified class name of the constraint annotation concatenated to `.message` as shown in the previous program listing.

Built-in Jakarta Validation constraints follow this convention.

3.1.1.2. groups

Every constraint annotation must define a `groups` element that specifies the processing groups with which the constraint declaration is associated. The type of the `groups` parameter is `Class<?>[]`.

```
Class<?>[] groups() default {};
```

The default value must be an empty array.

If no group is specified when declaring the constraint on an element, the `Default` group is considered declared.

See groups for more information.

Groups are typically used to control the order in which constraints are evaluated, or to perform validation of the partial state of a JavaBean.

3.1.1.3. payload

Constraint annotations must define a `payload` element that specifies the payload with which the constraint declaration is associated. The type of the `payload` parameter is `Payload[]`.

```
Class<? extends Payload>[] payload() default {};
```

The default value must be an empty array.

Each attachable payload extends `Payload`.

Listing 3.2: `Payload` *interface*

```
/**
 * Payload type that can be attached to a given
 * constraint declaration.
 * <p>
 * Payloads are typically used to carry on metadata information
 * consumed by a validation client.
 * </p>
 * With the exception of the {@link Unwrapping} payload types, the use of payloads is not
 * considered portable.
 *
 * @author Emmanuel Bernard
 * @author Gerhard Petracek
 */
public interface Payload {
}
```

Payloads are typically used by validation clients to associate some metadata information with a given constraint declaration. Describing payloads as interface extensions as opposed to a string-based approach allows an easier and more type-safe approach. Payloads are

typically non-portable. An exception are the `Unwrapping.Skip` and `Unwrapping.Unwrap` payload types which are defined by this specification (see Implicit unwrapping of containers).

One use case for payload shown in Use of payload to associate severity to a constraint is to associate a severity to a constraint. This severity can be exploited by a presentation framework to adjust how a constraint failure is displayed.

Example 3.1: Use of payload to associate severity to a constraint

```
package com.acme.severity;

public class Severity {
    public static class Info implements Payload {};
    public static class Error implements Payload {};
}

public class Address {
    @NotNull(message="would be nice if we had one", payload=Severity.Info.class)
    public String getZipCode() { [...] }

    @NotNull(message="the city is mandatory", payload=Severity.Error.class)
    String getCity() { [...] }
}
```

The `payload` information can be retrieved from error reports via the `ConstraintDescriptor` either accessed through the `ConstraintViolation` objects (see ConstraintViolation) or through the metadata API (see ConstraintDescriptor).

3.1.1.4. validationAppliesTo

`validationAppliesTo` is used at constraint declaration time to clarify what the constraint targets (i.e. the annotated element, the method return value or the method parameters).

The element `validationAppliesTo` must only be present for constraints that are both generic and cross-parameter, it is mandatory in this situation. A `ConstraintDefinitionException` is raised if these rules are violated.

The type of the `validationAppliesTo` parameter is `ConstraintTarget`. The default value must be `ConstraintTarget.IMPLICIT`.

Listing 3.3: validationAppliesTo and ConstraintTarget

```
ConstraintTarget validationAppliesTo() default ConstraintTarget.IMPLICIT;


/**
 * Defines the constraint target.
 *
 * @author Emmanuel Bernard
 * @since 1.1
 */
public enum ConstraintTarget {

    /**
     * Discover the type when no ambiguity is present
     * <ul>
     *     <li>if neither on a method nor a constructor, it implies the annotated element
     *     (type, field etc),</li>
     *     <li>if on a method or constructor with no parameter, it implies
     *     {@code RETURN_VALUE},</li>
     *     <li>if on a method with no return value ({@code void}), it implies
     *     {@code PARAMETERS}.</li>
     * </ul>
     * Otherwise, {@code IMPLICIT} is not accepted and either {@code RETURN_VALUE} or
     * {@code PARAMETERS} is required. This is the case for constructors with parameters
     * and methods with parameters and return value.
     */
    IMPLICIT,

    /**
     * Constraint applies to the return value of a method or a constructor.
```

```
     */
    RETURN_VALUE,

    /**
     * Constraint applies to the parameters of a method or a constructor
     */
    PARAMETERS
}
```

If a `ConstraintTarget` is used in an illegal situation, a `ConstraintDeclarationException` is raised either at validation time or when the metadata is requested. Examples of illegal situations are:

- using `IMPLICIT` in a situation that cannot be inferred (see the JavaDoc for the detailed rules),
- using `PARAMETERS` on a constructor or method that has no parameter,
- using `RETURN_VALUE` on a method with no return value,
- using `PARAMETERS` or `RETURN_VALUE` on a type - class or interface - or on a field.

Constraint users are encouraged to explicitly set the `ConstraintTarget` target when using a constraint supporting both on a method or constructor as it improves readability.

3.1.1.5. Constraint specific parameter

The constraint annotation definitions may define additional elements to parameterize the constraint. For example, a constraint that validates the length of a string can use an annotation element named `length` to specify the maximum length at the time the constraint is declared.

3.1.2. Examples

Example 3.2: Simple constraint definition

```
//assuming OrderNumberValidator is a generic constraint validator

package com.acme.constraint;

/**
 * Mark a String as representing a well formed order number
 */
@Documented
@Constraint(validatedBy = OrderNumberValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface OrderNumber {

    String message() default "{com.acme.constraint.OrderNumber.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Simple constraint definition marks a `String` as a well-formed order number. The constraint validator is implemented by `OrderNumberValidator`.

Example 3.3: Simple cross-parameter constraint definition

```
//assuming DateParametersConsistentValidator is a cross-parameter
//constraint validator

package com.acme.constraint;

/**
 * Cross-parameter constraint ensuring that two date parameters of a method are in the
 * correct order.
```

```
 */
@Documented
@Constraint(validatedBy = DateParametersConsistentValidator.class)
@Target({ METHOD, CONSTRUCTOR, ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface DateParametersConsistent {

    String message() default "{com.acme.constraint.DateParametersConsistent.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Simple cross-parameter constraint definition shows a cross-parameter constraint which ensures that two date parameters of a method are in the correct order. The constraint validator is implemented by `DateParametersConsistentValidator`.

Example 3.4: Constraint that is both generic and cross parameter

```
//assuming ELAssertValidator is both a generic and cross-parameter
//constraint validator

package com.acme.constraint;

/**
 * Jakarta Expresson Language expression to be validated. This constraint accepts any type and can
 * validate both the annotated type or apply restrictions across parameters.
 */
@Documented
@Constraint(validatedBy = ELAssertValidator.class)
@Target({ METHOD, FIELD, TYPE, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface ELAssert {

    String message() default "{com.acme.constraint.ELAssert.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    ConstraintTarget validationAppliesTo() default ConstraintTarget.IMPLICIT;

    String expression();
}


@ELAssert(
    message="Please check that your passwords match and try again.",
    expression="param[1]==param[2]",
    validationAppliesTo=ConstraintType.PARAMETERS
)
public User createUser(String email, String password, String repeatPassword) { [...] }
```

Constraint that is both generic and cross parameter shows a constraint that can be applied both on the annotated element and across parameters of a method or a constructor. Note in this case the presence of `validationAppliesTo`.

Example 3.5: Constraint definition with default parameter

```
package com.acme.constraint;

/**
 * A frequency in Hz as audible to human ear. Adjustable to the age of the person. Accepts
 * Numbers.
 */
@Documented
@Constraint(validatedBy = AudibleValidator.class)
```

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface Audible {

    Age age() default Age.YOUNG;

    String message() default "{com.acme.constraint.Audible.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    public enum Age {
        YOUNG,
        WONDERING,
        OLD
    }
}
```

Constraint definition with default parameter ensures that a given frequency is within the scope of human ears. The constraint definition includes an optional parameter that may be specified when the constraint is applied.

Example 3.6: Constraint definition with mandatory parameter

```
package com.acme.constraint;

/**
 * Defines the list of values accepted. Accepts int or Integer objects.
 */
@Documented
@Constraint(validatedBy = DiscreteListOfIntegerValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface Acceptable {

    int[] value();

    String message() default "{com.acme.constraint.Acceptable.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Constraint definition with mandatory parameter defines a list of acceptable values expressed as an array: the `value` property must be specified when the constraint is applied.

3.2. Applying multiple constraints of the same type

It is often useful to declare the same constraint more than once to the same target, with different properties. A common example is the `@Pattern` constraint, which validates that its target matches a specified regular expression. Other constraints have this requirement as well. The same constraint type can belong to different groups and have specific error messages depending on the targeted group.

To support this requirement, the Jakarta Validation provider treats regular annotations (annotations not annotated by `@Constraint`) whose `value` element has a return type of an array of constraint annotations in a special way. Each element in the `value` array are processed by the Jakarta Validation implementation as regular constraint annotations. This means that each constraint specified in the `value` element is applied to the target. The annotation must have retention `RUNTIME` and can be applied on a type, field, property, executable parameter, executable return value, executable cross-parameter or another annotation. It is recommended to use the same set of targets as the initial constraint.

Note to constraint designers: each constraint annotation should be coupled with its corresponding multi-valued annotation. The specification recommends, though does not mandate, the definition of an inner annotation named `List`. Each constraint annotation type should be meta-annotated with `java.lang.annotation.Repeatable`, referencing the corresponding `List` annotation. This marks the constraint annotation type as repeatable and lets users specify the constraint several times without explicitly using the `List` annotation. All built-in annotations follow this pattern.

Example 3.7: Multi-valued constraint definition

```
/**
 * Validate a zip code for a given country
 * The only supported type is String
 */
@Documented
@Constraint(validatedBy = ZipCodeValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
public @interface ZipCode {

    String countryCode();

    String message() default "{com.acme.constraint.ZipCode.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    /**
     * Defines several @ZipCode annotations on the same element
     * @see (@link ZipCode}
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        ZipCode[] value();
    }
}
```

Example 3.8: Multi-valued constraint declaration

```
public class Address {
    @ZipCode(countryCode = "fr", groups = Default.class, message = "zip code is not valid")
    @ZipCode(
        countryCode = "fr",
        groups = SuperUser.class,
        message = "zip code invalid. Requires overriding before saving."
    )
    private String zipCode;
}
```

In this example, both constraints apply to the `zipCode` field but with different groups and with different error messages. It is also possible to specify a constraint several times by explicitly using the `@List` annotation (though simply repeating the annotation is the preferred idiom as of Jakarta Validation 2.0 and Java 8):

Example 3.9: Multi-valued constraint declaration using explicit `@List` annotation (discouraged)

```
public class Address {
    @ZipCode.List( {
        @ZipCode(countryCode="fr", groups=Default.class,
            message = "zip code is not valid"),
        @ZipCode(countryCode="fr", groups=SuperUser.class,
            message = "zip code invalid. Requires overriding before saving.")
    } )
    private String zipCode;
}
```

Using two different multi-constraint annotations for the same underlying constraint type on the same target (i.e. class or property) is not considered portable and is discouraged.

3.3. Constraint composition

This specification allows you to compose constraints to create higher level constraints.

Constraint composition is useful in several ways:

- Avoid duplication and facilitate reuse of more primitive constraints.
- Expose primitive constraints as part of a composed constraint in the metadata API and enhance tool awareness.

Composition is done by annotating a constraint annotation with the composing constraint annotations.

Example 3.10: Composition is done by annotating the composed constraint

```
@Pattern(regexp = "[0-9]*")
@Size(min = 5, max = 5)
@Constraint(validatedBy = FrenchZipCodeValidator.class)
@Documented
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface FrenchZipCode {

    String message() default "Wrong zip code";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        FrenchZipCode[] value();
    }
}
```

Annotating an element with `@FrenchZipCode` (the composed annotation) is equivalent to annotating it with `@Pattern(regexp="[0-9]*")`, `@Size(min=5, max=5)` (the composing annotations) and `@FrenchZipCode`. More formally, each constraint annotation hosted on a constraint annotation is applied to the target element and this is done recursively. Note that the main annotation and its constraint validation implementation is also applied. By default, each failing constraint generates an error report. Groups from the main constraint annotation are inherited by the composing annotations. Any `groups` definition on a composing annotation is ignored. Payload from the main constraint annotation is inherited by the composing annotations. Any `payload` definition on a composing annotation is ignored. The constraint target from the main constraint annotation is inherited by the composing annotations. Any `validationAppliesTo` definition on a composing annotation is ignored.

The type upon which composed constraint is placed must be compatible with all constraints (composing and composed). A constraint designer should ensure that such a type exists and lists in the JavaDoc all the compatible types.

All composed and composing constraints must have a constraint type in common. In particular, it is not legal to mix a pure generic constraint and a pure cross-parameter constraint.

It is possible to ensure that composing annotations do not raise individual error reports. In this scenario, if one or more composing annotations are invalid, the main constraint is automatically considered invalid and the corresponding error report is generated. To mark a constraint as raising a single constraint error report if either the composed or one of the composing constraints fail, use the `@ReportAsSingleViolation` annotation.

Example 3.11: If any of the composing constraints fail, the error report corresponding to `@FrenchZipCode` is raised and none other.

```
@Pattern(regexp = "[0-9]*")
@Size(min = 5, max = 5)
@ReportAsSingleViolation
@Constraint(validatedBy = FrenchZipCodeValidator.class)
@Documented
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface FrenchZipCode {

    String message() default "Wrong zip code";
```

```
    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        FrenchZipCode[] value();
    }
}
```

The definition of `@ReportAsSingleViolation` is as follows.

Listing 3.4: `@ReportAsSingleViolation` *annotation*

```
/**
 * A constraint annotation hosting this annotation will return the
 * composed annotation error report if any of the composing annotations fail.
 * The error reports of each individual composing constraint are ignored.
 * <p>
 * Note: Evaluation of composed constraints stops on the first validation
 * error in case the composing constraint is annotated with
 * {@code @ReportAsSingleViolation}.
 *
 * @author Emmanuel Bernard
 */
@Target({ ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface ReportAsSingleViolation {
}
```

More specifically, if a composed constraint is marked as `@ReportAsSingleViolation`, the evaluation of the composing constraints stops at the first failing constraint and the error report corresponding to the composed constraint is generated and returned.

Composing annotations can define the value of `message` and custom attributes (excluding `groups`, `payload` and `validationAppliesTo`) but these are fixed in the composed constraint definition.

Example 3.12: Composing annotations can use attributes. They are fixed for a given main annotation. All `@FrenchZipCode` *constraints have a* `@Size` *restricted to 5.*

```
@Pattern(regexp = "[0-9]*")
@Size(min = 5, max = 5)
@Constraint(validatedBy = FrenchZipCodeValidator.class)
@Documented
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface FrenchZipCode {

    String message() default "Wrong zip code";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        FrenchZipCode[] value();
    }
}
```

It is possible to override attributes and messages defined on a composing annotation. An attribute from the main annotation is used to override one or more attributes of the composing annotations. Such an attribute is annotated with one or more `@OverridesAttribute` annotations.

Example 3.13: Attributes from composing annotations can be overridden by attributes from the composed annotation

```
@Pattern(regexp = "[0-9]*")
@Size
@Constraint(validatedBy = FrenchZipCodeValidator.class)
@Documented
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface FrenchZipCode {

    String message() default "Wrong zip code";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @OverridesAttribute(constraint = Size.class, name = "min")
    @OverridesAttribute(constraint = Size.class, name = "max")
    int size() default 5;

    @OverridesAttribute(constraint = Size.class, name = "message")
    String sizeMessage() default "{com.acme.constraint.FrenchZipCode.zipCode.size}";

    @OverridesAttribute(constraint = Pattern.class, name = "message")
    String numberMessage() default "{com.acme.constraint.FrenchZipCode.number.size}";

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        FrenchZipCode[] value();
    }
}
```

The value of the composed constraint attribute annotated with `@OverridesAttribute` (`@FrenchZipCode.sizeMessage`) is applied to the composing constraint attribute named after `@OverridesAttribute.name` and hosted on the composing constraint of type `@OverridesAttribute.constraint` (`@Size.message`). Similarly, `@FrenchZipCode.numberMessage` value is mapped to `@Pattern.message`.

If left undefined, the default value for `@OverridesAttribute.name` is the name of the composed constraint attribute hosting the `@OverridesAttribute` annotation.

The types of the overridden and overriding attributes must be identical.

NOTE A composing constraint can itself be a composed constraint. In this case, attribute values are overridden recursively according to the described rules. Note however, that a forwarding rule (as defined by `@OverridesAttribute`) is only applied to the direct composing constraints.

Using Attributes from composing annotations can be overridden by attributes from the composed annotation,

```
@FrenchZipCode(size=9, sizeMessage="Zip code should be of size {max}")
```

is equivalent to

```
@FrenchZipCode
```

if `@FrenchZipCode` is defined as

```
@Pattern(regexp = "[0-9]*")
@Size(min = 9, max = 9, message = "Zip code should be of size {max}")
@Constraint(validatedBy = FrenchZipCodeValidator.class)
@Documented
```

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface FrenchZipCode {

    String message() default "Wrong zip code";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        FrenchZipCode[] value();
    }
}
```

If a constraint is used more than once as a composing constraint, the multi value constraints model as described in Applying multiple constraints of the same type is used.

To select a specific composing constraint, `OverridesAttribute.constraintIndex` is used. If the composing constraints are directly given on the composed constraint (i.e. via the repeatable annotation feature), `constraintIndex` refers to the left-to-right order of the constraints of this type in which they are given on the composed constraint. If the composing constraints are specified using their corresponding `List` annotation, `constraintIndex` refers to the index within the `value` array.

A composing constraint must not be given directly on the composed constraint and using the corresponding `List` annotation at the same time. A `ConstraintDeclarationException` will be raised in this case.

If `index` is undefined, the single constraint declaration is targeted.

Example 3.14: Use of constraintIndex in @OverridesAttribute

```
@Documented
@Constraint(validatedBy = {})
@Pattern(regexp = "[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}") // email
@Pattern(regexp = ".*?emmanuel.*?") // emmanuel
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface EmmanuelsEmail {

    String message() default "Not emmanuel's email";

    @OverridesAttribute(constraint = Pattern.class, name = "message", constraintIndex = 0)
    String emailMessage() default "Not an email";

    @OverridesAttribute(constraint = Pattern.class, name = "message", constraintIndex = 1)
    String emmanuelMessage() default "Not Emmanuel";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        EmmanuelsEmail[] value();
    }
}
```

`@OverridesAttribute` definition is as follows:

Listing 3.5: `@OverridesAttribute` *annotation*

```
/**
 * Marks an attribute as overriding the attribute of a composing constraint.
 * Both attributes must share the same type.
 *
 * @author Emmanuel Bernard
 */
@Documented
@Retention(RUNTIME)
@Target({ METHOD })
@Repeatable(List.class)
public @interface OverridesAttribute {

    /**
     * @return constraint type the attribute is overriding
     */
    Class<? extends Annotation> constraint();

    /**
     * Name of the Constraint attribute overridden.
     * Defaults to the name of the attribute hosting {@code @OverridesAttribute}.
     *
     * @return name of constraint attribute overridden
     */
    String name() default "";

    /**
     * The index of the targeted constraint declaration when using
     * multiple constraints of the same type.
     * <p>
     * The index represents the index of the constraint in the
     * {@code value()} array.
     * <p>
     * By default, no index is defined and the single constraint declaration
     * is targeted.
     *
     * @return constraint declaration index if multivalued annotation is used
     */
    int constraintIndex() default -1;

    /**
     * Defines several {@link OverridesAttribute} annotations on the same element
     *
     * @see jakarta.validation.OverridesAttribute
     */
    @Documented
    @Target({ METHOD })
    @Retention(RUNTIME)
    public @interface List {

        OverridesAttribute[] value();
    }
}
```

The following elements uniquely identify an overridden constraint attribute:

- `@OverridesAttribute.constraint`
- `@OverridesAttribute.name`
- `@OverridesAttribute.constraintIndex`

If the composition is invalid, e.g.

- infinitely recursive composition
- wrong attribute overriding
- a single attribute mapped to more than one source attribute
- a composing and composed constraint marked as different constraint types (i.e., generic and cross-parameter)

- etc.

a `ConstraintDefinitionException` is raised either at validation time or when the metadata is requested.

Constraint designers are encouraged to make use of composition (recursively or not) based on the built-in constraints defined by the specification. The composing constraints are exposed through the Jakarta Validation metadata API (ConstraintDescriptor). This metadata is particularly useful for third-party metadata consumers like persistence frameworks generating database schemas (such as Jakarta Persistence) or presentation frameworks.

3.4. Constraint validation implementation

A constraint validation implementation performs the validation of a given constraint annotation for a given type. The implementation classes are specified by the `validatedBy` element of the `@Constraint` annotation that decorates the constraint definition. The constraint validation implementation implements the `ConstraintValidator` interface.

Listing 3.6: `ConstraintValidator` *interface*

```
/**
 * Defines the logic to validate a given constraint {@code A}
 * for a given object type {@code T}.
 * <p>
 * Implementations must comply to the following restriction:
 * <ul>
 *     <li>{@code T} must resolve to a non parameterized type</li>
 *     <li>or generic parameters of {@code T} must be unbounded
 *     wildcard types</li>
 * </ul>
 * <p>
 * The annotation {@link SupportedValidationTarget} can be put on a
 * {@code ConstraintValidator} implementation to mark it as supporting
 * cross-parameter constraints. Check out {@link SupportedValidationTarget}
 * and {@link Constraint} for more information.
 *
 * @param <A> the annotation type handled by an implementation
 * @param <T> the target type supported by an implementation
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConstraintValidator<A extends Annotation, T> {

    /**
     * Initializes the validator in preparation for
     * {@link #isValid(Object, ConstraintValidatorContext)} calls.
     * The constraint annotation for a given constraint declaration
     * is passed.
     * <p>
     * This method is guaranteed to be called before any use of this instance for
     * validation.
     * <p>
     * The default implementation is a no-op.
     *
     * @param constraintAnnotation annotation instance for a given constraint declaration
     */
    default void initialize(A constraintAnnotation) {
    }

    /**
     * Implements the validation logic.
     * The state of {@code value} must not be altered.
     * <p>
     * This method can be accessed concurrently, thread-safety must be ensured
     * by the implementation.
     *
     * @param value object to validate
     * @param context context in which the constraint is evaluated
     *
     * @return {@code false} if {@code value} does not pass the constraint
     */
```

```
    boolean isValid(T value, ConstraintValidatorContext context);
}
```

Some restrictions apply on the generic type `T` (used in the `isValid()` method). `T` must

- resolve to a non parameterized type (i.e. because the type is not using generics or because the raw type is used instead of the generic version)
- or generic parameters of `T` must be unbounded wildcard types (i.e. `<?>`).

NOTE This restriction is not a theoretical limitation and a future version of the specification might allow it.

By default, a `ConstraintValidator` targets the (returned) element annotated by the constraint. You can make a `ConstraintValidator` target the array of parameters of a method or constructor (aka cross-parameter) by annotating the validator implementation with `@SupportedValidationTarget`.

Listing 3.7: `@SupportedValidationTarget` *annotation and* `ValidationTarget` *enum*

```
package jakarta.validation.constraintvalidation;


/**
 * Defines the target(s) a {@link ConstraintValidator} can validate.
 * <p>
 * A {@code ConstraintValidator} can target the (returned) element
 * annotated by the constraint, the array of parameters of a method
 * or constructor (aka cross-parameter) or both.
 * <p>
 * If {@code @SupportedValidationTarget} is not present, the
 * {@code ConstraintValidator} targets the (returned) element annotated
 * by the constraint.
 * <p>
 * A {@code ConstraintValidator} targeting cross-parameter must accept
 * {@code Object[]} (or {@code Object}) as the type of object it validates.
 *
 * @author Emmanuel Bernard
 * @since 1.1
 */
@Documented
@Target({ TYPE })
@Retention(RUNTIME)
public @interface SupportedValidationTarget {

    ValidationTarget[] value();
}


package jakarta.validation.constraintvalidation;

/**
 * List of possible targets for a {@link ConstraintValidator}.
 *
 * @author Emmanuel Bernard
 * @since 1.1
 */
public enum ValidationTarget {

    /**
     * (Returned) element annotated by the constraint.
     */
    ANNOTATED_ELEMENT,

    /**
     * Array of parameters of the annotated method or constructor (aka cross-parameter).
     */
    PARAMETERS
}
```

A `ConstraintValidator` implementation can target both annotated elements and array of parameters.

If a `ConstraintValidator` targets array of parameters (cross-parameter), `T` must resolve to `Object[]` (or `Object`) in order to have the array of parameter values passed to the `isValid()` method. A `ConstraintDefinitionException` is raised otherwise.

Example 3.15: Example of cross parameter ConstraintValidator

```
@SupportedValidationTarget(ValidationTarget.PARAMETERS)
public class ScriptAssertValidator implements ConstraintValidator<ScriptAssert,Object[]> {
    @Override
    public void initialize(ScriptAssert constraintAnnotation) {
        [...]
    }

    @Override
    public boolean isValid(Object[] value, ConstraintValidatorContext context) {
        [...]
    }
}
```

Valid ConstraintValidator definitions shows some examples of valid definitions.

Example 3.16: Valid ConstraintValidator definitions

```
//String is not making use of generics
public class SizeValidatorForString implements ConstraintValidator<Size, String> {
    [...]
}

//Collection uses generics but the raw type is used
public class SizeValidatorForCollection implements ConstraintValidator<Size, Collection> {
    [...]
}

//Collection uses generics and unbounded wildcard type
public class SizeValidatorForCollection implements ConstraintValidator<Size, Collection<?>> {
    [...]
}

//Validator for cross-parameter constraint
@SupportedValidationTarget(ValidationTarget.PARAMETERS)
public class DateParametersConsistentValidator
    implements ConstraintValidator<DateParametersConsistent, Object[]> {
    [...]
}

//Validator for both annotated elements and executable parameters
@SupportedValidationTarget({ValidationTarget.ANNOTATED_ELEMENT, ValidationTarget.PARAMETERS})
public class ELScriptValidator implements ConstraintValidator<ELScript, Object> {
    [...]
}
```

And some invalid definitions in Invalid ConstraintValidator definitions.

Example 3.17: Invalid ConstraintValidator definitions

```
//parameterized type
public class SizeValidatorForString implements ConstraintValidator<Size, Collection<String>> {
    [...]
}

//parameterized type using bounded wildcard
public class SizeValidatorForCollection implements ConstraintValidator<Size, Collection<? extends Address>> {
    [...]
}
```

```
//cross-parameter validator accepting the wrong type
@SupportedValidationTarget(ValidationTarget.PARAMETERS)
public class NumberPositiveValidator implements ConstraintValidator<NumberPositive, Number> {
    [...]
}
```

The lifecycle of a constraint validation implementation instance is undefined. Jakarta Validation providers are allowed to cache `ConstraintValidator` instances retrieved from the `ConstraintValidatorFactory`.

The `initialize()` method is called by the Jakarta Validation provider prior to any use of the constraint implementation. As of Jakarta Validation 2.0, `initialize()` has an empty default implementation, allowing to omit the implementation from validators that don't need to access any constraint attributes.

The `isValid()` method is evaluated by the Jakarta Validation provider each time a given value is validated. It returns `false` if the value is not valid, `true` otherwise. `isValid()` implementations must be thread-safe.

If the property is of an unanticipated type, an `UnexpectedTypeException` is raised. `ConstraintValidator` implementations raise this exception themselves if they receive an unsupported type. However, constraint designers are encouraged to make use of specialized `ConstraintValidator` implementations and delegate the type resolution to the Jakarta Validation provider (see the type matching algorithm described in ConstraintValidator resolution algorithm).

If an exception occurs either in the `initialize()` or `isValid()` method, the runtime exception is wrapped into a `ValidationException` by the Jakarta Validation engine.

The constraint validation implementation is not allowed to change the state of the value passed to `isValid()`.

NOTE

> While not mandatory, it is considered a good practice to split the core constraint validation from the not null constraint validation (for example, an `@Email` constraint will return `true` on a null object, i.e. will not take care of the `@NotNull` validation).
>
> `null` can have multiple meanings but is commonly used to express that a value does not make sense, is not available or is simply unknown. Those constraints on the value are orthogonal in most cases to other constraints. For example a String, if present, must be an email but can be null. Separating both concerns is a good practice.

The `ConstraintValidatorContext` object passed to the `isValid()` method carries information and operations available in the context the constraint is validated to.

Listing 3.8: `ConstraintValidatorContext` *interface*

```
/**
 * Provides contextual data and operation when applying a given constraint validator.
 *
 * At least one {@link ConstraintViolation} must be defined (either the default one,
 * of if the default {@code ConstraintViolation} is disabled, a custom one).
 *
 * @author Emmanuel Bernard
 * @author Guillaume Smet
 */
public interface ConstraintValidatorContext {

    /**
     * Disables the default {@link ConstraintViolation} object generation (which
     * is using the message template declared on the constraint).
     * <p>
     * Useful to set a different violation message or generate a {@code ConstraintViolation}
     * based on a different property.
     */
    void disableDefaultConstraintViolation();

    /**
     * @return the current un-interpolated default message
     */
    String getDefaultConstraintMessageTemplate();

    /**
     * Returns the provider for obtaining the current time in the form of a {@link Clock},
     * e.g. when validating the {@code Future} and {@code Past} constraints.
```

```
     *
     * @return the provider for obtaining the current time, never {@code null}. If no
     * specific provider has been configured during bootstrap, a default implementation using
     * the current system time and the current default time zone as returned by
     * {@link Clock#systemDefaultZone()} will be returned.
     *
     * @since 2.0
     */
    ClockProvider getClockProvider();

    /**
     * Returns a constraint violation builder building a violation report
     * allowing to optionally associate it to a sub path.
     * The violation message will be interpolated.
     * <p>
     * To create the {@link ConstraintViolation}, one must call either one of
     * the {@code addConstraintViolation()} methods available in one of the
     * interfaces of the fluent API.
     * If another method is called after {@code addConstraintViolation()} on
     * {@code ConstraintViolationBuilder} or any of its associated nested interfaces
     * an {@code IllegalStateException} is raised.
     * <p>
     * If {@link ConstraintValidator#isValid(Object, ConstraintValidatorContext)} returns
     * {@code false}, a {@code ConstraintViolation} object will be built per constraint
     * violation report including the default one (unless
     * {@link #disableDefaultConstraintViolation()} has been called).
     * <p>
     * {@code ConstraintViolation} objects generated from such a call
     * contain the same contextual information (root bean, path and so on) unless
     * the path has been overridden.
     * <p>
     * To create a different {@code ConstraintViolation}, a new constraint violation builder
     * has to be retrieved from {@code ConstraintValidatorContext}
     *
     * Here are a few usage examples:
     * <pre>
     * //assuming the following domain model
     * public class User {
     *     public Map&lt;String,Address&gt; getAddresses() { ... }
     * }
     *
     * public class Address {
     *     public String getStreet() { ... }
     *     public Country getCountry() { ... }
     * }
     *
     * public class Country {
     *     public String getName() { ... }
     * }
     *
     * //From a property-level constraint on User.addresses
     * //Build a constraint violation on the default path - i.e. the "addresses" property
     * context.buildConstraintViolationWithTemplate( "this detail is wrong" )
     *             .addConstraintViolation();
     *
     * //From a class level constraint on Address
     * //Build a constraint violation on the default path + "street"
     * //i.e. the street property of Address
     * context.buildConstraintViolationWithTemplate( "this detail is wrong" )
     *             .addPropertyNode( "street" )
     *             .addConstraintViolation();
     *
     * //From a property-level constraint on  User.addresses
     * //Build a constraint violation on the default path + the bean stored
     * //under the "home" key in the map
     * context.buildConstraintViolationWithTemplate( "Incorrect home address" )
     *             .addBeanNode()
     *                 .inContainer( Map.class, 1 )
     *                 .inIterable().atKey( "home" )
     *             .addConstraintViolation();
```

```
 *
 * //From a class level constraint on User
 * //Build a constraint violation on the default path + addresses["home"].country.name
 * //i.e. property "country.name" on the object stored under "home" in the map
 * context.buildConstraintViolationWithTemplate( "this detail is wrong" )
 *             .addPropertyNode( "addresses" )
 *             .addPropertyNode( "country" )
 *                 .inContainer( Map.class, 1 )
 *                 .inIterable().atKey( "home" )
 *             .addPropertyNode( "name" )
 *             .addConstraintViolation();
 *
 * //From a class level constraint on User
 * //Build a constraint violation on the default path + addresses["home"].&lt;map key&gt;
 * //i.e. a container element constraint violation for the map key
 * context.buildConstraintViolationWithTemplate( "the map key is invalid" )
 *             .addPropertyNode( "addresses" )
 *             .addContainerElementNode( "&lt;map key&gt;", Map.class, 0 )
 *                 .inIterable().atKey( "invalid" )
 *             .addConstraintViolation();
 * </pre>
 * <p>
 * Cross-parameter constraints on a method can create a node specific
 * to a particular parameter if required. Let's explore a few examples:
 * <pre>
 * //Cross-parameter constraint on method
 * //createUser(String password, String passwordRepeat)
 * //Build a constraint violation on the default path + "passwordRepeat"
 * context.buildConstraintViolationWithTemplate("Passwords do not match")
 *             .addParameterNode(1)
 *             .addConstraintViolation();
 *
 * //Cross-parameter constraint on a method
 * //mergeAddresses(Map&lt;String,Address&gt; addresses,
 * //        Map&lt;String,Address&gt; otherAddresses)
 * //Build a constraint violation on the default path + "otherAddresses["home"]
 * //i.e. the Address bean hosted in the "home" key of the "otherAddresses" map parameter
 * context.buildConstraintViolationWithTemplate(
 *         "Map entry home present in both and does not match")
 *             .addParameterNode(1)
 *             .addBeanNode()
 *                 .inContainer( Map.class, 1 )
 *                 .inIterable().atKey("home")
 *             .addConstraintViolation();
 *
 * //Cross-parameter constraint on a method
 * //mergeAddresses(Map&lt;String,Address&gt; addresses,
 * //        Map&lt;String,Address&gt; otherAddresses)
 * //Build a constraint violation on the default path + "otherAddresses["home"].city
 * //i.e. on the "city" property of the Address bean hosted in
 * //the "home" key of the "otherAddresses" map
 * context.buildConstraintViolationWithTemplate(
 *         "Map entry home present in both but city does not match")
 *             .addParameterNode(1)
 *             .addPropertyNode("city")
 *                 .inContainer( Map.class, 1 )
 *                 .inIterable().atKey("home")
 *             .addConstraintViolation();
 * </pre>
 *
 * @param messageTemplate new un-interpolated constraint message
 * @return returns a constraint violation builder
 */
ConstraintViolationBuilder buildConstraintViolationWithTemplate(String messageTemplate);

/**
 * Returns an instance of the specified type allowing access to
 * provider-specific APIs. If the Jakarta Validation provider
 * implementation does not support the specified class,
 * {@link ValidationException} is thrown.
```

```
     *
     * @param type the class of the object to be returned
     * @param <T> the type of the object to be returned
     * @return an instance of the specified class
     * @throws ValidationException if the provider does not support the call
     *
     * @since 1.1
     */
    <T> T unwrap(Class<T> type);

    /**
     * {@link ConstraintViolation} builder allowing to optionally associate
     * the violation report to a sub path.
     * <p>
     * To create the {@code ConstraintViolation}, one must call either one of
     * the {@code addConstraintViolation()} methods available in one of the
     * interfaces of the fluent API.
     * <p>
     * If another method is called after {@code addConstraintViolation()} on
     * {@code ConstraintViolationBuilder} or any of its associated objects
     * an {@code IllegalStateException} is raised.
     */
    interface ConstraintViolationBuilder {

        /**
         * Adds a node to the path the {@link ConstraintViolation} will be associated to.
         * <p>
         * {@code name} describes a single property. In particular,
         * dot (.) is not allowed.
         *
         * @param name property name
         * @return a builder representing node {@code name}
         * @deprecated replaced by {@link #addPropertyNode(String)},
         *             {@link #addBeanNode()} and {@link #addParameterNode(int)}
         */
        @Deprecated(since = "1.1", forRemoval = true)
        NodeBuilderDefinedContext addNode(String name);

        /**
         * Adds a property node to the path the {@link ConstraintViolation}
         * will be associated to.
         * <p>
         * {@code name} describes a single property. In particular,
         * dot (.) is not allowed.
         *
         * @param name property name
         * @return a builder representing node {@code name}
         * @throws IllegalArgumentException if the name is null
         *
         * @since 1.1
         */
        NodeBuilderCustomizableContext addPropertyNode(String name);

        /**
         * Adds a bean node (class-level) to the path the {@link ConstraintViolation}
         * will be associated to.
         * Note that bean nodes are always leaf nodes.
         *
         * @return a builder representing the bean node
         *
         * @since 1.1
         */
        LeafNodeBuilderCustomizableContext addBeanNode();

        /**
         * Adds a container element node to the path the {@link ConstraintViolation}
         * will be associated to.
         *
         * @param name the node name
         * @param containerType the type of the container
```

```
     * @param typeArgumentIndex the index of the type argument
     * @return a builder representing the container element node
     * @throws IllegalArgumentException if the index is not valid
     *
     * @since 2.0
     */
    ContainerElementNodeBuilderCustomizableContext addContainerElementNode(String name,
            Class<?> containerType, Integer typeArgumentIndex);

    /**
     * Adds a method parameter node to the path the {@link ConstraintViolation}
     * will be associated to.
     * The parameter index must be valid (i.e. within the boundaries of the method
     * parameter indexes). May only be called from within cross-parameter validators.
     *
     * @param index the parameter index
     * @return a builder representing the index-th parameter node
     * @throws IllegalArgumentException if the index is not valid
     *
     * @since 1.1
     */
    NodeBuilderDefinedContext addParameterNode(int index);

    /**
     * Adds the new {@link ConstraintViolation} to be generated if the
     * constraint validator marks the value as invalid.
     * <p>
     * Methods of this {@code ConstraintViolationBuilder} instance and its nested
     * objects throw {@code IllegalStateException} from now on.
     *
     * @return the {@code ConstraintValidatorContext} instance the
     *         {@code ConstraintViolationBuilder} comes from
     */
    ConstraintValidatorContext addConstraintViolation();

    /**
     * Represents a node whose context is known
     * (i.e. index, key and isInIterable)
     * and that is a leaf node (i.e. no subnode can be added).
     *
     * @since 1.1
     */
    interface LeafNodeBuilderDefinedContext {

        /**
         * Adds the new {@link ConstraintViolation} to be generated if the
         * constraint validator marks the value as invalid.
         * <p>
         * Methods of the {@code ConstraintViolationBuilder} instance this object
         * comes from and the constraint violation builder nested
         * objects throw {@code IllegalStateException} after this call.
         *
         * @return {@code ConstraintValidatorContext} instance the
         *         {@code ConstraintViolationBuilder} comes from
         */
        ConstraintValidatorContext addConstraintViolation();
    }

    /**
     * Represents a node whose context is
     * configurable (i.e. index, key and isInIterable)
     * and that is a leaf node (i.e. no subnode can be added).
     *
     * @since 1.1
     */
    interface LeafNodeBuilderCustomizableContext {

        /**
         * Marks the node as being in an iterable, e.g. array, {@code Iterable} or a
         * {@code Map}.
```

```
         *
         * @return a builder representing iterable details
         */
        LeafNodeContextBuilder inIterable();

        /**
         * Marks the node as being in a container such as a {@code List}, {@code Map} or
         * {@code Optional}.
         *
         * @param containerClass the type of the container
         * @param typeArgumentIndex type index of the concerned type argument
         * @return a builder representing the current node
         * @throws IllegalArgumentException if the index is not valid
         *
         * @since 2.0
         */
        LeafNodeBuilderCustomizableContext inContainer(Class<?> containerClass,
                                                       Integer typeArgumentIndex);

        /**
         * Adds the new {@link ConstraintViolation} to be generated if the
         * constraint validator mark the value as invalid.
         * <p>
         * Methods of the {@code ConstraintViolationBuilder} instance this object
         * comes from and the constraint violation builder nested
         * objects throw {@code IllegalStateException} after this call.
         *
         * @return {@code ConstraintValidatorContext} instance the
         *         {@code ConstraintViolationBuilder} comes from
         */
        ConstraintValidatorContext addConstraintViolation();
    }

    /**
     * Represents refinement choices for a node which is
     * in an iterable, e.g. array, {@code Iterable} or {@code Map}.
     * <p>
     * If the iterable is an indexed collection or a map,
     * the index or the key should be set.
     * <p>
     * The node is a leaf node (i.e. no subnode can be added).
     *
     * @since 1.1
     */
    interface LeafNodeContextBuilder {

        /**
         * Defines the key the object is into the {@code Map}.
         *
         * @param key map key
         * @return a builder representing the current node
         */
        LeafNodeBuilderDefinedContext atKey(Object key);

        /**
         * Defines the index the object is into the {@code List} or array
         *
         * @param index index
         * @return a builder representing the current node
         */
        LeafNodeBuilderDefinedContext atIndex(Integer index);

        /**
         * Adds the new {@link ConstraintViolation} to be generated if the
         * constraint validator mark the value as invalid.
         * <p>
         * Methods of the {@code ConstraintViolationBuilder} instance this object
         * comes from and the constraint violation builder nested
         * objects throw {@code IllegalStateException} after this call.
         *
```

```
     * @return {@code ConstraintValidatorContext} instance the
     *           {@code ConstraintViolationBuilder} comes from
     */
    ConstraintValidatorContext addConstraintViolation();
}

/**
 * Represents a node whose context is known
 * (i.e. index, key and isInIterable)
 * and that is not necessarily a leaf node (i.e. subnodes can
 * be added).
 */
interface NodeBuilderDefinedContext {

    /**
     * Adds a node to the path the {@link ConstraintViolation} will be associated to.
     * <p>
     * {@code name} describes a single property. In particular,
     * dot (.) is not allowed.
     *
     * @param name property name
     * @return a builder representing node {@code name}
     * @deprecated replaced by {@link #addPropertyNode(String)}
     *             and {@link #addBeanNode()}
     */
    @Deprecated(since = "1.1", forRemoval = true)
    NodeBuilderCustomizableContext addNode(String name);

    /**
     * Adds a property node to the path the {@link ConstraintViolation}
     * will be associated to.
     * <p>
     * {@code name} describes a single property. In particular,
     * dot (.) is not allowed.
     *
     * @param name property name
     * @return a builder representing node {@code name}
     * @throws IllegalArgumentException if the name is null
     *
     * @since 1.1
     */
    NodeBuilderCustomizableContext addPropertyNode(String name);

    /**
     * Adds a bean node (class-level) to the path the {@link ConstraintViolation}
     * will be associated to.
     * Note that bean nodes are always leaf nodes.
     *
     * @return a builder representing the bean node
     *
     * @since 1.1
     */
    LeafNodeBuilderCustomizableContext addBeanNode();

    /**
     * Adds a container element node to the path the {@link ConstraintViolation}
     * will be associated to.
     *
     * @param name the node name
     * @param containerType the type of the container
     * @param typeArgumentIndex the index of the type argument
     * @return a builder representing the container element node
     * @throws IllegalArgumentException if the index is not valid
     *
     * @since 2.0
     */
    ContainerElementNodeBuilderCustomizableContext addContainerElementNode(
            String name, Class<?> containerType, Integer typeArgumentIndex);

    /**
```

```
     * Adds the new {@link ConstraintViolation} to be generated if the
     * constraint validator marks the value as invalid.
     * <p>
     * Methods of the {@code ConstraintViolationBuilder} instance this object
     * comes from and the constraint violation builder nested
     * objects throw {@code IllegalStateException} after this call.
     *
     * @return {@code ConstraintValidatorContext} instance the
     *           {@code ConstraintViolationBuilder} comes from
     */
    ConstraintValidatorContext addConstraintViolation();
}

/**
 * Represents a node whose context is
 * configurable (i.e. index, key and isInIterable)
 * and that is not necessarily a leaf node (i.e. subnodes can
 * be added).
 */
interface NodeBuilderCustomizableContext {

    /**
     * Marks the node as being in an iterable, e.g. array, {@code Iterable} or a
     * {@code Map}.
     *
     * @return a builder representing iterable details
     */
    NodeContextBuilder inIterable();

    /**
     * Marks the node as being in a container such as a {@code List}, {@code Map} or
     * {@code Optional}.
     *
     * @param containerClass the type of the container
     * @param typeArgumentIndex type index of the concerned type argument
     * @return a builder representing the current node
     * @throws IllegalArgumentException if the index is not valid
     *
     * @since 2.0
     */
    NodeBuilderCustomizableContext inContainer(Class<?> containerClass,
                                               Integer typeArgumentIndex);

    /**
     * Adds a node to the path the {@link ConstraintViolation} will be associated to.
     *
     * {@code name} describes a single property. In particular,
     * dot (.) is not allowed.
     *
     * @param name property name
     * @return a builder representing node {@code name}
     * @deprecated replaced by {@link #addPropertyNode(String)}
     *             and {@link #addBeanNode()}
     */
    @Deprecated(since = "1.1", forRemoval = true)
    NodeBuilderCustomizableContext addNode(String name);

    /**
     * Adds a property node to the path the {@link ConstraintViolation}
     * will be associated to.
     *
     * {@code name} describes a single property. In particular,
     * dot (.) is not allowed.
     *
     * @param name property name
     * @return a builder representing node {@code name}
     * @throws IllegalArgumentException if the name is null
     *
     * @since 1.1
     */
```

```
    NodeBuilderCustomizableContext addPropertyNode(String name);

    /**
     * Adds a bean node (class-level) to the path the {@link ConstraintViolation}
     * will be associated to.
     * Note that bean nodes are always leaf nodes.
     *
     * @return a builder representing the bean node
     *
     * @since 1.1
     */
    LeafNodeBuilderCustomizableContext addBeanNode();

    /**
     * Adds a container element node to the path the {@link ConstraintViolation}
     * will be associated to.
     *
     * @param name the node name
     * @param containerType the type of the container
     * @param typeArgumentIndex the index of the type argument
     * @return a builder representing the container element node
     * @throws IllegalArgumentException if the index is not valid
     *
     * @since 2.0
     */
    ContainerElementNodeBuilderCustomizableContext addContainerElementNode(
            String name, Class<?> containerType, Integer typeArgumentIndex);

    /**
     * Adds the new {@link ConstraintViolation} to be generated if the
     * constraint validator mark the value as invalid.
     * <p>
     * Methods of the {@code ConstraintViolationBuilder} instance this object
     * comes from and the constraint violation builder nested
     * objects throw {@code IllegalStateException} after this call.
     *
     * @return {@code ConstraintValidatorContext} instance the
     *           {@code ConstraintViolationBuilder} comes from
     */
    ConstraintValidatorContext addConstraintViolation();
}

/**
 * Represents refinement choices for a node which is
 * in an iterable, e.g. array, {@code Iterable} or {@code Map}.
 * <p>
 * If the iterable is an indexed collection or a map,
 * the index or the key should be set.
 * <p>
 * The node is not necessarily a leaf node (i.e. subnodes can
 * be added).
 */
interface NodeContextBuilder {

    /**
     * Defines the key the object is into the {@code Map}.
     *
     * @param key map key
     * @return a builder representing the current node
     */
    NodeBuilderDefinedContext atKey(Object key);

    /**
     * Defines the index the object is into the {@code List} or array.
     *
     * @param index index
     * @return a builder representing the current node
     */
    NodeBuilderDefinedContext atIndex(Integer index);
```

```
    /**
     * Adds a node to the path the {@code ConstraintViolation} will be associated to.
     *
     * {@code name} describes a single property. In particular,
     * dot (.) is not allowed.
     *
     * @param name property name
     * @return a builder representing node {@code name}
     * @deprecated replaced by {@link #addPropertyNode(String)}
     *             and {@link #addBeanNode()}
     */
    @Deprecated(since = "1.1", forRemoval = true)
    NodeBuilderCustomizableContext addNode(String name);

    /**
     * Adds a property node to the path the {@link ConstraintViolation}
     * will be associated to.
     *
     * {@code name} describes a single property. In particular,
     * dot (.) is not allowed.
     *
     * @param name property name
     * @return a builder representing node {@code name}
     * @throws IllegalArgumentException if the name is null
     *
     * @since 1.1
     */
    NodeBuilderCustomizableContext addPropertyNode(String name);

    /**
     * Adds a bean node (class-level) to the path the {@link ConstraintViolation}
     * will be associated to.
     * <p>
     * Note that bean nodes are always leaf nodes.
     *
     * @return a builder representing the bean node
     *
     * @since 1.1
     */
    LeafNodeBuilderCustomizableContext addBeanNode();

    /**
     * Adds a container element node to the path the {@link ConstraintViolation}
     * will be associated to.
     *
     * @param name the node name
     * @param containerType the type of the container
     * @param typeArgumentIndex the index of the type argument
     * @return a builder representing the container element node
     * @throws IllegalArgumentException if the index is not valid
     *
     * @since 2.0
     */
    ContainerElementNodeBuilderCustomizableContext addContainerElementNode(
            String name, Class<?> containerType, Integer typeArgumentIndex);

    /**
     * Adds the new {@link ConstraintViolation} to be generated if the
     * constraint validator mark the value as invalid.
     * <p>
     * Methods of the {@code ConstraintViolationBuilder} instance this object
     * comes from and the constraint violation builder nested
     * objects throw {@code IllegalStateException} after this call.
     *
     * @return {@code ConstraintValidatorContext} instance the
     *         {@code ConstraintViolationBuilder} comes from
     */
    ConstraintValidatorContext addConstraintViolation();
}
```

```
/**
 * Represents a container element node whose context is known
 * (i.e. index, key and isInIterable)
 * and that is not necessarily a leaf node (i.e. subnodes can
 * be added).
 *
 * @since 2.0
 */
interface ContainerElementNodeBuilderDefinedContext {

    /**
     * Adds a property node to the path the {@link ConstraintViolation}
     * will be associated to.
     * <p>
     * {@code name} describes a single property. In particular,
     * dot (.) is not allowed.
     *
     * @param name property name
     * @return a builder representing node {@code name}
     * @throws IllegalArgumentException if the name is null
     */
    NodeBuilderCustomizableContext addPropertyNode(String name);

    /**
     * Adds a bean node (class-level) to the path the {@link ConstraintViolation}
     * will be associated to.
     * Note that bean nodes are always leaf nodes.
     *
     * @return a builder representing the bean node
     */
    LeafNodeBuilderCustomizableContext addBeanNode();

    /**
     * Adds a container element node to the path the {@link ConstraintViolation}
     * will be associated to.
     *
     * @param name the node name
     * @param containerType the type of the container
     * @param typeArgumentIndex the index of the type argument
     * @return a builder representing the container element node
     * @throws IllegalArgumentException if the index is not valid
     */
    ContainerElementNodeBuilderCustomizableContext addContainerElementNode(
            String name, Class<?> containerType, Integer typeArgumentIndex);

    /**
     * Adds the new {@link ConstraintViolation} to be generated if the
     * constraint validator marks the value as invalid.
     * <p>
     * Methods of the {@code ConstraintViolationBuilder} instance this object
     * comes from and the constraint violation builder nested
     * objects throw {@code IllegalStateException} after this call.
     *
     * @return {@code ConstraintValidatorContext} instance the
     *           {@code ConstraintViolationBuilder} comes from
     */
    ConstraintValidatorContext addConstraintViolation();
}

/**
 * Represents a container element node whose context is
 * configurable (i.e. index, key and isInIterable)
 * and that is not necessarily a leaf node (i.e. subnodes can
 * be added).
 *
 * @since 2.0
 */
interface ContainerElementNodeBuilderCustomizableContext {

    /**
```

```
     * Marks the node as being in an iterable, e.g. array, {@code Iterable} or a
     * {@code Map}.
     *
     * @return a builder representing iterable details
     */
    ContainerElementNodeContextBuilder inIterable();

    /**
     * Adds a property node to the path the {@link ConstraintViolation}
     * will be associated to.
     *
     * {@code name} describes a single property. In particular,
     * dot (.) is not allowed.
     *
     * @param name property name
     * @return a builder representing node {@code name}
     * @throws IllegalArgumentException if the name is null
     */
    NodeBuilderCustomizableContext addPropertyNode(String name);

    /**
     * Adds a bean node (class-level) to the path the {@link ConstraintViolation}
     * will be associated to.
     * <p>
     * Note that bean nodes are always leaf nodes.
     *
     * @return a builder representing the bean node
     */
    LeafNodeBuilderCustomizableContext addBeanNode();

    /**
     * Adds a container element node to the path the {@link ConstraintViolation}
     * will be associated to.
     *
     * @param name the node name
     * @param containerType the type of the container
     * @param typeArgumentIndex the index of the type argument
     * @return a builder representing the container element node
     * @throws IllegalArgumentException if the index is not valid
     */
    ContainerElementNodeBuilderCustomizableContext addContainerElementNode(
            String name, Class<?> containerType, Integer typeArgumentIndex);

    /**
     * Adds the new {@link ConstraintViolation} to be generated if the
     * constraint validator mark the value as invalid.
     * <p>
     * Methods of the {@code ConstraintViolationBuilder} instance this object
     * comes from and the constraint violation builder nested
     * objects throw {@code IllegalStateException} after this call.
     *
     * @return {@code ConstraintValidatorContext} instance the
     *         {@code ConstraintViolationBuilder} comes from
     */
    ConstraintValidatorContext addConstraintViolation();
}

/**
 * Represents refinement choices for a container element node.
 * <p>
 * If the container is an indexed collection or a map,
 * the index or the key should be set.
 * <p>
 * The node is not necessarily a leaf node (i.e. subnodes can
 * be added).
 *
 * @since 2.0
 */
interface ContainerElementNodeContextBuilder {
```

```
            /**
             * Defines the key the object is into the {@code Map}.
             *
             * @param key map key
             * @return a builder representing the current node
             */
            ContainerElementNodeBuilderDefinedContext atKey(Object key);

            /**
             * Defines the index the object is into the {@code List} or array.
             *
             * @param index index
             * @return a builder representing the current node
             */
            ContainerElementNodeBuilderDefinedContext atIndex(Integer index);

            /**
             * Adds a property node to the path the {@link ConstraintViolation}
             * will be associated to.
             *
             * {@code name} describes a single property. In particular,
             * dot (.) is not allowed.
             *
             * @param name property name
             * @return a builder representing node {@code name}
             * @throws IllegalArgumentException if the name is null
             */
            NodeBuilderCustomizableContext addPropertyNode(String name);

            /**
             * Adds a bean node (class-level) to the path the {@link ConstraintViolation}
             * will be associated to.
             * <p>
             * Note that bean nodes are always leaf nodes.
             *
             * @return a builder representing the bean node
             */
            LeafNodeBuilderCustomizableContext addBeanNode();

            /**
             * Adds a container element node to the path the {@link ConstraintViolation}
             * will be associated to.
             *
             * @param name the node name
             * @param containerType the type of the container
             * @param typeArgumentIndex the index of the type argument
             * @return a builder representing the container element node
             * @throws IllegalArgumentException if the index is not valid
             */
            ContainerElementNodeBuilderCustomizableContext addContainerElementNode(
                    String name, Class<?> containerType, Integer typeArgumentIndex);

            /**
             * Adds the new {@link ConstraintViolation} to be generated if the
             * constraint validator mark the value as invalid.
             * <p>
             * Methods of the {@code ConstraintViolationBuilder} instance this object
             * comes from and the constraint violation builder nested
             * objects throw {@code IllegalStateException} after this call.
             *
             * @return {@code ConstraintValidatorContext} instance the
             *         {@code ConstraintViolationBuilder} comes from
             */
            ConstraintValidatorContext addConstraintViolation();
        }
    }
}
```

The `ConstraintValidatorContext` interface provides access to contextual information useful for the validation of specific constraints (e.g.

`getClockProvider()`, see Implementation of temporal constraint validators).

It also allows redefinition of the default constraint message generated when a constraint is not valid. By default, each invalid constraint leads to the generation of one error object represented by a `ConstraintViolation` object. This object is built from the default constraint message template as defined by the constraint declaration and the context in which the constraint declaration is placed (bean, property, executable parameter, cross-parameter, executable return value or container element).

The `ConstraintValidatorContext` methods let the constraint implementation disable the default `ConstraintViolation` generation and create one or more custom ones. The non-interpolated message passed as a parameter is used to build the `ConstraintViolation` message (the message interpolation operation is applied to it).

By default, the `Path` exposed on the `ConstraintViolation` represents the path to the bean, property, parameter, cross-parameter, return value or container element hosting the constraint (see ConstraintViolation for more information). You can point it to a subpath of this default path by using the constraint violation builder fluent API.

Using the fluent API to build custom constraint violations shows a few examples.

Example 3.18: Using the fluent API to build custom constraint violations

```
//assuming the following domain model
public class User {
    public Map<String,Address> getAddresses() { [...] }
}

public class Address {
    public String getStreet() { [...] }
    public Country getCountry() { [...] }
}

public class Country {
   public String getName() { [...] }
}

//From a property-level constraint on User.addresses
//Build a constraint violation on the default path - i.e. the "addresses" property
context.buildConstraintViolationWithTemplate( "this detail is wrong" )
            .addConstraintViolation();

//From a class level constraint on Address
//Build a constraint violation on the default path + "street"
//i.e. the street property of Address
context.buildConstraintViolationWithTemplate( "this detail is wrong" )
            .addPropertyNode( "street" )
            .addConstraintViolation();

//From a property-level constraint on  User.addresses
//Build a constraint violation on the default path + the bean stored
//under the "home" key in the map
context.buildConstraintViolationWithTemplate( "Incorrect home address" )
            .addBeanNode()
                .inContainer( Map.class, 1 )
                .inIterable().atKey( "home" )
            .addConstraintViolation();

//From a class level constraint on User
//Build a constraint violation on the default path + addresses["home"].country.name
//i.e. property "country.name" on the object stored under "home" in the map
context.buildConstraintViolationWithTemplate( "this detail is wrong" )
            .addPropertyNode( "addresses" )
            .addPropertyNode( "country" )
                .inContainer( Map.class, 1 )
                .inIterable().atKey( "home" )
            .addPropertyNode( "name" )
            .addConstraintViolation();

//From a class level constraint on User
//Build a constraint violation on the default path + addresses["home"].<map key>
//i.e. a container element constraint violation for the map key
context.buildConstraintViolationWithTemplate( "the map key is invalid" )
            .addPropertyNode( "addresses" )
```

```
            .addContainerElementNode( "<map key>", Map.class, 0 )
                .inIterable().atKey( "home" )
            .addConstraintViolation();

//To create a subnode representing a method parameter from a cross-parameter constraint violation

//Cross-parameter constraint on method createUser(String password, String passwordRepeat)
//Build a constraint violation on the default path + "passwordRepeat"
context.buildConstraintViolationWithTemplate("Passwords do not match")
            .addParameterNode( 1 )
            .addConstraintViolation();

//Cross-parameter constraint on a method
//mergeAddresses(Map<String,Address> addresses, Map<String,Address> otherAddresses)
//Build a constraint violation on the default path + "otherAddresses["home"]
//i.e. the Address bean hosted in the "home" key of the "otherAddresses" map parameter
context.buildConstraintViolationWithTemplate(
        "Map entry home present in both and does not match" )
            .addParameterNode( 1 )
            .addBeanNode()
                .inContainer( Map.class, 1 )
                .inIterable().atKey( "home" )
            .addConstraintViolation();

//Cross-parameter constraint on a method
//mergeAddresses(Map<String,Address> addresses, Map<String,Address> otherAddresses)
//Build a constraint violation on the default path + "otherAddresses["home"].city
//i.e. on the "city" property of the Address bean hosted in
//the "home" key of the "otherAddresses" map
context.buildConstraintViolationWithTemplate(
        "Map entry home present in both but city does not match" )
            .addParameterNode( 1 )
            .addPropertyNode( "city" )
                .inContainer( Map.class, 1 )
                .inIterable().atKey( "home" )
            .addConstraintViolation();
```

If `disableDefaultConstraintViolation()` is called, no custom error is added (using the error builder) and if the constraint is not valid, a `ValidationException` is raised.

3.4.1. Implementation of temporal constraint validators

Constraint validators for temporal constraints (either the built-in constraints `@Past`, `@PastOrPresent`, `@Future` and `@FutureOrPresent` or custom temporal constraints) can obtain the current instant from the `ClockProvider` object exposed by `ConstraintValidatorContext#getClockProvider()`.

Listing 3.9: `ClockProvider` *interface*

```
/**
 * Contract for obtaining the {@link Clock} used as the reference for {@code now} when
 * validating the {@code @Future} and {@code @Past} constraints.
 * <p>
 * The default implementation will return the current system time. Plugging in custom
 * implementations may be useful for instance in batch applications which need to run with a
 * specific logical date, e.g. with yesterday's date when re-running a failed batch job
 * execution.
 * <p>
 * Implementations must be safe for access from several threads at the same time.
 *
 * @author Gunnar Morling
 * @author Guillaume Smet
 * @since 2.0
 */
public interface ClockProvider {

    /**
     * Returns the clock which serves as the reference for {@code now}.
```

```
     *
     * @return the clock which serves as the reference for {@code now}; must not be
     * {@code null}
     */
    Clock getClock();
}
```

The `getClock()` method returns a `java.time.Clock` object which represents the current instant, date and time using a time zone. A conforming Jakarta Validation implementation provides a default clock provider which returns a clock representing the current system time and default time zone. It is recommended that implementations call `Clock#systemDefaultZone()` to obtain the clock.

When bootstrapping a validator factory or validator, an alternative clock provider can be registered (see Bootstrapping). This can for instance be useful for testing, for applying the time zone of the currently logged in user in a multi-user, multi time zone application or for running batch applications with a logical date and time different from the actual current date and time.

3.4.2. Examples

Example 3.19: ConstraintValidator implementation

```
/**
 * Check that a String begins with one of the given prefixes.
 */
public class BeginsWithValidator implements ConstraintValidator<BeginsWith, String> {

    private Set<String> allowedPrefixes;

    /**
     * Configure the constraint validator based on the elements specified at the time it was
     * defined.
     *
     * @param constraint the constraint definition
     */
    @Override
    public void initialize(BeginsWith constraint) {
        allowedPrefixes = Arrays.stream( constraint.value() )
                .collect( collectingAndThen( toSet(), Collections::unmodifiableSet ) );
    }

    /**
     * Validate a specified value. returns false if the specified value does not conform to
     * the definition.
     */
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if ( value == null )
            return true;

        return allowedPrefixes.stream()
                .anyMatch( value::startsWith );
    }
}
```

This `ConstraintValidator` checks that a String begins with one of the accepted prefixes. It also demonstrates an interesting best practice: return `true` on a null parameter.

The following listing shows a validator implementing the validation logic for a cross-parameter constraint.

Example 3.20: Cross-parameter validator implementation

```
/**
 * Check that two date parameters of a method are in the expected order. Expects the 2nd and
 * 3rd parameter of the validated method to be of type java.util.Date.
 */
@SupportedValidationTarget(ValidationTarget.PARAMETERS)
public class DateParametersConsistentValidator implements
```

```
        ConstraintValidator<DateParametersConsistent, Object[]> {

    /**
     * Validate a specified value. returns false if the specified value does not conform to
     * the definition
     */
    @Override
    public boolean isValid(Object[] value, ConstraintValidatorContext context) {
        if ( value.length != 3 ) {
            throw new IllegalArgumentException( "Unexpected method signature" );
        }
        // one or both limits are unbounded => always consistent
        if ( value[1] == null || value[2] == null ) {
            return true;
        }
        return ( (Date) value[1] ).before( (Date) value[2] );
    }
}
```

The following listing shows a validator implementing the validation logic for a constraint that is both generic and cross-parameter.

Example 3.21: Generic and cross-parameter validator implementation

```
/**
 * Checks that an object passes the Jakarta Expression Language expression
 * provided by the constraint.
 */
@SupportedValidationTarget({ValidationTarget.ANNOTATED_ELEMENT, ValidationTarget.PARAMETERS})
public class ELScriptValidator implements ConstraintValidator<ELScript, Object> {

    public void initialize(ELScript constraint) {
        [...]
    }

    public boolean isValid(Object value, ConstraintValidatorContext context) {
        [...]
    }
}
```

The next example shows how to use `ConstraintValidatorContext`.

Example 3.22: Use of ConstraintValidatorContext

```
/**
 * Check that a String begins with "SN-" and has a specified length.
 * <p>
 * Error messages are using either key:
 * <ul>
 *   <li>com.acme.constraint.SerialNumber.wrongprefix if the string doesn't begin with
 *   "SN-"</li>
 *   <li>com.acme.constraint.SerialNumber.wronglength if the string doesn't have the
 *   specified length</li>
 * </ul>
 */
public class SerialNumberValidator implements ConstraintValidator<SerialNumber, String> {

    private int length;

    /**
     * Configure the constraint validator based on the elements specified at the time it was
     * defined.
     *
     * @param constraint the constraint definition
     */
    @Override
    public void initialize(SerialNumber constraint) {
```

```
        this.length = constraint.length();
    }

    /**
     * Validate a specified value. returns false if the specified value does not conform to
     * the definition.
     */
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if ( value == null )
            return true;

        context.disableDefaultConstraintViolation();

        if ( !value.startsWith( "SN-" ) ) {
            String wrongPrefix = "{com.acme.constraint.SerialNumber.wrongprefix}";
            context.buildConstraintViolationWithTemplate( wrongPrefix )
                    .addConstraintViolation();
            return false;
        }
        if ( value.length() != length ) {
            String wrongLength = "{com.acme.constraint.SerialNumber.wronglength}";
            context.buildConstraintViolationWithTemplate( wrongLength )
                    .addConstraintViolation();
            return false;
        }
        return true;
    }
}
```

The default error message is disabled and replaced by a specific error message depending on the type of constraint violation detected. In this case, only one error report is returned at a given time but a constraint validation implementation can return several error reports.

The following example shows how to obtain the current date and time via the `ClockProvider` when validating a temporal constraint such as `@Past`:

Example 3.23: Validation of a temporal constraint

```
/**
 * Validates that the given {@link ZonedDateTime} is in the past.
 */
public class PastValidatorForZonedDateTime implements ConstraintValidator<Past, ZonedDateTime> {

    @Override
    public boolean isValid(ZonedDateTime value, ConstraintValidatorContext context) {
        if ( value == null ) {
            return true;
        }

        ZonedDateTime now = ZonedDateTime.now( context.getClockProvider().getClock() );

        return value.isBefore( now );
    }
}
```

3.5. The ConstraintValidatorFactory

Constraint validation implementation instances are created by a `ConstraintValidatorFactory`.

The lifecycle of `ConstraintValidator` instances is fully dependent of the Jakarta Validation provider and piloted by the `ConstraintValidatorFactory` methods. Therefore, `ConstraintValidatorFactory` implementations (such as dependency injection frameworks) must consider these instances as belonging to a dependent scope. Jakarta Validation providers must release each instance retrieved. The `ConstraintValidatorFactory` instance that has created a `ConstraintValidator` instance must be the one that releases it. In other words, passing an instance of `ConstraintValidator` to a `ConstraintValidatorFactory` that has not created it is an error.

NOTE `ConstraintValidator` instances created by the `ValidatorFactory` -level `ConstraintValidatorFactory` can be released when

> the `ValidatorFactory` is being closed.

Listing 3.10: `ConstraintValidatorFactory` *interface*

```
/**
 * Instantiates a {@link ConstraintValidator} instance based off its class.
 * The {@code ConstraintValidatorFactory} is <b>not</b> responsible
 * for calling {@link ConstraintValidator#initialize(java.lang.annotation.Annotation)}.
 *
 * @author Dhanji R. Prasanna
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConstraintValidatorFactory {

    /**
     * @param key The class of the constraint validator to instantiate
     * @param <T> The type of the constraint validator to instantiate
     *
     * @return A new constraint validator instance of the specified class
     */
    <T extends ConstraintValidator<?, ?>> T getInstance(Class<T> key);

    /**
     * Signals {@code ConstraintValidatorFactory} that the instance is no longer
     * being used by the Jakarta Validation provider.
     *
     * @param instance validator being released
     *
     * @since 1.1
     */
    void releaseInstance(ConstraintValidator<?, ?> instance);
}
```

The default `ConstraintValidatorFactory` provided by the Jakarta Validation provider implementation uses the public no-arg constructor of the `ConstraintValidator` class. A custom `ConstraintValidatorFactory` can be provided; for example it may benefit from dependency injection control in constraint implementations (see Bootstrapping considerations). Any constraint implementation relying on `ConstraintValidatorFactory` behaviors specific to an implementation (dependency injection, no no-arg constructor and so on) is not portable, hence great care should be given before walking that path. Make sure to configure the Jakarta Validation provider to honor any specific needs your `ConstraintValidator` has. As constraint designer and writer, make sure to document any specific non compliant requirements.

`ConstraintValidatorFactory` should not cache instances as the state of each instance can be altered in the `initialize()` method.

If an exception occurs in the factory while retrieving the `ConstraintValidator` instance, the runtime exception is wrapped in a `ValidationException`. If the instance returned by the factory is null, a `ValidationException` is raised.

4. Value extractor definition

Validation of container element constraints (see Container element constraints) as well as cascaded validation of generic container types (see Graph validation) requires access to the value(s) stored in the container. The retrieval of values stored in a container is handled via implementations of the `ValueExtractor` interface:

Listing 4.1: `ValueExtractor` *interface*

```
package jakarta.validation.valueextraction;

/**
 * Defines the logic used to extract the values from a container object of type {@code T}.
 * <p>
 * A value extractor for a generic type such as {@link Optional}, {@link List} or {@link Map}
 * is tied to one specific type parameter of {@code T}. The {@link ExtractedValue} annotation
 * is used to mark that type parameter. A value extractor for a non-generic type such as
 * {@link OptionalInt} needs to declare the type of the wrapped element(s) using
 * {@link ExtractedValue#type()}.
 * <p>
 * The extracted values are passed to the corresponding method of the {@link ValueReceiver}.
 * <p>
 * A typical value extractor implementation for {@code List} may look like this:
 *
 * <pre>
 * public class ListValueExtractor implements
 *         ValueExtractor&lt;List&lt;&#064;ExtractedValue ?&gt;&gt; {
 *
 *     &#064;Override
 *     public void extractValues(List&lt;?&gt; originalValue, ValueReceiver receiver) {
 *         for ( int i = 0; i &lt; originalValue.size(); i++ ) {
 *             receiver.indexedValue( "&lt;list element&gt;", i, originalValue.get( i ) );
 *         }
 *     }
 * }
 * </pre>
 *
 * @param <T> the container type handled by a specific implementation
 *
 * @author Gunnar Morling
 * @author Guillaume Smet
 * @see ExtractedValue
 * @see UnwrapByDefault
 * @since 2.0
 */
public interface ValueExtractor<T> {

    /**
     * Extracts the values to validate from the original object.
     *
     * @param originalValue the original value from which to extract the values, never
     * {@code null}
     * @param receiver the corresponding {@code ValueReceiver}
     */
    void extractValues(T originalValue, ValueReceiver receiver);

    /**
     * Provides a set of methods receiving value extracted by the {@link ValueExtractor}.
     * <p>
     * The value has to be passed to the method corresponding best to the type of the
     * original value.
     *
     * @since 2.0
     */
    interface ValueReceiver {

        /**
         * Receives the value extracted from an object.
         *
         * @param nodeName the name of the node representing the container element. If not
```

```
             * {@code null}, the name will be used when adding a container element node to the
             * {@link Path}
             * @param object the value to validate
             */
            void value(String nodeName, Object object);

            /**
             * Receives the value extracted from an iterable object that is not indexed (e.g.
             * a {@link Iterable}, {@link Set} or a {@link Map}).
             *
             * @param nodeName the name of the node representing the container element. If not
             * {@code null}, the name will be used when adding a container element node to the
             * {@link Path}
             * @param object the value to validate
             */
            void iterableValue(String nodeName, Object object);

            /**
             * Receives the value extracted from an indexed object (e.g. a {@link List}).
             *
             * @param nodeName the name of the node representing the container element. If not
             * {@code null}, the name will be used when adding a container element node to the
             * {@link Path}
             * @param i the index of the value in the original object
             * @param object the value to validate
             */
            void indexedValue(String nodeName, int i, Object object);

            /**
             * Receives the value extracted from a keyed object (e.g. a {@link Map}).
             *
             * @param nodeName the name of the node representing the container element. If not
             * {@code null}, the name will be used when adding a container element node to the
             * {@link Path}
             * @param key the key of the value in the original object
             * @param object the value to validate
             */
            void keyedValue(String nodeName, Object key, Object object);
        }
    }
```

The validation engine passes the container instance and a value receiver object to the `extractValues()` method. The value extractor is only invoked if the container is not `null`. Value extractor implementations must invoke one of the `ValueReceiver` methods for each element contained in the container, passing the element value and, optionally, a node name. When calling

- `value()`, the given value will be passed to the validation engine;
- `iterableValue()`, the given value will be passed to the validation engine and the corresponding property path node (see ConstraintViolation) will be marked as iterable, i.e. `Node#isInIterable()` returns `true`;
- `indexedValue()`, the given value will be passed to the validation engine and the corresponding property path node will be marked as iterable and it will have set the given index, i.e. `Node#getIndex()` returns the given index value;
- `keyedValue()`, the given value will be passed to the validation engine and the corresponding property path node will be marked as iterable and it will have set the given key, i.e. `Node#getKey()` returns the given key value.

When passing a non-null node name to any of the receiver methods, this node name will be used when adding a node of kind `CONTAINER_ELEMENT` to the property path (see ConstraintViolation for the property path construction rules). If null is passed as node name, no node will be appended to the property path. The resulting property path will then be the same as if the constraint had been given on the container instead of a container element. That is desirable for single-element wrapper types such as `Optional`, `OptionalInt` etc.

If an exception occurs during invocation of the `extractValues()` method, this exception is wrapped into a `ValidationException` by the Jakarta Validation engine.

The container value passed to a value extractor is retrieved from the element that hosts the type argument carrying the constraint or `@Valid` annotation:

```
public class Orders {

    private Map<String, @Valid @RetailOrder Order> ordersByName;
```

```
    public Map<@NotNull String, Order> getOrdersByName() {
        return ordersByName;
    }

    [...]
}
```

When validating the `@NotNull` constraint, the map as returned by the getter will be passed to the map key extractor in order to obtain the map keys. When validating the `@RetailOrder` constraint and performing cascaded validation, the map as obtained directly from the field will be passed to the map value extractor in order to obtain the map values.

4.1. @ExtractedValue

The `@ExtractedValue` annotation is used to denote the element extracted by a given value extractor:

Listing 4.2: `@ExtractedValue` *annotation*

```
package jakarta.validation.valueextraction;

/**
 * Marks the type parameter of a generic container type to which a {@link ValueExtractor} is
 * tied or specifies the type of the wrapped element(s) of non-generic container types.
 * <p>
 * Must be given exactly once for a value extractor type.
 *
 * @author Gunnar Morling
 * @author Guillaume Smet
 *
 * @see ValueExtractor
 * @since 2.0
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE_USE)
@Documented
public @interface ExtractedValue {

    /**
     * The type of the value extracted by the {@link ValueExtractor}. If not set, the type
     * will be automatically inferred from the type argument of the parameterized type.
     * <p>
     * Used to define value extractors for non-generic wrapper types e.g.
     * {@link OptionalInt}.
     * <p>
     * May not be used when {@code ExtractedValue} is defined on the type parameter of
     * a generic container type. A {@code ValueExtractorDefinitionException} will be thrown
     * in this case.
     *
     * @return the type of the value extracted by the value extractor
     */
    Class<?> type() default void.class;
}
```

The `@ExtractedValue` annotation must be specified exactly once for a value extractor type.

For generic container types (e.g. `java.util.List`), `@ExtractedValue` is to be specified on a type argument of the container type as used in the extractor definition. Only unbounded wildcard type arguments are supported as target for `@ExtractedValue` in this case.

> **NOTE** This implies that only one extractor is supported for a given generic type. I.e. there can be an extractor for `List<?>`, but not one for `List<String>` and one for `List<Integer>`.

For non-generic container types (e.g. `java.util.OptionalInt`), `@ExtractedValue` is to be specified on the container type as used in the extractor definition. The type of the wrapped element(s) must be specified using `@ExtractedValue#type()` in this case.

In case an illegal value extractor definition is detected, a `ValueExtractorDefinitionException` is raised.

4.2. @UnwrapByDefault

Value extractor definitions can be marked with the `@UnwrapByDefault` annotation. This causes constraints to be automatically applied to the wrapped value(s) if a constraint is found for an element of a type handled by that extractor (see Implicit unwrapping of containers):

Listing 4.3: `@UnwrapByDefault` *annotation*

```
package jakarta.validation.valueextraction;

/**
 * Marks a {@link ValueExtractor} definition so that it is applied automatically when
 * detecting constraints declared on the container type supported by the extractor, causing
 * the constraints to be applied to the container's elements instead of the container.
 * <p>
 * If needed, this behavior can be changed per constraint using {@link Unwrapping.Skip},
 * causing the constraints to be applied to the container itself:
 *
 * <pre>
 * &#064;SomeConstraint(payload = Unwrapping.Skip.class)
 * SomeContainerType container;
 * </pre>
 *
 * @author Guillaume Smet
 * @since 2.0
 */
@Target({ TYPE })
@Retention(RUNTIME)
@Documented
public @interface UnwrapByDefault {

}
```

4.3. Built-in value extractors

Compatible implementations provide value extractors for the following types out of the box:

- `java.util.Iterable`; `iterableValue()` must be invoked for each contained element, passing the string literal `<iterable element>` as node name
- `java.util.List`; `indexedValue()` must be invoked for each contained element, passing the string literal `<list element>` as node name
- `java.util.Map`; both map keys and map values are to be supported; `keyedValue()` must be invoked by the map key extractor for each contained key, passing the string literal `<map key>` as node name; `keyedValue()` must be invoked by the map value extractor for each contained value, passing the string literal `<map value>` as node name
- `java.util.Optional`; `value()` must be invoked, passing `null` as node name and passing the contained object as value or `null` if none is present
- `java.util.OptionalInt`, `java.util.OptionalLong` and `java.util.OptionalDouble`; the extracted value types must be `java.lang.Integer`, `java.lang.Long` and `java.lang.Double`, respectively. `value()` must be invoked, passing `null` as node name and passing the contained number as value or `null` if none is present. The extractors must be marked with `@UnwrapByDefault`.

In environments where JavaFX is present, compatible implementations additionally provide extractors for the following types out of the box:

- `javafx.beans.observable.ObservableValue`; `value()` must be invoked with the observable value, passing `null` as node name; the extractor must be marked with `@UnwrapByDefault`
- `javafx.beans.property.ReadOnlyListProperty` and `javafx.beans.property.ListProperty`; `indexedValue()` must be invoked for each contained element, passing the string literal `<list element>` as node name
- `javafx.beans.property.ReadOnlySetProperty` and `javafx.beans.property.SetProperty`; `iterableValue()` must be invoked for each contained element, passing the string literal `<iterable element>` as node name
- `javafx.beans.property.ReadOnlyMapProperty` and `javafx.beans.property.MapProperty`; both map keys and map values are to be supported; `keyedValue()` must be invoked by the map key extractor for each contained key, passing the string literal `<map key>` as node name; `keyedValue()` must be invoked by the map value extractor for each contained value, passing the string literal `<map value>` as node name

Additional value extractors (amending or overriding the set of built-in extractors) can be registered when bootstrapping the validation engine (see Registering ValueExtractor implementations).

4.4. Examples

A value extractor for the elements of `java.util.List`:

```
class ListValueExtractor implements ValueExtractor<List<@ExtractedValue ?>> {

    @Override
    public void extractValues(List<?> originalValue, ValueReceiver receiver) {
        for ( int i = 0; i < originalValue.size(); i++ ) {
            receiver.indexedValue( "<list element>", i, originalValue.get( i ) );
        }
    }
}
```

This extractor passes each element contained in the given list to the receiver object, using the literal `<list element>` as a node name.

A value extractor for `java.util.Optional`:

```
public class OptionalValueExtractor implements ValueExtractor<Optional<@ExtractedValue ?>> {

    @Override
    public void extractValues(Optional<?> originalValue, ValueReceiver receiver) {
        receiver.value( null, originalValue.orElse( null ) );
    }
}
```

This extractor passes the element wrapped by the given `Optional` to the receiver object, if present. `null` is passed as a node name, causing no node to be appended to the resulting property path. I.e. when the `@Size` constraint in `Optional<@Size(min=1) String> getName() { … }` is violated, the resulting property path will be the same as if a constraint hosted on the `getName` getter itself was violated.

A value extractor for `java.util.OptionalInt`:

```
@UnwrapByDefault
public class OptionalIntValueExtractor implements ValueExtractor<@ExtractedValue(type = Integer.class) OptionalInt> {

    @Override
    public void extractValues(OptionalInt originalValue, ValueReceiver receiver) {
        receiver.value( null, originalValue.isPresent() ? originalValue.getAsInt() : null );
    }
}
```

This extractor passes the `int` value wrapped by the given `OptionalInt` to the receiver object, if present. `null` is passed as a node name, causing no node to be appended to the resulting property path. As the extractor is marked with `@UnwrapByDefault`, any constraint declared on an element of type `OptionalInt` will implicitly be applied to the wrapped `int` value instead of the `OptionalInt` itself. As `OptionalInt` is a non-generic type (i.e. it has no type parameters), `@ExtractedValue` is given on the container type as used within the value extractor definition, specifying the type of the wrapped element via `type()`.

The following extractor definition is illegal as it specifies `@ExtractedValue` more than once:

```
public class IllegalMapExtractor implements ValueExtractor<Map<@ExtractedValue ?, @ExtractedValue ?>> { ... }
```

The following extractor definition is unsupported as it specifies `@ExtractedValue` on a non-wildcard type argument:

```
public class StringListValueExtractor implements ValueExtractor<List<@ExtractedValue String>> { ... }
```

5. Constraint declaration and validation process

The Jakarta Validation specification defines a framework for declaring constraints on JavaBean classes, fields and properties. Constraints are declared on types and evaluated against instances or graphs of instances.

Jakarta Validation also offers a way to declare constructor and method constraints where parameters and return values are the constrained elements. We will discuss method constraints declaration in detail in Method and constructor constraints.

Furthermore, constraints can be applied to the elements of generic container types such as `Map`, `List` or `Optional` or of non-generic container types such as `OptionalInt`. Container element constraints are discussed in detail in Container element constraints.

5.1. Requirements on classes to be validated

Objects hosting constraints and expecting to be validated by Jakarta Validation providers must fulfill the following requirements:

- Properties to be validated must follow the method signature conventions for JavaBeans read properties, as defined by the JavaBeans specification. These properties are commonly referred as getters.
- Static fields and static methods are excluded from validation.
- Constraints can be applied to interfaces and superclasses.

NOTE

What is a getter?

The JavaBeans specification specifies that a getter is a method whose

- name starts with `get` and has a return type but no parameter
- name starts with `is`, has no parameter and is returning `boolean`

The target of an annotation definition can be a

- type
- field or property
- constructor or method return value
- constructor or method parameter
- constructor or method cross-parameter
- container element

provided that:

- the constraint definition supports the specified target (`java.lang.annotation.Target`)
- one of the `ConstraintValidator`s declared on the constraint supports the declared type of the target or in the case of cross-parameter, one cross-parameter `ConstraintValidator` is present (see ConstraintValidator resolution algorithm to learn about `ConstraintValidator` resolution)
- in the case of container element constraints, a corresponding value extractor exists (see ValueExtractor resolution for the details of value extractor resolution)

5.1.1. Object validation

Constraint declarations can be applied to a class or an interface. Applying a constraint to a class or interface expresses a validation over the state of the class or the class implementing the interface.

5.1.2. Field and property validation

Constraint declarations can be applied on both fields and properties for the same object type. The same constraint should however not be duplicated between a field and its associated property (the constraint validation would be applied twice). It is recommended for objects holding constraint declarations to adhere to a single state access strategy (either annotated fields or properties).

NOTE

Jakarta Persistence and Jakarta Validation

For maximum portability, persistent properties hosting Jakarta Validation constraints should use the same access strategy used in Jakarta Persistence. In other words, place your Jakarta Validation constraint annotations on the same element (field or getter) as your Jakarta Persistence annotations.

When a field is annotated with a constraint declaration, field access strategy is used to access the state validated by such constraint.

When a property is annotated with a constraint declaration, property access strategy is used to access the state validated by such constraint.

When using field access strategy, the Jakarta Validation provider accesses the instance variable directly. When using the property access strategy, the Jakarta Validation provider accesses the state via the property accessor method. It is required that the class follows the method signature conventions for JavaBeans read properties (as defined by the JavaBeans `Introspector` class) for constrained properties when constrained properties are used. In this case, for every constraint property of type `T`, there is a getter method named `get<Property-name>`. The method must have no parameters. For `boolean` properties, `is<Property-name>` is an alternative name for the getter method. Specifically, if `getX` is the name of the getter method, where `X` is a string, the name of the persistent property is defined by the result of `java.beans.Introspector.decapitalize(X)`.

The fields or methods visibility are not constrained.

5.1.3. Graph validation

In addition to supporting instance validation, validation of graphs of objects is also supported. The result of a graph validation is returned as a unified set of constraint violations. `@Valid` is used to express validation traversal of an association.

Listing 5.1: `@Valid` *annotation*

```
/**
 * Marks a property, method parameter or method return type for validation cascading.
 * <p>
 * Constraints defined on the object and its properties are validated when the
 * property, method parameter or method return type is validated.
 * <p>
 * This behavior is applied recursively.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
@Target({ METHOD, FIELD, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Documented
public @interface Valid {
}
```

Consider the situation where bean `X` contains a field of type `Y`. By annotating field `Y` with the `@Valid` annotation, the Validator will validate `Y` (and its properties) when `X` is validated. The exact type `Z` of the value contained in the field declared of type `Y` (subclass, implementation) is determined at runtime. The constraint definitions of `Z` are used. This ensures proper polymorphic behavior for associations marked with `@Valid`.

Collection-valued, array-valued and generally `Iterable` fields and properties may also be decorated with the `@Valid` annotation. This causes the contents of the iterator to be validated. Any object implementing `java.lang.Iterable` is supported. This includes specifically:

- arrays of objects
- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map` (special treatment see below)

Each object provided by the iterator is validated. For `Map`, the value (retrieved by `getValue`) of each `Map.Entry` is validated (the key is not validated).

Like regular references, its type is determined at runtime and the constraint definitions for this particular type are used.

As of Jakarta Validation 2.0, `@Valid` can be applied to the elements of any generic container by putting it to the type argument(s) when using such container (e.g. `MultiMap<String, @Valid Address> addressesByType`), provided a value extractor implementation (see Value extractor definition) for that container type and the targeted type argument is present. There are built-in value extractors for the generic collection types listed above. In addition, there is a built-in extractor for the key objects of maps. See Built-in value extractors for the complete list of built-in value extractors.

`@Valid` also allows the validation of the elements of nested generic containers. `@Valid` must be put to a type argument of that nested container type in order to trigger validation of the elements of all the nested containers.

For a given container, the `@Valid` annotation should either be put to the container itself *or* to the type argument(s) of the container, but not both. This specification doesn't define the behaviour for having the `@Valid` annotation on the container *and* on the type argument(s) of the container. A specific implementation may, for example, execute the validation twice.

The `@Valid` annotation is applied recursively. A conforming implementation avoids infinite loops according to the rules described in Object

graph validation.

It is not supported to put `@Valid` to the type parameters of generic types or methods. It is also not supported to put `@Valid` to type arguments within the `extends` or `implements` clauses of type definitions. A future revision of this specification might define support for such usages of `@Valid`.

5.1.3.1. Examples

Example 5.1: Making elements of a list subject to graph validation

```
public class User {

    // preferred style as of Jakarta Validation 2.0
    private List<@Valid PhoneNumber> phoneNumbers;

    // traditional style; continues to be supported
    @Valid
    private List<PhoneNumber> phoneNumbers;

    // discouraged; either the container or the type argument(s) should be
    // annotated with @Valid, but not both
    @Valid
    private List<@Valid PhoneNumber> phoneNumbers;
}
```

Example 5.2: Making values of a map subject to graph validation

```
public class User {

    // preferred style as of Jakarta Validation 2.0
    private Map<AddressType, @Valid Address> addressesByType;

    // traditional style; continues to be supported
    @Valid
    private Map<AddressType, Address> addressesByType;

    // discouraged; either the map or the map value type argument should be
    // annotated with @Valid, but not both
    @Valid
    private Map<AddressType, @Valid Address> addressesByType;
}
```

Example 5.3: Making keys and values of a map subject to graph validation

```
public class User {

    private Map<@Valid AddressType, @Valid Address> addressesByType;
}
```

Example 5.4: Making elements of a nested list subject to graph validation

```
public class User {
    private Map<String, List<@Valid Address>> addressesByType;
}
```

In this example, all `Address` objects contained in the lists of the `addressesByType` map will be validated. Two value extractors are invoked for this:

- the extractor for `Map` values will be invoked to obtain all map values (lists of `Address`)

- for each extracted list of addresses, the extractor for `List` elements will be invoked, providing the `Address` objects from each list in the map

Example 5.5: Making keys and values of a nested map subject to graph validation

```
public class User {
    private Map<String, Map<@Valid AddressType, @Valid Address>> addressesByUserAndType;
}
```

In this example, all `AddressType` objects and all `Address` objects contained in the maps of the `addressesByUserAndType` map will be validated. The following value extractors are invoked for this:

- the extractor for `Map` values will be invoked to obtain all map values (maps of addresses by address type)
- for each extracted map, the extractor for `Map` keys will be invoked, providing the `AddressType` objects from each of the nested maps
- for each extracted map, the extractor for `Map` values will be invoked, providing the `Address` objects from each of the nested maps

5.2. Constraint declaration

Constraint declarations are placed on classes or interfaces primarily through annotations. A constraint annotation (see Constraint annotation), can be applied to a type, on any of the type's fields or on any of the JavaBeans-compliant properties.

When a constraint is defined on a class, the class instance being validated is passed to the `ConstraintValidator`. When a constraint is defined on a field, the value of the field is passed to the `ConstraintValidator`. When a constraint is defined on a getter, the result of the getter invocation is passed to the `ConstraintValidator`.

Method and constructor constraints discusses in detail constraints on methods and constructors.

Constraints can also be applied to the elements of container types, e.g. to the elements of a `List`-typed property or to the value wrapped by an `Optional` object returned by a method. Container element constraints are discussed in Container element constraints.

5.3. Inheritance (interface and superclass)

A constraint declaration can be placed on an interface. For a given class, constraint declarations held on superclasses as well as interfaces are evaluated by the Jakarta Validation provider. Rules are formally described in Formal group definitions.

The effect of constraint declarations is cumulative. Constraints declared on a superclass getter will be validated along with any constraints defined on an overridden version of the getter according to the Java Language Specification visibility rules.

5.4. Group and group sequence

A group defines a subset of constraints. Instead of validating all constraints for a given object graph, only a subset is validated. This subset is defined by the group or groups targeted. Each constraint declaration defines the list of groups it belongs to. If no group is explicitly declared, a constraint belongs to the `Default` group.

Groups are represented by interfaces.

Example 5.6: Definition of groups

```
/**
 * Validation group verifing that a user is billable
 */
public interface Billable {}

/**
 * Customer can buy without any harrassing checking process
 */
public interface BuyInOneClick {
}
```

A constraint can belong to one or more groups.

Example 5.7: Assign groups to constraints

```
/**
 * User representation
 */
public class User {
    @NotNull
    private String firstname;

    @NotNull(groups = Default.class)
    private String lastname;

    @NotNull(groups = {Billable.class, BuyInOneClick.class})
    private CreditCard defaultCreditCard;
}
```

During the validation call, one or more groups are validated. All the constraints belonging to this set of groups is evaluated on the object graph. In Assign groups to constraints, `@NotNull` is checked on `defaultCreditCard` when either the `Billable` or `BuyInOneClick` group is validated. `@NotNull` on `firstname` and `lastname` are validated when the `Default` group is validated. Reminder: constraints held on superclasses and interfaces are considered.

`Default` is a group predefined by the specification.

Listing 5.2: `Default` *group*

```
package jakarta.validation.groups;

/**
 * Default Jakarta Validation group.
 * <p>
 * Unless a list of groups is explicitly defined:
 * <ul>
 *     <li>constraints belong to the {@code Default} group</li>
 *     <li>validation applies to the {@code Default} group</li>
 * </ul>
 * Most structural constraints should belong to the default group.
 *
 * @author Emmanuel Bernard
 */
public interface Default {
}
```

5.4.1. Group inheritance

In some situations, a group is a superset of one or more groups. This can be described by Jakarta Validation. A group may inherit one or more groups by using interface inheritance.

Example 5.8: Groups can inherit other groups

```
/**
 * Customer can buy without harrassing checking process
 */
public interface BuyInOneClick extends Default, Billable {}
```

For a given interface `Z`, constraints marked as belonging to the group `Z` (i.e. where the annotation element `groups` contains the interface `Z`) or any of the super interfaces of `Z` (inherited groups) are considered part of the group `Z`.

In the following example:

Example 5.9: Use of a inherited group

```
/**
```

```
 * User representation
 */
public class User {
    @NotNull
    private String firstname;

    @NotNull(groups = Default.class)
    private String lastname;

    @NotNull(groups = Billable.class)
    private CreditCard defaultCreditCard;
}
```

validating the group `BuyInOneClick` will lead to the following constraints checking:

- `@NotNull` on `firstname` and `lastname`
- `@NotNull` on `defaultCreditCard`

because `Default` and `Billable` are superinterfaces of `BuyInOneClick`.

5.4.2. Group sequence

By default, constraints are evaluated in no particular order regardless of which groups they belong to. It is however useful in some situations to control the order of constraints evaluation. There are often scenarios where a preliminary set of constraints should be evaluated prior to other constraints. Here are two examples:

- The second group depends on a stable state to run properly. This stable state is verified by the first group.
- The second group is a heavy consumer of time, CPU or memory and its evaluation should be avoided if possible.

To implement such ordering, a group can be defined as a sequence of other groups. Each group in a group sequence must be processed sequentially in the order defined by `@GroupSequence.value` when the group defined as a sequence is requested. Note that a group member of a sequence can itself be composed of several groups via inheritance or sequence definition. In this case, each composed group must respect the sequence order as well.

Processing a group is defined in Validation routine ; if one of the groups processed in the sequence generates one or more constraint violations, the groups following in the sequence must not be processed. This ensures that a set of constraints is evaluated only if another set of constraints is valid.

Groups defining a sequence and groups composing a sequence must not be involved in a cyclic dependency:

- either directly or indirectly
- either through cascaded sequence definitions or group inheritance

If a group containing such a circularity is evaluated, a `GroupDefinitionException` is raised.

Groups defining a sequence should not directly inherit other groups. In other words, the interface hosting the group sequence should not have any super interface.

Groups defining a sequence should not be used directly in constraint declarations. In other words, the interface hosting the group sequence should not be used in a constraint declaration.

To define a group as a sequence, the interface must be annotated with the `@GroupSequence` annotation.

Listing 5.3: `@GroupSequence` *annotation*

```
/**
 * Defines group sequence.
 * <p>
 * The interface hosting {@code @GroupSequence} is representing
 * the group sequence.
 * When hosted on a class, represents the {@link Default} group
 * for that class.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
@Target({ TYPE })
@Retention(RUNTIME)
```

```
@Documented
public @interface GroupSequence {

    Class<?>[] value();
}
```

Here is a usage example:

Example 5.10: Make use of group sequence

```
@ZipCodeCoherenceChecker(groups = Address.HighLevelCoherence.class)
public class Address {
    @NotNull @Size(max = 50)
    private String street1;

    @NotNull @ZipCode
    private String zipCode;

    @NotNull @Size(max = 30)
    private String city;

    /**
     * check coherence on the overall object
     * Needs basic checking to be green first
     */
    public interface HighLevelCoherence {}

    /**
     * check both basic constraints and high level ones.
     * high level constraints are not checked if basic constraints fail
     */
    @GroupSequence({Default.class, HighLevelCoherence.class})
    public interface Complete {}
}
```

In Make use of group sequence, when the `Address.Complete` group is validated, all constraints belonging to the `Default` group are validated. If any of them fail, the validation skips the `HighLevelCoherence` group. If all `Default` constraints pass, `HighLevelCoherence` constraints are evaluated.

NOTE A given constraint can belong to two or more groups ordered by a sequence. In this case, the constraint is evaluated as part of the first group and ignored in the subsequent group(s). See Validation routine for more information.

5.4.3. Redefining the Default group for a class

In Make use of group sequence, validating the `Default` group does not validate `HighLevelCoherence` constraints. To ensure a complete validation, a user must use the `Complete` group. This breaks some of the encapsulation you could expect. You can work around this by redefining what the `Default` group means for a given class. To redefine `Default` for a class, place a `@GroupSequence` annotation on the class; this sequence expresses the sequence of groups that does substitute `Default` for this class.

Example 5.11: Redefining Default group for Address

```
@GroupSequence({Address.class, HighLevelCoherence.class})
@ZipCodeCoherenceChecker(groups = Address.HighLevelCoherence.class)
public class Address {
    @NotNull @Size(max = 50)
    private String street1;

    @NotNull @ZipCode
    private String zipCode;

    @NotNull @Size(max = 30)
    private String city;

    /**
```

```
     * check coherence on the overall object
     * Needs basic checking to be green first
     */
    public interface HighLevelCoherence {}
}
```

In Redefining Default group for Address, when an address object is validated for the group `Default`, all constraints belonging to the group `Default` and hosted on `Address` are evaluated. If none fails, all `HighLevelCoherence` constraints present on `Address` are evaluated. In other words, when validating the `Default` group for `Address`, the group sequence defined on the `Address` class is used.

Since sequences cannot have circular dependencies, using `Default` in the declaration of a sequence is not an option. Constraints hosted on a class `A` and belonging to the `Default` group (by default or explicitly) implicitly belong to the group `A`.

A sequence defined on a class `A` (i.e. redefining the `Default` groups for the class) must contain the group `A`. In other words, the default constraints hosted on a class must be part of the sequence definition. If a `@GroupSequence` redefining the `Default` group for a class `A` does not contain the group `A`, a `GroupDefinitionException` is raised when the class is validated or when its metadata is requested.

5.4.4. Implicit grouping

It is possible to implicitly group several constraints in the same group without explicitly listing such a group in the constraint declaration. Every constraint hosted on an interface `Z` and part of the `Default` group (implicitly or explicitly) belongs to the group `Z`. This is useful to validate the partial state of an object based on a role represented by an interface.

Example 5.12: Example of interface / group hosting constraints

```
/**
 * Auditable object contract
 */
public interface Auditable {
    @NotNull String getCreationDate();
    @NotNull String getLastUpdate();
    @NotNull String getLastModifier();
    @NotNull String getLastReader();
}

/**
 * Represents an order in the system
 */
public class Order implements Auditable {
    private String creationDate;
    private String lastUpdate;
    private String lastModifier;
    private String lastReader;

    private String orderNumber;

    public String getCreationDate() {
        return this.creationDate;
    }

    public String getLastUpdate() {
        return this.lastUpdate;
    }

    public String getLastModifier() {
        return this.lastModifier;
    }

    public String getLastReader() {
        return this.lastReader;
    }

    @NotNull @Size(min=10, max=10)
    public String getOrderNumber() {
        return this.orderNumber;
    }
}
```

When an `Order` object is validated on the `Default` group, the following constraints are validated: `@NotNull` on `getCreationDate`, `getLastUpdate`, `getLastModifier`, `getLastReader`, `getOrderNumber` and `@Size` on `getOrderNumber` as all belong to the `Default` group.

When an `Order` object is validated on the `Auditable` group, the following constraints are validated: `@NotNull` on `getCreationDate`, `getLastUpdate`, `getLastModifier`, `getLastReader`. Only the constraints present on `Auditable` (and any of its super interfaces) and belonging to the `Default` group are validated when the group `Auditable` is requested. It allows the caller to validate that a given object can be safely audited even if the object state itself is not valid.

5.4.5. Group conversion

When performing cascading validation, it is possible to use a different group than the one originally requested using the group conversion feature. Group conversions are declared by using the `@ConvertGroup` annotation.

Listing 5.4: @ConvertGroup annotation

```
package jakarta.validation.groups;

/**
 * Converts group {@code from} to group {@code to} during cascading.
 * <p>
 * Can be used everywhere {@link Valid} is used and must be on an element
 * annotated with {@code Valid}.
 *
 * @author Emmanuel Bernard
 * @since 1.1
 */
@Target({ METHOD, FIELD, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
public @interface ConvertGroup {

    /**
     * The source group of this conversion.
     * @return the source group of this conversion
     */
    Class<?> from() default Default.class;

    /**
     * The target group of this conversion.
     * @return the target group of this conversion
     */
    Class<?> to();

    /**
     * Defines several {@link ConvertGroup} annotations
     * on the same element.
     */
    @Target({ METHOD, FIELD, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    public @interface List {

        ConvertGroup[] value();
    }
}
```

`@ConvertGroup` and `@ConvertGroup.List` can be used everywhere `@Valid` can be used (associations, method/constructor parameters and return value). If these annotations are used without `@Valid`, a `ConstraintDeclarationException` is raised.

When an element is annotated with `@Valid`, validation is propagated. Groups are passed as is to the nested elements unless the `@ConvertGroup` annotation is used.

If the group expected to be passed to the nested element validation is defined as the `from` attribute of a `@ConvertGroup` annotation, the group used to effectively validate the nested element is the corresponding group defined in the `to` attribute.

If no value for the `from` attribute is specified, `Default.class` will be used as the source group of the conversion.

Rules are not executed recursively. If a rule is found matching, subsequent rules are no longer evaluated. In particular, if a set of `@ConvertGroup` declaration chains group `A` to `B` and `B` to `C`, the group `A` will be converted to `B` and not to `C`. This both makes rules clearer and lets you switch two groups.

It is not legal to have more than one conversion rule containing the same `from` value. In this case, a `ConstraintDeclarationException` is raised.

Like regular constraint declarations, the `from` attribute cannot refer to a group sequence. A `ConstraintDeclarationException` is raised in this situation. The `to` attribute can. The group sequence will then be expanded before validating the associated object.

NOTE

When validation is done, group sequences are expanded before validating the object and its cascaded objects with the expected groups. Group conversion on an associated object happens on the already expanded groups.

The group referred to in `@ConvertGroup.from` works on expanded groups (i.e., after the group sequence has been expanded), not necessarily groups passed to the various `validate` methods.

The group referred to in `@ConvertGroup.to` will be expanded before validating the cascaded object just like a call to the various `validate` method would have done.

NOTE

Like most Jakarta Validation error cases, an illegal set of rules can be discovered statically (at compile time). For example, an annotation processor could detect such errors.

NOTE

Group circularity in a group conversion are not problematic because:

- only one rule is applied for a given cascade (rules are not applied recursively)
- validation cascading is stopped when the same instance / property is validated with the same group in a given path (existing rule)

`@ConvertGroup` and `@ConvertGroup.List` can only be placed where `@Valid` is present to ensure proper respect of the Liskov substitution principle: if rules were to be defined on an overriding method of a method marked as cascading validation, the rules could end up altering the list of constraints validated by the super type and thus violating the Liskov substitution principle.

Likewise, if a sub type overrides/implements a method originally defined in several parallel types of the hierarchy (e.g. two interfaces not extending each other, or a class and an interface not implemented by said class) and if that method's return value has been marked for cascading validation in one of the parallel types, no group conversion rule may be declared for that method's return value in the parallel types of the hierarchy. This again is to avoid an unexpected altering of the post conditions to be guaranteed to the caller.

If any of these rules is violated, a `ConstraintDeclarationException` is raised by default as defined in Method constraints in inheritance hierarchies.

Group conversion is quite useful to facilitate object graph reuse without spreading the validation group definitions across several layers. Let's look at an example.

5.4.5.1. Group conversion examples

In this example we will reuse the `Address` group split and match it to the `User` group split.

Example 5.13: Example of group conversion

```
public interface Complete extends Default {}
public interface BasicPostal {}
public interface FullPostal extends BasicPostal {}

public class Address {
    @NotNull(groups=BasicPostal.class)
    String street1;

    String street2;

    @ZipCode(groups=BasicPostal.class)
    String zipCode;

    @CodeChecker(groups=FullPostal.class)
    String doorCode;
}

public class User {
```

```
    @Valid
    @ConvertGroup(from=Default.class, to=BasicPostal.class)
    @ConvertGroup(from=Complete.class, to=FullPostal.class)
    Set<Address> getAddresses() { [...] }
}
```

When validating an instance of `User` with the `Default` group, the associated addresses are validated with the `BasicPostal` group. When validating an instance of `User` with the `Complete` group, the associated addresses are validated with the `FullPostal` group.

Group conversions can also be applied during container element validation:

Example 5.14: Example of container element validation with group conversion

```
public class User {
    Set<
        @Valid
        @ConvertGroup(from=Default.class, to=BasicPostal.class)
        @ConvertGroup(from=Complete.class, to=FullPostal.class)
        Address
    > getAddresses() { [...] }
}
```

The following example shows an illegal declaration of a group conversion rule on a method's return value:

Example 5.15: Example of an illegal group conversion

```
public interface BasicPostal {}

public class Order { [...] }

public interface RetailOrderService {

    @Valid
    Order placeOrder(String itemNo, int quantity);
}

public interface B2BOrderService {

    @Valid
    @ConvertGroup(from=Default.class, to=BasicPostal.class)
    Order placeOrder(String itemNo, int quantity);
}

public class OrderService implements RetailOrderService, B2BOrderService {

    @Override
    public Order placeOrder(String itemNo, int quantity) {
        [...]
    }
}
```

Here the class `OrderService` implements the two unrelated interfaces `RetailOrderService` and `B2BOrderService`, which both define a method `placeOrder()`, marking the return value as cascaded.

The group conversion declared in `B2BOrderService` is illegal as per the rules defined in the previous section, since the set of applied validation groups might be altered unexpectedly for a client of the `RetailOrderService` interface.

5.4.6. Formal group definitions

The formal rules defining groups are as followed. *Text in italic are comments about the rules.*

For every class `X`:

1. For each superclass `Y` of `X`, the group `Y` contains all constraints of the group `Y` of `Y` *this rule prepares formal concepts for recursive*

discovery

2. The group `X` contains the following constraints: group `X` is a group used on sequences redefining the default group on a class (see Redefining the Default group for a class)
   a. every constraint declared by the class `X` which does not declare a group or does declare the group `Default` explicitly. *all Default constraints hosted on X*
   b. every constraint declared by any interface implemented by `X` and not annotated `@GroupSequence` which does not explicitly declare a group or does declare the group `Default` explicitly. *all Default constraints hosted on interfaces of X: constraints are inherited by the class hierarchy. Interfaces marked as @GroupSequence are ignored.*
   c. if `X` has a direct superclass `Y`, every constraint in the group `Y` *all Default constraints hosted on the superclasses of X: constraints are inherited by the class hierarchy*

3. If `X` has no `@GroupSequence` annotation, the group `Default` contains the following constraints: *this rule defines which constraints are evaluated when validating Default on X.*
   a. every constraint in the group `X`
   b. if `X` has a direct superclass `Y`, every constraint in the group `Default` of `Y` *this rule is necessary in case Y redefines the group Default*

4. If `X` does have a `@GroupSequence` annotation, the group `Default` contains every constraint belonging to every group declared by the `@GroupSequence` annotation. *this rule describes how a class can redefine the group Default for itself (see Redefining the Default group for a class)*
   - the `@GroupSequence` annotation must declare the group `X`

5. For every interface `Z`, the group `Z` contains the following constraints: *this rule defines how non Default groups are defined*
   a. every constraint declared by the interface `Z` which does not explicitly declare a group or does declare the group `Default` explicitly. *all Default constraints hosted on Z: this rule formally defines implicit grouping per interface (see Implicit grouping)*
   b. every constraint (which does not explicitly declare a group) declared by any superinterface not annotated `@GroupSequence` of the interface `Z` *all Default constraints hosted on interfaces of Z: groups can be inherited (see Group inheritance)*
   c. every constraint declared by the class `X` which explicitly declares the group `Z` *every constraint hosted by X and marked as belonging to the group Z*
   d. every constraint declared by any interface implemented by `X` and not annotated `@GroupSequence` which explicitly declares the group `Z` *every constraint hosted by any interface of X and marked as belonging to the group Z*
   e. if `X` has a direct superclass `Y`, every constraint in the group `Z` of `Y` *every constraint hosted by any superclass of X and marked as belonging to the group Z*

6. For every interface `Z` annotated `@GroupSequence`, the group `Z` contains every constraint belonging to every group declared by the `@GroupSequence` annotation. *defines the composition side of group sequence but does not define the ordering behavior of sequence (see Group sequence)*

When a given group `G` (represented by an interface `G`) is requested for the validation of a class `X`:

- constraints belonging to the group `G` are evaluated
- if the interface `G` is not annotated `@GroupSequence`, every group represented by the super interface of `G` are requested for validation
- if the interface `G` is annotated with `@GroupSequence`, every group represented by the interfaces declared by the `@GroupSequence` annotation are requested for validation
  - the validation of groups declared to the `@GroupSequence` must happen in the sequencing order declared by `@GroupSequence`: the sequencing order is propagated to the groups composing the sequenced group (via inheritance or group sequence)
  - if a group validation triggers the failure of one or more constraints, groups following in the sequence must not be evaluated.
- if the group `G` represents the `Default` group of `X` overridden by `@GroupSequence`, operations are equivalent

When the `Default` group of a given class `X` is overridden via `@GroupSequence`, its validation is as followed:

- every group represented by the interfaces declared by the `@GroupSequence` annotation are requested for validation
  - the validation of groups declared to the `@GroupSequence` must happen in the sequencing order declared by `@GroupSequence`: the sequencing order is propagated to the groups composing the sequenced group (via inheritance or group sequence)
  - if a group validation triggers the failure of one or more constraints, groups following in the sequence must not be evaluated.

Unless defined by a `@GroupSequence`, evaluation ordering is not constrained. In particular, several groups can be validated in the same pass. If a group definition leads to a circular sequencing order between groups, a `GroupDefinitionException` is raised.

NOTE A group `G` sequenced (directly or indirectly) to be executed before itself is not considered a circular reference.

5.5. Container element constraints

Constraints can be applied to the elements of generic containers, e.g. `List`, `Map` or `Optional`. This is done by putting constraint annotations to the type arguments of such containers.

Container element constraints can be used within the following declarations:

- fields,
- properties,
- method or constructor parameters or
- method return values.

Example 5.16: Container element constraints

```
private List<@Email String> emails;

public Optional<@Email String> getEmail() {
    [...]
}

public Map<@NotNull String, @ValidAddress Address> getAddressesByType() {
    [...]
}

public List<@NotBlank String> getMatchingRecords(List<@NotNull @Size(max=20) String> searchTerms) {
    [...]
}
```

When a field, property, executable parameter or method return value which is of a container type gets validated, then all values contained in the container will be validated provided that their container element type is constrained. Any container element constraints of that element will be validated alongside any other constraints hosted by that element. For container element constraints, the same rules for validation groups and group sequences apply as for any other constraint on the same element.

When a container element is constrained, the validation engine invokes a value extractor (see Value extractor definition) which retrieves the value(s) from the container so they can be validated. This may be a single value - e.g. in the case of `Optional` which wraps exactly one value if the `Optional` is not empty - or multiple values in the case of collection types.

Container element constraints can be applied to nested container types:

Example 5.17: Container element constraints on nested containers

```
private Map<String, @NotEmpty List<@ValidAddress Address>> addressesByType;
```

In such case multiple value extractors will be invoked.

In the example above,

- the extractor for `Map` values will be invoked to obtain all map values (lists of `Address`)
- for each extracted list of addresses, the `@NotEmpty` constraint will be validated and the extractor for `List` elements will be invoked, providing the `Address` objects from each list in the map
- the `@ValidAddress` constraint will be applied to all elements of all lists stored in the map

It is not supported to declare container element constraints on the type parameters of generic types or methods. It is also not supported to declare container element constraints on type arguments within the `extends` or `implements` clauses of type definitions. I.e. the following usages are unsupported:

Example 5.18: Unsupported usage of container element constraints on generic types and methods

```
public class NonNullList<@NotNull T> {
    [...]
}

public class ContainerFactory {
    <@NotNull T> Container<T> instantiateContainer(T wrapped) { [...] }
```

```
}

public class NonNullSet<T> extends Set<@NotNull T> {
    [...]
}
```

A future revision of this specification might define support for such usages of container element constraints.

5.5.1. Implicit unwrapping of containers

Besides specifying container element constraints on type arguments, it is also possible to declare container element constraints on non-generic container types. This is done by means of *implicit unwrapping*, i.e. a constraint doesn't apply to the annotated container itself but to its element(s). Examples for types being subject to implicit unwrapping are `java.util.OptionalInt`, `OptionalLong` and `OptionalDouble` as well as JavaFX's non-generic property types such as `StringProperty` or `IntegerProperty`:

Example 5.19: Implicit unwrapping of container elements

```
@Min(1)
private OptionalInt optionalNumber;

@Negative
private LongProperty negativeLong;

@Positive
private IntegerProperty positiveInt;

private final ListProperty<@NotBlank StringProperty> notBlankStrings;
```

Here the `@Min`, `@Negative`, `@Positive` and `@NotBlank` constraints don't apply to the annotated `OptionalInt`, `LongProperty`, `IntegerProperty` and `StringProperty` objects themselves, but rather to the wrapped numeric and string values, respectively.

For this to work, an unambiguously resolvable value extractor (see ValueExtractor resolution algorithm for applying container-level constraints to container elements) must be defined which carries the `@UnwrapByDefault` annotation (see @UnwrapByDefault).

If needed, the target (container or container element) of a constraint declared on a container can be explicitly specified via the `Unwrap` and `Skip` payload definitions:

Listing 5.5: `Payload` types for unwrapping control

```
package jakarta.validation.valueextraction;

/**
 * Set of interfaces used in the {@code payload()} of a constraint to indicate if a value
 * should be unwrapped before validation.
 * <p>
 * This is used to overwrite the default configuration defined on the {@link ValueExtractor}.
 *
 * @author Guillaume Smet
 * @since 2.0
 */
public interface Unwrapping {

    /**
     * Unwrap the value before validation.
     *
     * @since 2.0
     */
    public interface Unwrap extends Payload {
    }

    /**
     * Skip the unwrapping if it has been enabled on the {@link ValueExtractor} by the
     * {@link UnwrapByDefault}
     * annotation.
     *
```

```
     * @since 2.0
     */
    public interface Skip extends Payload {
    }

}
```

This is useful for applying a constraint given on a non-generic container to

- the container element(s) if there is no value extractor marked with `@UnwrapByDefault` (by using `Unwrap`)
- the container itself in case there is a value extractor marked with `@UnwrapByDefault` (by using `Skip`)

For instance the `@NotNull` constraint is applied to the `StringProperty` container:

```
@NotNull(payload = Unwrapping.Skip.class)
private StringProperty name;
```

If both `Unwrap` and `Skip` are present in the definition of a payload, a `ConstraintDeclarationException` is raised.

NOTE

For the sake of readability, when applying constraints to the elements of a generic container, it is strongly recommended to put the constraints to the type argument instead of the container itself in conjunction with `Unwrapping.Unwrap`. I.e. you should prefer

```
List<@Email String> emails;
```

over

```
@Email(payload = Unwrapping.Unwrap.class)
List<String> emails;
```

5.6. Method and constructor constraints

NOTE In the following, the term "method constraint" refers to constraints declared on methods as well as constructors.

Method constraints are declared by adding constraint annotations directly to methods or constructors and/or their parameters. In the former case, all the parameters of an executable (cross-parameter constraint) or the return value is constrained, in the latter individual parameters are constrained. As with bean constraints, this can be done using either actual Java annotations or using an XML constraint mapping file (see Method-level overriding). Jakarta Validation providers are free to provide additional means of defining method constraints such as an API-based approach.

Getters are not considered constrained methods by default (see Method and constructor validation).

5.6.1. Requirements on methods to be validated

Static methods are ignored by validation. Putting constraints on a static method is not portable. No other restrictions exist from the perspective of this specification, however it is possible that technologies integrating with method validation impose further restrictions to methods for which a validation shall be applied. For instance certain integration technologies might require that methods to be validated must have `public` visibility and/or must not be `final`.

5.6.2. Declaring parameter constraints

Parameter constraints are declared by putting constraint annotations on method or constructor parameters.

Example 5.20: Declaring parameter constraints

```
public class OrderService {

    public OrderService(@NotNull CreditCardProcessor creditCardProcessor) {
        [...]
    }

    public void placeOrder(
        @NotNull @Size(min=3, max=20) String customerCode,
```

```
            @NotNull Item item,
            @Min(1) int quantity) {
            [...]
        }
    }
```

Using constraint annotations, several preconditions are defined here. These preconditions which must be satisfied in order to legally invoke the methods of `OrderService` are:

- The `CreditCardProcessor` passed to the constructor must not be null.
- The customer code passed to the `placeOrder()` method must not be null and must be between 3 and 20 characters long.
- The `Item` passed to the `placeOrder()` method must not be null.
- The quantity value passed to the `placeOrder()` method must be 1 at least.

Note that declaring these constraints does not automatically cause their validation when the concerned methods are invoked. It's the responsibility of an integration layer to trigger the validation of the constraints using a method interceptor, dynamic proxy or similar. See section Triggering method validation for more details.

TIP

In order to use constraint annotations for method or constructor parameters, their element type must be `ElementType.PARAMETER`. In order to use constraint annotations for cross-parameter validation or on the return values of methods or constructors (see the following sections), their element type must be `ElementType.METHOD` respectively `ElementType.CONSTRUCTOR`. All built-in constraints support these element types, and it is considered a best practice to do the same for custom constraints.

5.6.2.1. Cross-parameter constraints

Cross-parameter constraints allow to express constraints based on the value of several method parameters, similar to class-level constraints which are based on several properties of a given class. Cross-parameter constraints are declared by putting cross-parameter constraint annotations on methods or constructors as shown in the following example.

Example 5.21: Declaring cross-parameter constraints

```
public class CalendarService {

    @ConsistentDateParameters
    public void createEvent(
        String title,
        @NotNull Date startDate,
        @NotNull Date endDate) {
        [...]
    }
}
```

The cross-parameter constraint annotation expresses here that the given start date must be before the passed end date in order to legally invoke the `createEvent()` method. The example also shows that it is often useful to combine constraints directly placed on individual parameters (`e.g. @NotNull`) and cross-parameter constraints.

TIP

Cross-parameter constraints as well as return value constraints are declared directly on a method or a constructor. To make it obvious for a reader that an annotation refers to the parameters of a method or constructor and not its return value, it is recommended to choose a name which clearly expresses this intention.

It is not legal to declare a cross-parameter constraint on a method or constructor which has no parameters. A `ConstraintDeclarationException` is raised in this case.

Some constraints can target an executable's return value as well as its array of parameters. They are known to be both generic and cross-parameter constraints. When using such a constraint on an executable to target the parameters, one must set `validationAppliesTo` if there is an ambiguity. The set of ambiguities is described in validationAppliesTo. Even without ambiguity, it is recommended to explicitly set `validationAppliesTo` to `ConstraintTarget.PARAMETERS` as it improves readability.

5.6.2.2. Naming parameters

In case the validation of a parameter constraint fails, the concerned parameter needs to be identified in the resulting `ConstraintViolation` (see section ConstraintViolation). Jakarta Validation defines the `jakarta.validation.ParameterNameProvider` API to which the retrieval of

parameter names is delegated:

Listing 5.6: `ParameterNameProvider` *interface*

```
/**
 * Provides names for method and constructor parameters.
 * <p>
 * Used by the Jakarta Validation runtime when creating constraint violation
 * objects for violated method constraints.
 * <p>
 * Implementations must be thread-safe.
 *
 * @author Gunnar Morling
 * @since 1.1
 */
public interface ParameterNameProvider {

    /**
     * Returns the names of the parameters of the given constructor.
     *
     * @param constructor the constructor for which the parameter names shall be
     *        retrieved; never {@code null}
     * @return a list containing the names of the parameters of the given
     *         constructor; may be empty but never {@code null}
     */
    List<String> getParameterNames(Constructor<?> constructor);

    /**
     * Returns the names of the parameters of the given method.
     *
     * @param method the method for which the parameter names shall be retrieved;
     *        never {@code null}
     * @return a list containing the names of the parameters of the given method;
     *         may be empty but never {@code null}
     */
    List<String> getParameterNames(Method method);
}
```

A conforming Jakarta Validation implementation provides a default `ParameterNameProvider` implementation which returns parameter names as stored in the class file containing the validated executable, if present. A conforming implementation must either use the Java reflection API or ensure behavioral compatibility to using the reflection API in the following way:

- Obtain the method's or constructor's parameters via `java.lang.reflect.Executable.getParameters()`
- Obtain each parameter's name via `java.lang.reflect.Parameter.getName()`

Depending on whether the class file of the validated executable contains parameter name information or not, the actual parameter names as provided in the executable's definition will be returned or synthetic names in the form arg`PARAMETER_INDEX`, where `PARAMETER_INDEX` starts at 0 for the first parameter, e.g. `arg0`, `arg1` etc.

Jakarta Validation providers and integrators are free to provide additional implementations (e.g. based on annotations specifying parameter names, debug symbols etc.). If a user wishes to use another parameter name provider than the default implementation, they may specify the provider to use with help of the bootstrap API (see Bootstrapping) or the XML configuration (see XML configuration: META-INF/validation.xml).

If an exception occurs during invocation of the `getParameterNames()` methods, this exception is wrapped into a `ValidationException` by the Jakarta Validation engine.

5.6.3. Declaring return value constraints

Return value constraints are declared by putting constraint annotations directly on a method or constructor.

Some constraints can target both the return value and the array of parameters of an executable. They are known to be both generic and cross-parameter constraints. When using such constraint on an executable to target the return value, one must set `validationAppliesTo` in case there is an ambiguity. The set of ambiguities is described in validationAppliesTo. Even without ambiguity, it is recommended to explicitly set `validationAppliesTo` to `ConstraintTarget.RETURN_VALUE` as it improves readability.

Example 5.22: Declaring return value constraints

```
public class OrderService {

    private CreditCardProcessor creditCardProcessor;

    @ValidOnlineOrderService
    public OrderService(OnlineCreditCardProcessor creditCardProcessor) {
        this.creditCardProcessor = creditCardProcessor;
    }

    @ValidBatchOrderService
    public OrderService(BatchCreditCardProcessor creditCardProcessor) {
        this.creditCardProcessor = creditCardProcessor;
    }

    @NotNull
    @Size(min=1)
    public Set<CreditCardProcessor> getCreditCardProcessors() { [...] }

    @NotNull
    @Future
    public Date getNextAvailableDeliveryDate() { [...] }
}
```

Here the following postconditions are defined which are guaranteed to the caller of the methods and constructors of the `OrderService` class:

- The newly created `OrderService` object returned by the first constructor satisfies the conditions of the custom `@ValidOnlineOrderService` constraint.
- The newly created `OrderService` object returned by the second constructor satisfies the conditions of the custom `@ValidBatchOrderService` constraint.
- The set of `CreditCardProcessor` objects returned by `getCreditCardProcessors()` will neither be null nor be empty.
- The `Date` object returned by `getNextAvailableDeliveryDate()` will not be null and will be in the future.

Like parameter constraints, these return value constraints are not per-se validated upon method invocation, but instead an integration layer invoking the validation is required.

5.6.4. Marking parameters and return values for cascaded validation

The `@Valid` annotation is used to declare that a cascaded validation of the given method/constructor parameters or return values is performed by the Jakarta Validation provider. When marked, the parameter or return value is considered a bean object to validate. The same rules as for standard object graph validation (see Object graph validation) apply, in particular

- null arguments and null return values are ignored
- The validation is recursive; that is, if validated parameter or return value objects have references marked with `@Valid` themselves, these references will also be validated
- Jakarta Validation providers must guarantee the prevention of infinite loops during cascaded validation

Example 5.23: Marking parameters and return values for cascaded validation

```
public class OrderService {

    @NotNull @Valid
    private CreditCardProcessor creditCardProcessor;

    @Valid
    public OrderService(@NotNull @Valid CreditCardProcessor creditCardProcessor) {
        this.creditCardProcessor = creditCardProcessor;
    }

    @NotNull @Valid
    public Order getOrderByPk(@NotNull @Valid OrderPK orderPk) { [...] }

    @NotNull
```

```
    public Set<@Valid Order> getOrdersByCustomer(@NotNull @Valid CustomerPK customerPk) { [...] }
}
```

Here the following recursive validations will happen when validating the methods of the `OrderService` class:

- Validation of the constraints on the object passed for the `creditCardProcessor` parameter of the constructor
- Validation of the constraints on the newly created `OrderService` instance returned by the constructor, i.e. the `@NotNull` constraint on the field `creditCardProcessor` and the constraints on the referenced `CreditCardProcessor` instance (as the field is annotated with `@Valid`).
- Validation of the constraints on the object passed for the `orderPk` parameter and the returned `Order` object of the `getOrderByPk()` method
- Validation of the constraints on the object passed for the `customerPk` parameter and the constraints on each object contained within the returned `Set<Order>` of the `getOrdersByCustomer()` method

Again, solely marking parameters and return values for cascaded validation does not trigger the actual validation.

5.6.5. Method constraints in inheritance hierarchies

When defining method constraints within inheritance hierarchies (that is, class inheritance by extending base classes and interface inheritance by implementing or extending interfaces) one has to obey the Liskov substitution principle which mandates that:

- a method's preconditions (as represented by parameter constraints) must not be strengthened in sub types
- a method's postconditions (as represented by return value constraints) must not be weakened in sub types

TIP

> Very informally speaking, the Liskov substitution principle says that where a given type T is used, it should be possible to replace T with a sub-type S of T ("Behavioral subtyping"). If S overrides/implements a method from T and S would strengthen the method's preconditions (e.g. by adding parameter constraints) this principle would be violated as client code working correctly against T might fail when working against S. Also if S overrides/implements a method from T and S weakens the method's postconditions this principle would be violated. However S may strengthen the method's postconditions (by adding return value constraints), as client code working against T still will work against S.

Therefore the following rules with respect to the definition of method level constraints in inheritance hierarchies apply:

- In sub types (be it sub classes/interfaces or interface implementations), no parameter constraints may be declared on overridden or implemented methods, nor may parameters be marked for cascaded validation. This would pose a strengthening of preconditions to be fulfilled by the caller.
- If a sub type overrides/implements a method originally defined in several parallel types of the hierarchy (e.g. two interfaces not extending each other, or a class and an interface not implemented by said class), no parameter constraints may be declared for that method at all nor parameters be marked for cascaded validation. This again is to avoid an unexpected strengthening of preconditions to be fulfilled by the caller.
- In sub types (be it sub classes/interfaces or interface implementations), return value constraints may be declared on overridden or implemented methods and the return value may be marked for cascaded validation. Upon validation, all return value constraints of the method in question are validated, wherever they are declared in the hierarchy. This only poses possibly a strengthening but no weakening of the method's postconditions guaranteed to the caller.
- One must not mark a method return value for cascaded validation more than once in a line of a class hierarchy. In other words, overriding methods on sub types (be it sub classes/interfaces or interface implementations) cannot mark the return value for cascaded validation if the return value has already been marked on the overridden method of the super type or interface.

Out of the box, a conforming Jakarta Validation provider must throw a `ConstraintDeclarationException` when discovering that any of these rules are violated. In addition providers may implement alternative, potentially more liberal, approaches for handling constrained methods in inheritance hierarchies. Possible means for activating such alternative behavior include provider-specific configuration properties or annotations. Note that client code relying on such alternative behavior is not portable between Jakarta Validation providers.

The above rules do not apply when validating constructor constraints as constructors do not override one another. Parameter and return value constraints can be applied to any constructor in the type hierarchy, but only the constraints defined directly on the validated constructor are evaluated.

5.6.5.1. Examples

This sections provides some examples of illegal constraint definitions which violate the rules stated above in one way or another.

Example 5.24: Illegally declared parameter constraints on interface implementation

```
public interface OrderService {
```

```
    void placeOrder(String customerCode, Item item, int quantity);
}

public class SimpleOrderService implements OrderService {

    @Override
    public void placeOrder(
        @NotNull @Size(min=3, max=20) String customerCode,
        @NotNull Item item,
        @Min(1) int quantity) {
        [...]
    }
}
```

The constraints in `SimpleOrderService` are illegal, as they strengthen the preconditions of the `placeOrder()` method as constituted by the interface `OrderService`.

Example 5.25: Illegally declared parameter constraints on sub class

```
public class OrderService {

    void placeOrder(String customerCode, Item item, int quantity) { [...] }
}

public class SimpleOrderService extends OrderService {

    @Override
    public void placeOrder(
        @NotNull @Size(min=3, max=20) String customerCode,
        @NotNull Item item,
        @Min(1) int quantity) {
        [...]
    }
}
```

The constraints in `SimpleOrderService` are illegal, as they strengthen the preconditions of the `placeOrder()` method as constituted by the super class `OrderService`.

Example 5.26: Illegally declared parameter constraints on parallel types

```
public interface OrderService {

    void placeOrder(String customerCode, Item item, int quantity);
}

public interface OrderPlacementService {

    public void placeOrder(
        @NotNull @Size(min=3, max=20) String customerCode,
        @NotNull Item item,
        @Min(1) int quantity);
}

public class SimpleOrderService implements OrderService, OrderPlacementService {

    @Override
    public void placeOrder(String customerCode, Item item, int quantity) {
        [...]
    }
}
```

Here the class `SimpleOrderService` implements the interfaces `OrderService` and `OrderPlacementService`, which themselves are unrelated to each other but both define a method `placeOrder()` with an identical signature. This hierarchy is illegal with respect to the parameter constraints as a client of `SimpleOrderService` would have to fulfill the constraints defined on the interface `OrderPlacementService` even if

the client only expects `OrderService`.

Example 5.27: Correctly declared return value constraints on sub class

```
public class OrderService {

    Order placeOrder(String customerCode, Item item, int quantity) {
        [...]
    }
}

public class SimpleOrderService extends OrderService {

    @Override
    @NotNull
    @Valid
    public Order placeOrder(String customerCode, Item item, int quantity) {
        [...]
    }
}
```

The return value constraints in `DefaultOrderService` are legal, as they strengthen the postconditions of the `placeOrder()` method as constituted by the super class `OrderService` but don't weaken them.

5.7. Validation routine

For a given group, the validation routine applied on a given bean instance is expected to execute the following constraint validations in no particular order:

- for all *reachable* fields, execute all field level validations (including the ones expressed on superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine for a given navigation path (see Object graph validation) as part of a previous group match.
- for all *reachable* getters, execute all getter level validations (including the ones expressed on interfaces and superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine for a given navigation path (see Object graph validation) as part of a previous group match.
- execute all class level validations (including the ones expressed on interfaces and superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine for a given navigation path (see Object graph validation) as part of a previous group match.
- for all *reachable* and *cascadable* associations, execute all cascading validations (see Object graph validation) including the ones expressed on interfaces and superclasses (see Formal group definitions). Note that group conversion can apply (see Group conversion).

Reachable fields, getters and associations as well as cascadable associations are defined in Traversable property.

Note that this implies that a given validation constraint will not be processed more than once per validation per path. Some implementations might even process a single constraint only once across paths provided that they return the expected set of `ConstraintViolation`.

Unless ordered by group sequences, groups can be validated in no particular order. This implies that the validation routine can be run for several groups in the same pass.

The object validation routine is described as such. For each constraint declaration:

- determine for the constraint declaration, the appropriate `ConstraintValidator` to use (see ConstraintValidator resolution algorithm).
- execute the `isValid` operation (from the constraint validation implementation) on the appropriate data (see Constraint validation implementation)
- if `isValid()` returns `true`, continue to the next constraint,
- if `isValid()` returns `false`, the Jakarta Validation provider populates `ConstraintViolation` object(s) according to the rules defined in Constraint validation implementation and appends these objects to the list of constraint violations.

5.7.1. Object graph validation

The `@Valid` annotation on a given association (i.e. object reference or collection, array, `Iterable` of objects), dictates the Jakarta Validation implementation to apply recursively the Jakarta Validation routine on (each of) the associated object(s). This mechanism is recursive: an associated object can itself contain cascaded references.

Null references are ignored.

To prevent infinite loops, the Jakarta Validation implementation must ignore the cascading operation if the associated object instance has already been validated in the current navigation path (starting from the root object). See Object graph limits for an example. A navigation path is defined as a set of `@Valid` associations starting from the root object instance and reaching the associated instance. A given navigation path cannot contain the same instance multiple times (the complete validated object graph can though). See Object graph limits for an example.

NOTE This object graph navigation can lead to multiple validations of the same constraint and the same object instance but the set of constraint validation is deterministic and the algorithm prevents infinite loops.

Example 5.28: Object graph limits

```
#assuming the following object graph

Order -(lines)→ Orderline1
Order -(lines)→ Orderline2
Orderline1 -(order)→ Order
Orderline2 -(order)→ Order
Order -(customer)→ User
Order -(shippingAddress)→ Address1
Order -(billingAddress)→ Address2
Address1 -(inhabitant)→ User
Address2 -(inhabitant)→ User
User -(addresses)→ Address1
User -(addresses)→ Address2

#validation branches are as followed
Order -(lines)→ Orderline1
  - order is ignored: Order is already present in the branch

Order -(lines)→ Orderline2
  - order is ignored: Order is already present in the branch

Order -(customer)→ User -(addresses)→ Address1
  - inhabitant is ignored: User is already present in the branch

Order -(customer)→ User -(addresses)→ Address2
  - inhabitant is ignored: User is already present in the branch

Order -(shippingAddress)→ Address1 -(inhabitant)→ User
  - addresses to Address1 is ignored: Address1 is already present in the branch

Order -(shippingAddress)→ Address1 -(inhabitant)→ User -(addresses)→ Address2
  - inhabitant is ignored: User is already present in the branch

Order -(billingAddress)→ Address2 -(inhabitant)→ User
  - addresses to Address2 is ignored: Address2 is already present in the branch

Order -(billingAddress)→ Address2 -(inhabitant)→ User -(addresses)→ Address1
  - inhabitant is ignored: User is already present in the branch
```

The `ConstraintViolation` objects are built when a failing constraint on an associated object is found. They reflect the path to reach the object from the root validated object (See ConstraintViolation).

`@Valid` is an orthogonal concept to the notion of group. If two groups are in sequence, the first group must pass for all associated objects before the second group is evaluated. Note however that the `Default` group sequence overriding is local to the class it is defined on and is not propagated to the associated objects. The following example illustrates this:

Example 5.29: Class Driver with redefined default group

```
@GroupSequence({ Minimal.class, Driver.class })
public class Driver {
  @Min(value = 18, groups = Minimal.class)
  int age;
```

```
  @AssertTrue
  Boolean passedDrivingTest;

  @Valid
  Car car;

  // setter/getters
}
```

Example 5.30: Class Car with redefined default group

```
@GroupSequence({ Car.class, Later.class })
public class Car {
  @NotNull
  String type;

  @AssertTrue(groups = Later.class)
  Boolean roadWorthy;

  // setter/getters
}
```

Example 5.31: Defining a group sequence

```
@GroupSequence({ Minimal.class, Later.class })
public interface SequencedGroups {
}
```

Example 5.32: Group sequence overriding is not propagated to associated objects

```
Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

Driver driver = new Driver();
driver.setAge(16);
Car porsche = new Car();
driver.setCar(porsche);


Set<ConstraintViolation<Driver>> violations = validator.validate( driver );

assert violations.size() == 2;

violations = validator.validate( driver, SequencedGroups.class );

assert violations.size() == 1;
```

The default group sequence is redefined for the `Driver` as well as `Car`. When the default group is requested via `validator.validate( driver )` the group `Minimal` gets validated in class `Driver`. The constraint will fail since the driver's age in the example is only 16. The constraint on `passedDrivingTest` will not be evaluated due to the redefined default sequence of `Driver`. However, there is one more constraint violation, namely the `@NotNull` on `Car.type`. The reason for this is that the group `Default` gets propagated to `Car` (not `Minimal`). Class `Driver` defines its own group sequence which means that `only @NotNull` on `type` gets evaluated.

In the second call to `validate` the group `SequencedGroups` is requested which defines a sequence of `Minimal` followed by `Later`. In this case there is only one constraint violation. Again `@Min` on `age` fails, but in this case the group `Minimal` gets propagated to `Car` which does not have any constraints defined against this group. Constraints belonging to the group `Later` won't get validated until all constraints belonging to `Minimal` pass.

5.7.2. Method and constructor validation

For a given group, the validation routine applied to validate parameters of a method or constructor is expected to execute the following

constraint validations in no particular order:

- execute all parameter validations (in case of overriding method validation, including the ones expressed on overridden methods of the interfaces and superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine for a given navigation path (see Object graph validation) as part of a previous group match.
- execute all cross parameter validations (in case of overriding method validation, including the ones expressed on overridden methods of the interfaces and superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine for a given navigation path (see Object graph validation) as part of a previous group match.
- for all parameters marked for cascaded validation, execute all cascading validations (see Object graph validation), in case of overriding method validation including the ones expressed on overridden methods of the interfaces and superclasses (see Formal group definitions). Note that group conversion can apply (see Group conversion).

For a given group, the validation routine applied to validate the return value of a method or constructor is expected to execute the following constraint validations in no particular order:

- execute all return value validations (including the ones expressed on interfaces and superclasses) matching the targeted group unless the given validation constraint has already been processed during this validation routine for a given navigation path (see Object graph validation) as part of a previous group match.
- if the return value is marked for cascaded validation, execute all cascading validations (see Object graph validation) including the ones expressed on interfaces and superclasses (see Formal group definitions). Note that group conversion can apply (see Group conversion).

Note that this implies that a given validation constraint will not be processed more than once per validation per path. Some implementations might even process a single constraint only once across paths provided that they return the expected set of `ConstraintViolation`.

Unless ordered by group sequences, groups can be validated in no particular order. This implies that the validation routine can be run for several groups in the same pass.

The object validation routine is as defined in described in Validation routine.

5.7.3. Traversable property

In some cases, the state of some properties should not be accessed. For example, if a property loaded by a Jakarta Persistence provider is a lazy property or a lazy association, accessing its state would trigger a load from the database. An undesired behavior.

Jakarta Validation offers a way to control which property can and cannot be accessed via the `TraversableResolver.isReachable()` contract.

Likewise, it is sometimes undesirable to cascade validation despite the use of `@Valid`. Jakarta Persistence 2 for example does not cascade to associated entities during flush. You can control this behavior by implementing `TraversableResolver.isCascadable()`.

Listing 5.7: `TraversableResolver` *interface*

```
/**
 * Contract determining if a property can be accessed by the Jakarta Validation provider.
 * This contract is called for each property that is being either validated or cascaded.
 * <p>
 * A traversable resolver implementation must be thread-safe.
 *
 * @author Emmanuel Bernard
 */
public interface TraversableResolver {
    /**
     * Determines if the Jakarta Validation provider is allowed to reach the property state.
     *
     * @param traversableObject object hosting {@code traversableProperty}
     *        or {@code null} if {@code validateValue} is called
     * @param traversableProperty the traversable property
     * @param rootBeanType type of the root object passed to the Validator
     *        or hosting the method or constructor validated
     * @param pathToTraversableObject path from the root object to
     *        {@code traversableObject}
     *        (using the path specification defined by Validation)
     * @param elementType either {@code FIELD} or {@code METHOD}
     * @return {@code true} if the Jakarta Validation provider is allowed to
     *         reach the property state, {@code false} otherwise
     */
    boolean isReachable(Object traversableObject,
                        Node traversableProperty,
                        Class<?> rootBeanType,
```

```
                        Path pathToTraversableObject,
                        ElementType elementType);

    /**
     * Determines if the Jakarta Validation provider is allowed to cascade validation on
     * the bean instance returned by the property value
     * marked as {@code @Valid}.
     * <p>
     * Note that this method is called only if
     * {@link #isReachable(Object, jakarta.validation.Path.Node, Class, Path, java.lang.annotation.ElementType)}
     * returns {@code true} for the same set of arguments and if the property
     * is marked as {@link Valid}.
     *
     * @param traversableObject object hosting {@code traversableProperty}
     *        or {@code null} if {@code validateValue} is called
     * @param traversableProperty the traversable property
     * @param rootBeanType type of the root object passed to the Validator
     *        or hosting the method or constructor validated
     * @param pathToTraversableObject path from the root object to
     *        {@code traversableObject}
     *        (using the path specification defined by Validation)
     * @param elementType either {@code FIELD} or {@code METHOD}
     * @return {@code true} if the Jakarta Validation provider is allowed to
     *         cascade validation, {@code false} otherwise
     */
    boolean isCascadable(Object traversableObject,
                         Node traversableProperty,
                         Class<?> rootBeanType,
                         Path pathToTraversableObject,
                         ElementType elementType);
}
```

`isReachable()` is called for every property about to be accessed either for validation or for cascading. A property is *reachable* if this method returns `true`.

`isCascadable()` is called for every property about to be cascaded (i.e. marked as `@Valid`). A property is *cascadable* if it is reachable and if the `isCascadable` method returns `true`.

> **NOTE** `isCascadable()` for a given property is only called if `isReachable()` returns `true`. In other words, `isReachable()` is always called before `isCascadable()` for a given property.

`traversableObject` is the object instance being evaluated. `null` if the check is triggered as part of a `validateValue()` call.

`traversableProperty` is the `Node` representing the property hosted by the `traversableObject` being considered for traversal. The name of a property is defined in Field and property validation.

`rootBeanType` is the class of the root being validated, i.e. either the type of the object passed to the `validate` method or the type declaring the validated method/constructor in case of method validation.

`pathToTraversableObject` is the `Path` from the `rootBeanType` down to the `traversableObject`. If the root object is `traversableObject`, `pathToTraversableObject` is composed of a single Node whose name is `null`. The path is described following the conventions described in ConstraintViolation (`getPropertyPath`).

`elementType` is the `java.lang.annotation.ElementType` the annotation is placed on. It can be either `FIELD` or `METHOD`. Any other value is not expected.

The Jakarta Validation provider must not access the state of a property, nor validate its constraints if the property is not traversable. A property is traversable if `TraversableResolver` returns `true` for this property.

If an exception occurs when the `TraversableResolver` is called, the exception is wrapped into a `ValidationException`.

The following elements are not passed through the traversable resolver filter:

- the bean instance validated
- the method and constructor parameter values being validated
- the method and constructor return value being validated

But the properties of these elements (if validated) are. In this case the complete path is provided via `pathToTraversableObject`.

The traversable resolver used by default by a Jakarta Validation provider behaves as followed:

- if Jakarta Persistence is available in the runtime environment, a property is considered reachable if Jakarta Persistence considers the property as loaded. A typical implementation will use `Persistence.getPersistenceUtil().isLoaded(Object, String)` to implement such contract.
- if Jakarta Persistence is not available in the runtime environment, all properties are considered reachable.
- all properties are considered cascadable.

An example implementation of such a resolver is shown in Jakarta Persistence aware TraversableResolver.

Example 5.33: Jakarta Persistence aware TraversableResolver

```
public class JPATraversableResolver implements TraversableResolver {

    public boolean isReachable(Object traversableObject,
                               Path.Node traversableProperty,
                               Class<?> rootBeanType,
                               Path pathToTraversableObject,
                               ElementType elementType) {
        return traversableObject == null ||
                Persistence.getPersistenceUtil().isLoaded(
                        traversableObject,
                        traversableProperty.getName() );
    }

    public boolean isCascadable(Object traversableObject,
                               Path.Node traversableProperty,
                               Class<?> rootBeanType,
                               Path pathToTraversableObject,
                               ElementType elementType) {
        return true;
    }
}
```

See Bootstrapping to learn how to pass a custom `TraversableResolver`.

5.7.3.1. Examples

The following example assumes the object graph defined in Definitions used in the example and assumes the validation operation is applied on an address object.

Example 5.34: Definitions used in the example

```
public class Country {
    @NotNull
    private String name;
    @Size(max=2)
    private String ISO2Code;
    @Size(max=3)
    private String ISO3Code;

    public String getName() {
        return name;
    }

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

    public String getISO2Code() {
        return ISO2Code;
    }

    public void setISO2Code(String ISO2Code) {
        this.ISO2Code = ISO2Code;
    }
```

```
    public String getISO3Code() {
        return ISO3Code;
    }

    public void setISO3Code(String ISO3Code) {
        this.ISO3Code = ISO3Code;
    }
}

public class Address {
    @NotNull @Size(max=30)
    private String addressline1;
    @Size(max=30)
    private String addressline2;
    @Size(max=11)
    private String zipCode;
    @Valid
    private Country country;

    private String city;

    public String getAddressline1() {
        return addressline1;
    }

    public void setAddressline1(String addressline1) {
        this.addressline1 = addressline1;
    }

    public String getAddressline2() {
        return addressline2;
    }

    public void setAddressline2(String addressline2) {
        this.addressline2 = addressline2;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    @Size(max=30) @NotNull
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public Country getCountry() {
        return country;
    }

    public void setCountry(Country country) {
        this.country = country;
    }
}
```

When the Jakarta Validation provider is about to check constraints of `ISO3Code`, it calls the `TraversableResolver.isReachable()` method to ensure that the `ISO3Code` property is reachable with the following parameter values:

- `traversableObject`: country. The instance returned by `address.getCountry()`.
- `traversableProperty`: a `PropertyNode` whose name is "ISO3Code". Represents the property of `traversableObject` being verified.

- `rootBeanType`: `Address.class`. The type of the root object being validated.
- `pathtoTraversableObject`: a `Path` containing a single `PropertyNode` whose name is "country". The path from address to the country instance.
- `elementType`: `ElementType.FIELD`. The ISO3Code property is annotated on its field.

When the Jakarta Validation provider is about to cascade validation on `country` (`Address` object), it calls the `TraversableResolver.isReachable()` method to ensure that the `country` property is reachable and if this method returns `true`, it calls `TraversableResolver.isCascadable()` with the following parameter values:

- `traversableObject`: address. The address instance.
- `traversableProperty`: a `PropertyNode` whose name is "country". Represents the property of `traversableObject` being verified.
- `rootBeanType`: `Address.class`. The type of the root object being validated.
- `pathtoTraversableObject`: a `Path` containing a single `BeanNode` whose name is `null`.
- `elementType`: `ElementType.FIELD`. The country property is annotated on its field.

The following example shows invocations of the `TraversableResolver` as to be performed by the Jakarta Validation provider during method validation. The example is based on the object graph defined in Definitions used in the example and the `AddressService` class shown in Examplary class `AddressService`. It assumes that a call of `persistAddress()` is subject to method parameter validation.

Example 5.35: Examplary class `AddressService`

```
public class AddressService {
    public void persistAddress(@NotNull @Valid Address address) {
        [...]
    }
}
```

When the Jakarta Validation provider is about to validate the `@NotNull` constraint on the `address` parameter, no call to `isReachable()` is expected, since parameters are assumed to always be reachable. Similarly, no call to `isCascadable()` is expected when performing cascaded validation of the `address` parameter, since parameters are assumed to always be cascadable.

When the Jakarta Validation provider is about to validate constraints on the field `addressline1` of the passed `Address` object, it calls the `isReachable()` method to ensure that the property is reachable with the following parameter values:

- `traversableObject`: address. The instance passed to `persistAddress()`.
- `traversableProperty`: a `PropertyNode` whose name is "addressline1". Represents the property of `traversableObject` being verified.
- `rootBeanType`: `AddressService.class`. The type of the root object being validated.
- `pathtoTraversableObject`: a `Path` comprising a `MethodNode` (named "persistService") and a `ParameterNode` (with parameter index 0). The path from `AddressService` to the `Address` instance.
- `elementType`: `ElementType.FIELD`. The `addressline1` property is annotated on its field.

When the Jakarta Validation provider is about to perform a cascaded validation of the `country` property of the passed `Address` object, it calls the `isReachable()` method to ensure that the property is reachable. If this method returns `true`, it calls `TraversableResolver.isCascadable()` with the following parameter values:

- `traversableObject`: address. The instance passed to `persistAddress()`.
- `traversableProperty`: a `PropertyNode` whose name is "country". Represents the property of `traversableObject` being verified.
- `rootBeanType`: `AddressService.class`. The type of the root object being validated.
- `pathtoTraversableObject`: a `Path` comprising a `MethodNode` (named "persistService") and a `ParameterNode` (with parameter index 0). The path from `AddressService` to the `Address` instance.
- `elementType`: `ElementType.FIELD`. The `country` property is annotated on its field.

5.7.4. ConstraintValidator resolution

5.7.4.1. Registering ConstraintValidator implementations

Constraint validators can be registered with the validation engine in the following ways:

- Provided by the validation engine itself (see Built-in constraints)
- Via the Java service loader mechanism; for this the file *META-INF/services/jakarta.validation.ConstraintValidator* must be provided, with the fully-qualified name(s) of one or more constraint validator implementations as its contents.
- Provided in the `validatedBy()` attribute of the `@jakarta.validation.Constraint` annotation.

- Through XML mapping (see Constraint declaration in XML).

If multiple constraint validators for the exactly same constraint and target type are registered via one of the above methods, a `jakarta.validation.ValidationException` is thrown, unless the constraint validator configuration source allows specifying that it overrides any previously defined constraint validators for the particular constraint (e.g. Class-level overriding/Constructor-level overriding/Field-level overriding/Method-level overriding/Property-level overriding).

NOTE Jakarta Validation providers may have other means to register and/or override constraint validators.

5.7.4.2. ConstraintValidator resolution algorithm

A constraint is associated to one or more `ConstraintValidator` implementations. Each `ConstraintValidator<A, T>` accepts the type `T`. The `ConstraintValidator` executed depends on the type hosting the constraint. For a given constraint evaluation, a single `ConstraintValidator` is considered.

The list of `ConstraintValidator`s can contain at most one which targets cross-parameter. If the constraint targets the parameters of an executable either implicitly or by the use of `validationAppliesTo` in the constraint - see validationAppliesTo, then the cross-parameter `ConstraintValidator` is used. If none is present, a `ConstraintDefinitionException` is raised. If more than one cross-parameter `ConstraintValidator` is present, a `ConstraintDefinitionException` is raised.

If the constraint is a generic constraint, the following rules apply:

- If the constraint declaration is hosted on a class or an interface, the targeted type is the class or the interface.
- If the constraint is hosted on a class attribute, the type of the attribute is the targeted type.
- If the constraint is hosted on a method (getter or non-getter) or constructor, the return type is the targeted type.
- If the constraint is hosted on a method or constructor parameter, the parameter type is the targeted type.
- If the constraint is hosted on a type argument of a parameterized type (i.e. a container element constraint, see Container element constraints), the type argument's type is the targeted type.
- If the constraint is subject to implicit unwrapping (see Implicit unwrapping of containers) and the applicable value extractor is defined for a generic type (e.g. `javafx.beans.value.ObservableValue`), the targeted type is the type captured for the type parameter handled by the value extractor (e.g. `String` if the constraint is placed on a `StringProperty`).
- If the constraint is subject to implicit unwrapping and the applicable value extractor is defined for a non-generic type, the targeted type is the type defined by the extractor via `@ExtractedValue#type()` (e.g. `Integer` if the constraint is placed on a `java.util.OptionalInt`).

In other words, the resolution algorithm considers the type as defined in the source code and not the runtime type of the value.

The rules written below describe formally the following statement: the `ConstraintValidator` chosen to validate the generic constraint on a declared type `T` is the one where the `ConstraintValidator` targets the annotated element, where the type supported by the `ConstraintValidator` is a supertype of `T` and where there is no other `ConstraintValidator` whose supported type is a supertype of `T` and not a supertype of the chosen `ConstraintValidator` supported type.

When validating a generic constraint `A` placed on a target declaring the type `T`, the following resolution rules apply:

- Only `ConstraintValidator` implementations targeting annotated elements are considered.
- Primitive types are considered equivalent to their respective primitive wrapper class.
- A `ConstraintValidator<A, U>` is said to be *compliant* with `T` if `T` is a subtype of `U` (according to the Java Language Specification, Java SE 8 Edition, chapter 4.10, "Subtyping"). Note that `T` is a subtype of `U` if `T = U`.
- If no `ConstraintValidator` compliant with `T` is found among the `ConstraintValidator`s listed by the constraint `A`, an `UnexpectedTypeException` is raised.
- A `ConstraintValidator<A, U>` compliant with `T` is considered *strictly more specific* than a `ConstraintValidator<A, V>` compliant with `T` if `U` is a strict subtype of `V`. `U` is a strict subtype of `V` if `U` is a subtype of `V` and `U != V` (according to the Java Language Specification).
- A `ConstraintValidator<A, U>` compliant with `T` is considered maximally specific if no other `ConstraintValidator<A, V>` compliant with `T` is strictly more specific than `ConstraintValidator<A, U>`.
- If more than one maximally specific `ConstraintValidator` is found, an `UnexpectedTypeException` is raised.

NOTE While the Java compiler itself cannot determine if a constraint declaration will lead to a `UnexpectedTypeException`, rules can be statically checked. A tool such as an IDE or an annotation processor can apply these rules and prevent compilation in case of ambiguity. The specification encourages Jakarta Validation providers to provide such a tool to their users.

Let's see a couple of declarations and their respective `ConstraintValidator` resolution. Assuming the definitions shown in ConstraintValidator and type resolution:

Example 5.36: ConstraintValidator and type resolution

```
[...]
```

```
@Constraint(validatedBy={
    SizeValidatorForCollection.class,
    SizeValidatorForSet.class,
    SizeValidatorForSerializable.class })
public @interface Size { [...] }

public class SizeValidatorForCollection implements ConstraintValidator<Size, Collection> {
    [...]
}
public class SizeValidatorForSet implements ConstraintValidator<Size, Set> {
    [...]
}
public class SizeValidatorForSerializable implements ConstraintValidator<Size, Serializable> {
    [...]
}

public interface SerializableCollection extends Serializable, Collection {
}
```

The resolutions shown in Resolution of ConstraintValidator for various constraints declarations occur.

Table 5.1: Resolution of ConstraintValidator for various constraints declarations

Declaration	Resolution
`@Size Collection getAddresses() { […] }`	`SizeValidatorForCollection`: direct match
`@Size Collection<?> getAddresses() { […] }`	`SizeValidatorForCollection`: `Collection` is a direct supertype of `Collection<?>`
`@Size Collection<Address> getAddresses() { […] }`	`SizeValidatorForCollection`: `Collection` is a direct supertype of `Collection<Address>`
`@Size Set<Address> getAddresses() { […] }`	`SizeValidatorForSet`: direct supertype of `Set<Address>`
`@Size SortedSet<Address> getAddresses() { […] }`	`SizeValidatorForSet`: `Set` is the closest supertype of `SortedSet<Address>`
`@Size SerializableCollection getAddresses() { […] }`	`UnexpectedTypeException`: `SerializableCollection` is a subtype of both `Collection` and `Serializable` and neither `Collection` nor `Serializable` are subtypes of each other.
`@Size String getName() { […] }`	`SizeValidatorForSerializable`: `String` is a subtype of `Serializable`.

5.7.5. ValueExtractor resolution

When detecting a container element constraint or a container element marked with `@Valid`, a value extractor must be determined so that the elements can be obtained from that container.

A value extractor handles one container type and - in the case of a generic container type - one type parameter of that container type. The applicable extractor is identified based on the container type and - in the case of a generic container type - the type argument hosting the container element constraint or `@Valid`.

Exactly one value extractor must be identified when processing a container element constraint or container element marked with `@Valid`.

5.7.5.1. Registering ValueExtractor implementations

Value extractors can be registered with the validation engine in the following ways (in increasing order of priority):

- Provided by the validation engine itself (see Built-in value extractors)
- Via the Java service loader mechanism; for this the file *META-INF/services/jakarta.validation.valueextraction.ValueExtractor* must be provided, with the fully-qualified name(s) of one or more extractor implementations as its contents. It is undefined which value extractor will be selected if multiple extractors for the same type and type parameter are registered via the service loader mechanism.
- By specifying the fully-qualified class name of one or several extractors in `META-INF/validation.xml` (see XML configuration: META-INF/validation.xml). Not more than one extractor for the same type and type parameter may be given.
- By invoking the method `Configuration#addValueExtractor(ValueExtractor<?>)` (to apply it at the validator factory level). Not more

than one extractor for the same type and type parameter may be passed.
- By invoking the method `ValidatorContext#addValueExtractor(ValueExtractor<?>)` (to apply it for a single `Validator` instance). Not more than one extractor for the same type and type parameter may be passed.

A value extractor for a given type and type parameter specified at a higher priority overrides any other extractors for the same type and type parameter given at lower priorities. If e.g. a value extractor defined as `class MyListValueExtractor implements ValueExtractor<List<@ExtractedValue ?>> { … }` is given via `ValidatorContext#addValueExtractor(ValueExtractor<?>)`, it will take precedence over any other value extractor implementing `List<@ExtractedValue ?>` given via `Configuration#addValueExtractor(ValueExtractor<?>)`, *META-INF/validation.xml* or the service loader mechanism as well as the built-in extractor for `List` values.

5.7.5.2. ValueExtractor resolution algorithm for container element constraints

For a container with the declared type `C` whose element type is hosting a constraint, the following resolution rules apply for identifying the value extractor:

- A `ValueExtractor<T>` is said to be *type-compliant* with `C`, if `C` is a subtype of `T` (according to the Java Language Specification, Java SE 8 Edition, chapter 4.10, "Subtyping"). Note that `C` is a subtype of `T` if `C = T`.
- A `ValueExtractor` implementation is said to be *container-element-compliant* with `C`, if `C` is a generic container type and the value extractor implementation handles a type parameter that maps to the constrained type argument.
- If no `ValueExtractor` type-compliant and container-element-compliant with `C` is found among the available value extractors, a `ConstraintDeclarationException` is raised.
- A `ValueExtractor<U>` type-compliant with `C` is considered *strictly more specific* than a `ValueExtractor<V>` compliant with `C` if `U` is a strict subtype of `V`. `U` is a strict subtype of `V` if `U` is a subtype of `V` and `U != V`.
- A `ValueExtractor<U>` type-compliant with `C` is considered maximally specific if no other `ValueExtractor<V>` type-compliant with `C` is strictly more specific than `ValueExtractor<U>`.
- If more than one maximally specific and container-element-compliant `ValueExtractor` is found, a `ConstraintDeclarationException` is raised.

NOTE

Implementation note

Extractor retrieval for container element constraints is based on the declared type of constrained elements, hence it is recommended that implementations perform the resolution once and then cache the value extractor for a given constraint.

5.7.5.3. ValueExtractor resolution algorithm for cascaded validation

For a container with the declared type `C` and the runtime type `C'` whose element type is marked for cascaded validation, the following resolution rules apply for identifying the value extractor:

- A `ValueExtractor<T>` is said to be *type-compliant* with `C'`, if `C'` is a subtype of `T`. Note that `C'` is a subtype of `T` if `C' = T`.
- A `ValueExtractor` implementation is said to be *container-element-compliant* with `C`, if it handles a type parameter that maps to the type argument marked with `@Valid`.
- If no `ValueExtractor` type-compliant with `C'` and container-element-compliant with `C` is found among the available value extractors, a `ConstraintDeclarationException` is raised.
- A `ValueExtractor<U>` type-compliant with `C'` is considered *strictly more specific* than a `ValueExtractor<V>` compliant with `C'` if `U` is a strict subtype of `V`. `U` is a strict subtype of `V` if `U` is a subtype of `V` and `U != V`.
- A `ValueExtractor<U>` type-compliant with `C'` is considered maximally specific if no other `ValueExtractor<V>` type-compliant with `C'` is strictly more specific than `ValueExtractor<U>`.
- If more than one maximally specific and container-element-compliant `ValueExtractor` is found, a `ConstraintDeclarationException` is raised.

NOTE

Implementation note

Extractor retrieval for cascaded validation is based on the runtime type of the container hosting `@Valid` (this is to be consistent with the semantics of property paths for cascaded validation), hence it is not possible to perform the extractor resolution only once per usage of `@Valid` and cache the result.

5.7.5.4. ValueExtractor resolution algorithm for applying container-level constraints to container elements

When detecting a constraint given as a *declaration annotation* and not as a *type annotation* (i.e. it is given on field, parameter etc. and not given on a type argument of a parameterized type), the applicable value extractor, if any, is determined as follows:

- If the constraint carries the `Unwrapping.Skip` payload, no value extractor is applied.
- If the constraint carries the `Unwrapping.Unwrap` payload and there is exactly one maximally-specific type-compliant value extractor, this

extractor is applied; if no type-compliant extractor or multiple maximally-specific type-compliant extractors exist, a `ConstraintDeclarationException` is raised.

- If the constraint carries neither the `Unwrapping.Unwrap` nor the `Unwrapping.Skip` payload:
  - If there is exactly one maximally-specific type-compliant value extractor and this extractor is marked with `@UnwrapByDefault`, this extractor is applied;
  - Otherwise, no value extractor is applied.

5.7.5.5. Examples

Let's consider a couple of value extractor definitions and their respective `ValueExtractor` resolution against container element constraint declarations:

Example 5.37: `ValueExtractor` *resolution*

```
public interface ConcurrentList<T> {
    [...]
}

public class MyList<T> implements List<T>, ConcurrentList<T> {
    [...]
}

public interface Table<R, C, V> {
    [...]
}

interface ConfusingMap<K, V> extends Map<V, K> {
    [...]
}

interface SingleTypeMap<T> extends Map<T, T> {
    [...]
}

interface StringMap extends Map<String, String> {
    [...]
}

interface Property<T> {
    [...]
}

class StringProperty implements Property<String> {
    [...]
}

public class IterableValueExtractor implements ValueExtractor<Iterable<@ExtractedValue ?>> {
    [...]
}

public class ListValueExtractor implements ValueExtractor<List<@ExtractedValue ?>> {
    [...]
}

public class ConcurrentListValueExtractor implements ValueExtractor<
        ConcurrentList<@ExtractedValue ?>> {

    [...]
}

public class MapKeyExtractor implements ValueExtractor<Map<@ExtractedValue ?, ?>> {
    [...]
}

public class MapValueExtractor implements ValueExtractor<Map<?, @ExtractedValue ?>> {
    [...]
}
```

```
public class TableValueExtractor implements ValueExtractor<Table<?, ?, @ExtractedValue ?>> {
    [...]
}

@UnwrapByDefault
public class PropertyValueExtractor implements ValueExtractor<Property<@ExtractedValue ?>> {
    [...]
}

@UnwrapByDefault
public class OptionalIntValueExtractor implements ValueExtractor<
        @ExtractedValue(type = Integer.class) OptionalInt> {

    [...]
}
```

The following value extractor resolutions occur:

Table 5.2: Resolution of `ValueExtractor` for various container element constraints

Declaration	Resolution
`List<@Email String> emails`	`ListValueExtractor`; strictly more specific than `IterableValueExtractor`
`Iterable<@Valid Address> addresses = new ArrayList<>()`	`ListValueExtractor` (the runtime type `ArrayList` is used for resolving value extractors for cascaded validation)
`Map<@Email String, String> emails`	`MapKeyExtractor`; equally specific as `MapValueExtractor`, but the latter isn't container-element-compliant
`ConfusingMap<@Email String, String> map`	`MapValueExtractor`; the constrained type argument maps to the type parameter `V` of `Map`
`@Email StringProperty`	`PropertyValueExtractor`; the extractor is marked with `@UnwrapByDefault`, i.e. implicit unwrapping is performed; `String` will be used for validator resolution as that's the type captured for the type parameter handled by the applied value extractor
`@Min(1) OptionalInt`	`OptionalIntValueExtractor`; the extractor is marked with `@UnwrapByDefault`, i.e. implicit unwrapping is performed; `Integer` as given via `@ExtractedValue#type()` will be used for validator resolution
`Optional<@Email String> getEmail() {…}`	`ConstraintDeclarationException`; no compliant extractor exists
`Table<@Min(1) String, String, String> table`	`ConstraintDeclarationException`; `TableValueExtractor` is type-compliant but not container-element-compliant
`MyList<@Email String> emails`	`ConstraintDeclarationException`; `ListValueExtractor` and `ConcurrentListValueExtractor` are equally specific
`SingleTypeMap<@NotEmpty String> map`	`ConstraintDeclarationException`; `MapKeyExtractor` and `MapValueExtractor` are equally specific and both container-element-compliant
`@NotEmpty(payload=Unwrapping.Unwrap.class) StringMap`	`ConstraintDeclarationException`; more than one maximally-specific extractor is found (`MapKeyExtractor` and `MapValueExtractor`)

5.8. Examples

The first example demonstrates how beans, fields and getters are annotated to express some constraints.

Example 5.38: Place constraint declarations on the element to validate

```
@ZipCodeCityCoherenceChecker
public class Address {
```

```
    @NotNull @Size(max=30)
    private String addressline1;

    @Size(max=30)
    private String addressline2;

    private String zipCode;

    private String city;

    public String getAddressline1() {
        return addressline1;
    }

    public void setAddressline1(String addressline1) {
        this.addressline1 = addressline1;
    }

    public String getAddressline2() {
        return addressline2;
    }

    public void setAddressline2(String addressline2) {
        this.addressline2 = addressline2;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    @Size(max=30) @NotNull
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}
```

During the validation routine execution on an `Address` object,

- `addressline1` field value is passed to the `@NotNull` as well as `@Size` constraint validation implementations.
- `addressline2` field value is passed to the `@Size` constraint validation implementation.
- `getCity` value is passed to the `@Size` and `@NotNull` constraint validation implementations.
- `@ZipCodeCoherenceChecker` is a constraint whose validation implementation's `isValid` method receives the `Address` object.

The second example demonstrates object graph validation.

Example 5.39: Define object graph validation

```
public class Country {
    @NotNull
    private String name;
    @Size(max=2)
    private String ISO2Code;
    @Size(max=3)
    private String ISO3Code;

    public String getName() {
        return name;
    }
```

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

    public String getISO2Code() {
        return ISO2Code;
    }

    public void setISO2Code(String ISO2Code) {
        this.ISO2Code = ISO2Code;
    }

    public String getISO3Code() {
        return ISO3Code;
    }

    public void setISO3Code(String ISO3Code) {
        this.ISO3Code = ISO3Code;
    }
}

public class Address {
    @NotNull @Size(max=30)
    private String addressline1;
    @Size(max=30)
    private String addressline2;
    @Size(max=11)
    private String zipCode;
    @NotNull @Valid
    private Country country;

    private String city;

    public String getAddressline1() {
        return addressline1;
    }

    public void setAddressline1(String addressline1) {
        this.addressline1 = addressline1;
    }

    public String getAddressline2() {
        return addressline2;
    }

    public void setAddressline2(String addressline2) {
        this.addressline2 = addressline2;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    @Size(max=30) @NotNull
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public Country getCountry() {
        return country;
    }
```

```
        public void setCountry(Country country) {
            this.country = country;
        }
    }
```

During the validation routine execution on an `Address` object, constraints on `addressLine1`, `addressLine2`, `zipCode`, `getCity` and `country` are processed as well as the validation of the `Country` object itself, more specifically `country.name` is checked for `@NotNull`, `ISO2Code` and `ISO3Code` are checked for `@Size`.

Assuming that `@NonEmpty` is defined as such:

```
@Documented
@NotNull
@Size(min = 1)
@ReportAsSingleViolation
@Constraint(validatedBy = NonEmpty.NonEmptyValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface NonEmpty {

    String message() default "{com.acme.constraint.NonEmpty.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        NonEmpty[] value();
    }

    class NonEmptyValidator implements ConstraintValidator<NonEmpty, String> {

        @Override
        public boolean isValid(String value, ConstraintValidatorContext context) {
            return true;
        }
    }
}
```

The third example demonstrates superclass, inheritance and composite constraints.

Example 5.40: Use inheritance, constraints on superclasses and composite constraints

```
    public interface Person {
        @NonEmpty
        String getFirstName();

        String getMiddleName();

        @NonEmpty
        String getLastName();
    }

    public class Customer implements Person {
        private String firstName;
        private String middleName;
        private String lastName;
        @NotNull
        private String customerId;
        @Password(robustness=5)
        private String password;

        public String getFirstName() {
            return firstName;
```

```
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getMiddleName() {
        return middleName;
    }

    public void setMiddleName(String middleName) {
        this.middleName = middleName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCustomerId() {
        return customerId;
    }

    public void setCustomerId(String customerId) {
        this.customerId = customerId;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}

public class PreferredGuest extends Customer {
    @CreditCard
    private String guestCreditCardNumber;

    public String getGuestCreditCardNumber() {
        return guestCreditCardNumber;
    }

    public void setGuestCreditCardNumber(String guestCreditCardNumber) {
        this.guestCreditCardNumber = guestCreditCardNumber;
    }
}

public class CommonGuest extends customer {}
```

When validating a `PreferredGuest` the following constraints are processed:

- `@NonEmpty`, `@NotNull` and `@Size(min=1)` on `firstName`
- `@NonEmpty`, `@NotNull` and `@Size(min=1)` on `lastName`
- `@NotNull` on `customerId`, `@Password` on `password`
- `@CreditCard` on `guestCreditCardNumber`

When validating `CommonGuest`, the following constraints are processed:

- `@NonEmpty`, `@NotNull` and `@Size(min=1)` on `firstName`
- `@NonEmpty`, `@NotNull` and `@Size(min=1)` on `lastName`
- `@NotNull` on `customerId`, `@Password` on `password`

The fourth example demonstrates the influence of group sequence.

Example 5.41: Use groups and group sequence to define constraint ordering

```
@GroupSequence({First.class, Second.class, Last.class})
public interface Complete {}

public class Book {
    @NonEmpty(groups=First.class)
    private String title;

    @Size(max=30, groups=Second.class)
    private String subtitle;

    @Valid
    @NotNull(groups=First.class)
    private Author author;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getSubtitle() {
        return subtitle;
    }

    public void setSubtitle(String subtitle) {
        this.subtitle = subtitle;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }
}

public class Author {
    @NonEmpty(groups=Last.class)
    private String firstName;

    @NonEmpty(groups=First.class)
    private String lastName;

    @Size(max=30, groups=Last.class)
    private String company;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCompany() {
        return company;
```

```
    }

    public void setCompany(String company) {
        this.company = company;
    }
}
```

Assuming the validation of the `Complete` group on the following book instance:

```
Author author = new Author();
author.setLastName( "Baudelaire" );
author.setFirstName( "" );
Book book = new Book();
book.setAuthor( author );
```

the validation routine will return the following failure:

- `@NotNull` failure (from `@NonEmpty`) on the `title` field

As both `title` and `author.lastname` are checked as part of the `First` group. If the instance is updated:

```
book.setTitle( "Les fleurs du mal" );
author.setCompany( "Some random publisher with a very very very long name" );
```

the validation routine will return the following failures:

- `author.firstName` fails to pass the `@Size(min=1)` (from `@NonEmpty`) constraint
- `author.company` fails to pass the `@Size` constraint

As the `First` and `Second` groups pass without failure, the `Last` group is going through validation.

Java records are supported by Jakarta Validation. The following examples demonstrate how to use records with Jakarta Validation constraints.

Example 5.42: Example record constuctor parameter constraint and constructor return value constraint

```
@Target({ METHOD, CONSTRUCTOR, ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = ValidStockItemRecord.Validator.class)
@Documented
public @interface ValidStockItemRecord {
    String message() default "{ValidStockItem.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    public class Validator implements ConstraintValidator<ValidStockItemRecord, StockItemRecord> {

        @Override
        public boolean isValid(StockItemRecord object, ConstraintValidatorContext context) {
            return false;
        }
    }
}

public record StockItemRecord(@NotNull String name) {
    @ValidStockItemRecord
    public StockItemRecord {
    }
}
```

Here the constructor parameter constraint is declared on the record components list and the return value constraint is declared on the compact constructor.

Example 5.43: Example record cross-parameter constraint with parameter constraints

```
@Target({ METHOD, CONSTRUCTOR, ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = MyCrossParameterConstraintValidator.class)
@Documented
public @interface MyCrossParameterConstraint {
    String message() default "{MyCrossParameterConstraint.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    @Target({ METHOD, CONSTRUCTOR, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        MyCrossParameterConstraint[] value();
    }
}

public record ComplexStockItemRecord(@NotNull String name, @NotNull String description) {
    @MyCrossParameterConstraint
    public ComplexStockItemRecord {
    }
}
```

Here the constructor parameter constraints are declared on the record components list and the cross-parameter constraint is declared on the compact constructor.

6. Validation APIs

The default package for the Jakarta Validation APIs is `jakarta.validation`.

6.1. Validator API

The main Jakarta Validation API is the `jakarta.validation.Validator` interface.

A `Validator` instance is able to validate instances of beans and their associated objects if any. It is recommended to leave the caching of `Validator` instances to the `ValidatorFactory`. `Validator` implementations must be thread-safe.

Listing 6.1: `Validator` *interface*

```
/**
 * Validates bean instances. Implementations of this interface must be thread-safe.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 * @author Gunnar Morling
 */
public interface Validator {

    /**
     * Validates all constraints on {@code object}.
     *
     * @param object object to validate
     * @param groups the group or list of groups targeted for validation (defaults to
     *        {@link Default})
     * @param <T> the type of the object to validate
     * @return constraint violations or an empty set if none
     * @throws IllegalArgumentException if object is {@code null}
     *         or if {@code null} is passed to the varargs groups
     * @throws ValidationException if a non recoverable error happens
     *         during the validation process
     */
    <T> Set<ConstraintViolation<T>> validate(T object, Class<?>... groups);

    /**
     * Validates all constraints placed on the property of {@code object}
     * named {@code propertyName}.
     *
     * @param object object to validate
     * @param propertyName property to validate (i.e. field and getter constraints)
     * @param groups the group or list of groups targeted for validation (defaults to
     *        {@link Default})
     * @param <T> the type of the object to validate
     * @return constraint violations or an empty set if none
     * @throws IllegalArgumentException if {@code object} is {@code null},
     *         if {@code propertyName} is {@code null}, empty or not a valid object property
     *         or if {@code null} is passed to the varargs groups
     * @throws ValidationException if a non recoverable error happens
     *         during the validation process
     */
    <T> Set<ConstraintViolation<T>> validateProperty(T object,
                                                     String propertyName,
                                                     Class<?>... groups);

    /**
     * Validates all constraints placed on the property named {@code propertyName}
     * of the class {@code beanType} would the property value be {@code value}.
     * <p>
     * {@link ConstraintViolation} objects return {@code null} for
     * {@link ConstraintViolation#getRootBean()} and
     * {@link ConstraintViolation#getLeafBean()}.
     *
     * @param beanType the bean type
     * @param propertyName property to validate
     * @param value property value to validate
```

```
     * @param groups the group or list of groups targeted for validation (defaults to
     *        {@link Default}).
     * @param <T> the type of the object to validate
     * @return constraint violations or an empty set if none
     * @throws IllegalArgumentException if {@code beanType} is {@code null},
     *         if {@code propertyName} is {@code null}, empty or not a valid object property
     *         or if {@code null} is passed to the varargs groups
     * @throws ValidationException if a non recoverable error happens
     *         during the validation process
     */
    <T> Set<ConstraintViolation<T>> validateValue(Class<T> beanType,
                                                  String propertyName,
                                                  Object value,
                                                  Class<?>... groups);

    /**
     * Returns the descriptor object describing bean constraints.
     * <p>
     * The returned object (and associated objects including
     * {@link ConstraintDescriptor}s) are immutable.
     *
     * @param clazz class or interface type evaluated
     * @return the bean descriptor for the specified class
     * @throws IllegalArgumentException if clazz is {@code null}
     * @throws ValidationException if a non recoverable error happens
     *         during the metadata discovery or if some
     *         constraints are invalid.
     */
    BeanDescriptor getConstraintsForClass(Class<?> clazz);

    /**
     * Returns an instance of the specified type allowing access to
     * provider-specific APIs.
     * <p>
     * If the Jakarta Validation provider implementation does not support
     * the specified class, {@link ValidationException} is thrown.
     *
     * @param type the class of the object to be returned
     * @param <T> the type of the object to be returned
     * @return an instance of the specified class
     * @throws ValidationException if the provider does not support the call
     */
    <T> T unwrap(Class<T> type);

    /**
     * Returns the contract for validating parameters and return values of methods
     * and constructors.
     *
     * @return contract for method and constructor validation
     *
     * @since 1.1
     */
    ExecutableValidator forExecutables();
}
```

The methods `validate()`, `validateProperty()` and `validateValue()` are used for the validation of Java beans respectively single bean properties. See the next section for more details.

`forExecutables()` provides access to the contract for validating method and constructor parameters and return values. The individual methods for method and constructor validation are described in Methods for validating method and constructor constraints.

`getConstraintsForClass()` returns constraint-related metadata for given types and is described in detail in Constraint metadata request APIs.

`unwrap()` is provided as a way to access objects of a given type specific to a Jakarta Validation provider typically as a complement to the `Validator` contract. Using this method makes your code non portable.

Example 6.1: Using unwrap to access a provider specific contract

```
//if using the ACME provider
ACMEValidator acmeValidator = factory.unwrap(ACMEValidator.class);
acmeValidator.setSpecificConfiguration( [...] );
```

6.1.1. Validation methods

`<T> Set<ConstraintViolation<T>> validate(T object, Class<?>... groups)` is used to validate a given object. This method implements the logic described in Validation routine. An `IllegalArgumentException` is thrown when null is passed for the `object` parameter or the varargs `groups` parameter. A `Set` containing all `ConstraintViolation` objects representing the failing constraints is returned, an empty `Set` is returned otherwise.

`<T> Set<ConstraintViolation<T>> validateProperty(T object, String propertyName, Class<?>... groups)` validates a given field or property of an object. An `IllegalArgumentException` is thrown when `validateProperty()` is called and `object` is null or `propertyName` is null, empty or invalid or null is passed to the varargs `groups` parameter. The property name is the JavaBeans property name (as defined by the JavaBeans `Introspector` class). This method implements the logic described in Validation routine but only to the given property. `@Valid` is not honored by this method. This method is useful for partial object validation.

`<T> Set<ConstraintViolation<T>> validateValue(Class<T> beanType, String propertyName, Object value, Class<?>... groups)` validates the property referenced by `propertyName` present on `beanType` or any of its superclasses, if the property value were `value`. An `IllegalArgumentException` is thrown when `validateValue()` is called and `beanType` is null or `propertyName` is null, empty or invalid or null is passed to the varargs `groups` parameter. This method implements the logic described in Validation routine and apply it only to the given property and for the given value. `@Valid` is not honored by this method. This method is useful for ahead of time validation (i.e. before the JavaBean is populated or updated).

NOTE

If multiple constrained fields or getters share the same name and hide one another in the class hierarchy according to the Java visibility rules, the list of constraints evaluated is unspecified. This will be clarified in a later version of this specification. Note that method overriding is not impacted.

If getters and fields share the same name and are present at different levels of the hierarchy, the list of constraints evaluated is unspecified. This will be clarified in a later version of this specification.

However, constraints hosted on the most specific (hierarchy wise) element type are always evaluated.

NOTE

`validateProperty()` and `validateValue()` accept property names and not full paths. Jakarta Validation implementations might accept string representations of paths but this behavior is not portable.

If some unrecoverable failure happens during validation, a `ValidationException` is raised. This exception can be specialized in some situations (invalid group definition, invalid constraint definition, invalid constraint declaration). See Exception model or the relative sections for more information.

6.1.1.1. Examples

All the examples will be based on the following class definition, constraint declarations and address instance.

```
public class Address {
    @NotNull @Size(max=30)
    private String addressline1;

    @Size(max=30)
    private String addressline2;

    private String zipCode;

    private String city;

    public String getAddressline1() {
        return addressline1;
    }

    public void setAddressline1(String addressline1) {
        this.addressline1 = addressline1;
    }
```

```
    public String getAddressline2() {
        return addressline2;
    }

    public void setAddressline2(String addressline2) {
        this.addressline2 = addressline2;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    @Size(max=30) @NotNull
    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
}

Address address = new Address();
address.setAddressline1( null );
address.setAddressline2( null );
address.setCity("Llanfairpwllgwyngyllgogerychwyrndrobwyll-llantysiliogogogoch");
//town in North Wales
```

The following code will return two `ConstraintViolation` objects. One for `addressline1` violating `@NotNull` and one for `city` violating `@Size`.

```
validator.validate(address).size() == 2
```

The following code will return one `ConstraintViolation` since `city` violates `@Size` and only `city` is validated.

```
validator.validateProperty(address, "city").size() == 1
```

The following code will return no `ConstraintViolation` objects because the value "Paris" for `city` would not raise any constraint failures.

```
validator.validateValue(Address.class, "city", "Paris").size() == 0
```

6.1.2. Methods for validating method and constructor constraints

The methods for the validation of parameters and return values of methods and constructors can be found on the interface `jakarta.validation.executable.ExecutableValidator`.

Listing 6.2: `ExecutableValidator` *interface*

```
package jakarta.validation.executable;


/**
 * Validates parameters and return values of methods and constructors.
 * Implementations of this interface must be thread-safe.
 *
 * @author Gunnar Morling
 * @since 1.1
 */
public interface ExecutableValidator {

    /**
     * Validates all constraints placed on the parameters of the given method.
     *
     * @param <T> the type hosting the method to validate
```

```
     * @param object the object on which the method to validate is invoked
     * @param method the method for which the parameter constraints is validated
     * @param parameterValues the values provided by the caller for the given method's
     *        parameters
     * @param groups the group or list of groups targeted for validation (defaults to
     *        {@link Default})
     * @return a set with the constraint violations caused by this validation;
     *         will be empty if no error occurs, but never {@code null}
     * @throws IllegalArgumentException if {@code null} is passed for any of the parameters
     *         or if parameters don't match with each other
     * @throws ValidationException if a non recoverable error happens during the
     *         validation process
     */
    <T> Set<ConstraintViolation<T>> validateParameters(T object,
                                                       Method method,
                                                       Object[] parameterValues,
                                                       Class<?>... groups);

    /**
     * Validates all return value constraints of the given method.
     *
     * @param <T> the type hosting the method to validate
     * @param object the object on which the method to validate is invoked
     * @param method the method for which the return value constraints is validated
     * @param returnValue the value returned by the given method
     * @param groups the group or list of groups targeted for validation (defaults to
     *        {@link Default})
     * @return a set with the constraint violations caused by this validation;
     *         will be empty if no error occurs, but never {@code null}
     * @throws IllegalArgumentException if {@code null} is passed for any of the object,
     *         method or groups parameters or if parameters don't match with each other
     * @throws ValidationException if a non recoverable error happens during the
     *         validation process
     */
    <T> Set<ConstraintViolation<T>> validateReturnValue(T object,
                                                        Method method,
                                                        Object returnValue,
                                                        Class<?>... groups);

    /**
     * Validates all constraints placed on the parameters of the given constructor.
     *
     * @param <T> the type hosting the constructor to validate
     * @param constructor the constructor for which the parameter constraints is validated
     * @param parameterValues the values provided by the caller for the given constructor's
     *        parameters
     * @param groups the group or list of groups targeted for validation (defaults to
     *        {@link Default})
     * @return a set with the constraint violations caused by this validation;
     *         Will be empty if no error occurs, but never {@code null}
     * @throws IllegalArgumentException if {@code null} is passed for any of the parameters
     *         or if parameters don't match with each other
     * @throws ValidationException if a non recoverable error happens during the
     *         validation process
     */
    <T> Set<ConstraintViolation<T>> validateConstructorParameters(Constructor<? extends T> constructor,
                                                                  Object[] parameterValues,
                                                                  Class<?>... groups);

    /**
     * Validates all return value constraints of the given constructor.
     *
     * @param <T> the type hosting the constructor to validate
     * @param constructor the constructor for which the return value constraints is validated
     * @param createdObject the object instantiated by the given method
     * @param groups the group or list of groups targeted for validation (defaults to
     *        {@link Default})
     * @return a set with the constraint violations caused by this validation;
     *         will be empty, if no error occurs, but never {@code null}
     * @throws IllegalArgumentException if {@code null} is passed for any of the parameters
```

```
         *         or if parameters don't match with each other
         * @throws ValidationException if a non recoverable error happens during the
         *         validation process
         */
        <T> Set<ConstraintViolation<T>> validateConstructorReturnValue(Constructor<? extends T> constructor,
                                                                       T createdObject,
                                                                       Class<?>... groups);
    }
```

`<T> Set<ConstraintViolation<T>> validateParameters(T object, Method method, Object[] parameterValues, Class<?>... groups)` validates the arguments (as given in `parameterValues`) for the parameters of a given method (identified by `method`). Cross-parameter constraints are also validated. A set containing all `ConstraintViolation` objects representing the failing constraints is returned, an empty set is returned if no constraint violations occurred. An `IllegalArgumentException` will be thrown if null is passed for any of the parameters or if the parameters don't match with each other (i.e. `object` and `method` don't match, `parameterValues` and `method` don't match).

`<T> Set<ConstraintViolation<T>> validateReturnValue(T object, Method method, Object returnValue, Class<?>... groups)` validates the return value (specified by `returnValue`) of a given method (identified by `method`). A set containing all `ConstraintViolation` objects representing the failing constraints is returned, an empty set is returned if no constraint violations occurred. An `IllegalArgumentException` will be thrown if null is passed for any of the parameters `object`, `method` and `groups` or if the parameters don't match with each other (i.e. `object` and `method` don't match, `returnValue` and `method` don't match).

`<T> Set<ConstraintViolation<T>> validateConstructorParameters(Constructor<T> constructor, Object[] parameterValues, Class<?>... groups)` validates the arguments (as given in `parameterValues`) for the parameters of a given constructor (identified by `constructor`). Cross-parameter constraints are also validated. A set containing all `ConstraintViolation` objects representing the failing constraints is returned, an empty set is returned if no constraint violations occurred. An `IllegalArgumentException` will be thrown if null is passed for any of the parameters or if the parameters don't match with each other (i.e. `parameterValues` and `constructor` don't match).

`<T> Set<ConstraintViolation<T>> validateConstructorReturnValue(Constructor<T> constructor, T createdObject, Class<?>... groups)` validates the object (specified by `createdObject`) of a given constructor (identified by `constructor`). A set containing all `ConstraintViolation` objects representing the failing constraints is returned, an empty set is returned if no constraint violations occurred. An `IllegalArgumentException` will be thrown if null is passed for any of the parameters or if the parameters don't match with each other (i.e. `createdObject` and `constructor` don't match).

None of those methods honor the XML configuration around executable validation nor the presence of `@ValidateOnExecution`. In other words, elements will be validated regardless of these settings when explicitly calling the validation methods.

6.1.2.1. Examples

All the examples will be based on the following class definitions, constraint declarations and instances.

```
public class OrderService {

    @NotNull
    private CreditCardProcessor creditCardProcessor;

    @Valid
    public OrderService(@NotNull CreditCardProcessor creditCardProcessor) {
        [...]
    }

    @NotNull
    public Order placeOrder(
        @NotNull @Size(min=3, max=20) String customerCode,
        @NotNull @Valid Item item,
        @Min(1) int quantity) {

        [...]
    }
}

public class Item {

    @NotNull;
    private String name;

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

```
Item item1 = new Item();
item1.setName("Kiwi");

Item item2 = new Item();
item2.setName(null);

Constructor<OrderService> constructor = [...]; //get constructor object
Method placeOrder = [...]; //get method object

OrderService orderService = new OrderService(new DefaultCreditCardProcessor());

ExecutableValidator executableValidator = Validation
    .buildDefaultValidatorFactory().getValidator().forExecutables();
```

The following method parameter validation will return one `ConstraintViolation` object as the customer code is null:

```
//orderService.placeOrder(null, item1, 1);
executableValidator.validateParameters(
    orderService, placeOrder, new Object[] { null, item1, 1 }).size() == 1;
```

The following method parameter validation will return one `ConstraintViolation` object as the `item` parameter is marked for cascaded validation and the given `Item` instance is not valid (its name is null):

```
//orderService.placeOrder("CUST-123", item2, 1);
executableValidator.validateParameters(
    orderService, placeOrder, new Object[] { "CUST-123", item2, 1 }).size() == 1;
```

The following constructor parameter validation will return one `ConstraintViolation` object as null is passed for the `creditCardProcessor` parameter:

```
//new OrderService(null);
executableValidator.validateConstructorParameters(constructor, new Object[] { null })
    .size() == 1;
```

Assuming the `placeOrder()` method returned `null`, the following return value validation will return one `ConstraintViolation`:

```
executableValidator.validateReturnValue(orderService, placeOrder, null).size() == 1;
```

Assuming the constructor of `OrderService` failed to store the given credit card processor into the `creditCardProcessor` field, the following validation of the constructor return value would fail as the constructor is marked with `@Valid` and the `@NotNull` constraint of the `OrderService` class would be violated:

```
executableValidator.validateConstructorReturnValue(constructor, orderService).size() == 1;
```

Let's now look at how a validation interceptor would use these methods.

```
@Interceptor
public class SampleMethodInterceptor {
    @Inject
    private Validator validator;

    @AroundInvoke
    public Object validateMethodInvocation(InvocationContext ctx) throws Exception {
        //validate parameters
        Set<ConstraintViolation<Object>> violations;
        violations = validator.forExecutables().validateParameters(
                ctx.getTarget(),
                ctx.getMethod(),
                ctx.getParameters()
        );

        //if a violation occurs for parameters, raise an exception
        if ( !violations.isEmpty() ) {
            throw new ConstraintViolationException(
                    buildMessage( ctx.getMethod(), ctx.getParameters(), violations ),
```

```
                    violations
            );
        }

        //execute the method proper
        Object result = ctx.proceed();

        //validate the return type
        violations = validator.forExecutables().validateReturnValue(
                ctx.getTarget(),
                ctx.getMethod(),
                result
        );

        //if a violation occurs for the return type, raise an exception
        if ( !violations.isEmpty() ) {
            throw new ConstraintViolationException(
                    buildMessage( ctx.getMethod(), ctx.getParameters(), violations ),
                    violations
            );
        }

        //return the result
        return result;
    }
}
```

6.1.3. groups

Groups allow you to restrict the set of constraints applied during validation. Groups targeted are passed as parameters to the `validate()`, `validateProperty()` and `validateValue()` methods as well as the methods to validate method/constructor constraints (see Methods for validating method and constructor constraints). All constraints belonging to the targeted group(s) are applied during the Validation routine. If no group is passed, the `Default` group is assumed. groups describes how to define groups on constraints.

When more than one group is evaluated and passed to the various validate methods, order is not constrained. It is equivalent to the validation of a group `G` inheriting all groups (i.e. implementing all interfaces) passed to the validation method.

6.1.3.1. Examples

```
/** Validates a minimal set of constraints */
public interface Minimal {}

public class Address {

    @NonEmpty(groups = Minimal.class)
    @Size(max=50)
    private String street1;

    @NonEmpty
    private String city;

    @NonEmpty(groups = {Minimal.class, Default.class})
    private String zipCode;

    [...]
}
```

In the previous example, `@NonEmpty` (and its composing constraints, assuming the definition given in chapter Constraint declaration and validation process) on `street1` applies to the group `Minimal`, `@Size` on `street1` applies to the group `Default` and `@NonEmpty`(and its composing constraints) on `zipCode` applies to the groups `Default` and `Minimal`.

```
validator.validate(address);
```

validates the group `Default` (implicitly) and applies `@Size` on `street1`, `@NonEmpty` (and its composing constraints) on `city`, `@NonEmpty` (and its composing constraints) on `zipCode`. Particularly, `@NonEmpty` (and its composing constraints) on `street1` are not applied.

```
validator.validate(address, Minimal.class);
```

applies `@NonEmpty` (and its composing constraints) on `street1` and `@NonEmpty` (and its composing constraints) on `zipCode` because they belong to the `Minimal` group.

```
validator.validate(address, Minimal.class, Default.class);
```

validates both `Default` and `Minimal` groups. The routine applies `@NonEmpty` (and its composing constraints) and `@Size` on `street1`, `@NonEmpty` (and its composing constraints) on `city`, `@NonEmpty` (and its composing constraints) on `zipCode`. Note that if `zipCode` is empty, only one `ConstraintViolation` object will represent the failure and the not empty validation will only be executed once.

Let's look at a more complex example involving group sequence.

```
public class Address {
    @NonEmpty(groups = Minimal.class)
    @Size(max=50, groups=FirstStep.class)
    private String street1;

    @NonEmpty(groups=SecondStep.class)
    private String city;

    @NonEmpty(groups = {Minimal.class, SecondStep.class})
    private String zipCode;

    [...]

    public interface FirstStep {}

    public interface SecondStep {}

    @GroupSequence({Firststep.class, SecondStep.class})
    public interface Total {}
}
```

When running:

```
validator.validate(address, Minimal.class, Total.class);
```

the validation process will process `@NonEmpty` (and its composing constraints) and `@Size` from `street1` and `@NonEmpty` (and its composing constraints) from `zipCode`. If `@Size` from `street1` does not generate a failure, then `@NonEmpty` (and its composing constraints) from `city` will be processed as part of `SecondStep`. Note that `@NonEmpty` (and its composing constraints) from `zipCode` are not reprocessed as they have already been processed before.

When running:

```
validator.validate(address, Total.class, SecondStep.class);
```

`@NonEmpty` (and its composing constraints) from `city` and `@NonEmpty` (and its composing constraints) from `zipCode` will be processed even if `@Size` from `street1` fails: while `SecondStep` is in the `Total` group sequence and hence should not be triggered if `FirstStep` has a failure, it also has been requested outside the sequence (in this case explicitly).

NOTE If the group definition is invalid, a `GroupDefinitionException` is raised.

6.2. ConstraintViolation

`ConstraintViolation` is the class describing a single constraint failure. A set of `ConstraintViolation` is returned for an object validation.

Listing 6.3: `ConstraintViolation` *interface*

```
/**
 * Describes a constraint violation. This object exposes the constraint
 * violation context as well as the message describing the violation.
 *
 * @param <T> the type of the root bean
 *
```

```
 * @author Emmanuel Bernard
 */
public interface ConstraintViolation<T> {

    /**
     * @return the interpolated error message for this constraint violation
     */
    String getMessage();

    /**
     * @return the non-interpolated error message for this constraint violation
     */
    String getMessageTemplate();

    /**
     * Returns the root bean being validated. For method validation, returns
     * the object the method is executed on.
     * <p>
     * Returns {@code null} when:
     * <ul>
     *     <li>the {@code ConstraintViolation} is returned after calling
     *     {@link Validator#validateValue(Class, String, Object, Class[])}</li>
     *     <li>the {@code ConstraintViolation} is returned after validating a
     *     constructor.</li>
     * </ul>
     *
     * @return the validated object, the object hosting the validated element or {@code null}
     */
    T getRootBean();

    /**
     * Returns the class of the root bean being validated.
     * For method validation, this is the object class the
     * method is executed on.
     * For constructor validation, this is the class the constructor
     * is declared on.
     *
     * @return the class of the root bean or of the object hosting the validated element
     */
    Class<T> getRootBeanClass();

    /**
     * Returns:
     * <ul>
     *     <li>the bean instance the constraint is applied on if it is
     *     a bean constraint</li>
     *     <li>the bean instance hosting the property the constraint
     *     is applied on if it is a property constraint or a container element constraint
     *     hosted on a property</li>
     *     <li>{@code null} when the {@code ConstraintViolation} is returned
     *     after calling {@link Validator#validateValue(Class, String, Object, Class[])}
     *     </li>
     *     <li>the object the method is executed on if it is
     *     a method parameter, cross-parameter or return value constraint or a
     *     container element constraint hosted on a method parameter or return value</li>
     *     <li>{@code null} if it is a constructor parameter or
     *     cross-parameter constraint or a container element constraint hosted on a
     *     constructor parameter</li>
     *     <li>the object the constructor has created if it is a
     *     constructor return value constraint</li>
     * </ul>
     *
     * @return the leaf bean
     */
    Object getLeafBean();

    /**
     * Returns an {@code Object[]} representing the constructor or method invocation
     * arguments if the {@code ConstraintViolation} is returned after validating the
     * method or constructor parameters.
```

```
     * Returns {@code null} otherwise.
     *
     * @return parameters of the method or constructor invocation or {@code null}
     *
     * @since 1.1
     */
    Object[] getExecutableParameters();

    /**
     * Returns the return value of the constructor or method invocation
     * if the {@code ConstraintViolation} is returned after validating the method
     * or constructor return value.
     * <p>
     * Returns {@code null} if the method has no return value.
     * Returns {@code null} otherwise.
     *
     * @return the method or constructor return value or {@code null}
     *
     * @since 1.1
     */
    Object getExecutableReturnValue();

    /**
     * @return the property path to the value from {@code rootBean}
     */
    Path getPropertyPath();

    /**
     * Returns the value failing to pass the constraint.
     * For cross-parameter constraints, an {@code Object[]} representing
     * the method invocation arguments is returned.
     *
     * @return the value failing to pass the constraint
     */
    Object getInvalidValue();

    /**
     * Returns the constraint metadata reported to fail.
     * The returned instance is immutable.
     *
     * @return constraint metadata
     */
    ConstraintDescriptor<?> getConstraintDescriptor();

    /**
     * Returns an instance of the specified type allowing access to
     * provider-specific APIs. If the Jakarta Validation provider
     * implementation does not support the specified class,
     * {@link ValidationException} is thrown.
     *
     * @param type the class of the object to be returned
     * @param <U> the type of the object to be returned
     * @return an instance of the specified class
     * @throws ValidationException if the provider does not support the call
     *
     * @since 1.1
     */
    <U> U unwrap(Class<U> type);
}
```

The `getMessage()` method returns the interpolated (localized) message for the failing constraint (see Message interpolation for more information on message interpolator). This can be used by clients to expose user friendly messages.

The `getMessageTemplate()` method returns the non-interpolated error message (usually the `message` attribute on the constraint declaration). Frameworks can use this as an error code key.

The `getRootBean()` method returns the root object being validated that led to the failing constraint (i.e. the object the client code passes to the `Validator.validate()` method). For method validation, returns the object the method is executed on. For constructors or when `Validator.validateValue()` is used, returns `null`.

The `getRootBeanClass()` method returns the class of the root bean being validated. For method validation, this is the object class the method is executed on. For constructor validation, this is the class the constructor is declared on.

The `getLeafBean()` method returns the following object:

- If a bean constraint, the bean instance the constraint is applied on.
- If a property constraint or a container element constraint hosted on a property, the bean instance hosting the property the constraint is applied on.
- If a property constraint, `null` when the `ConstraintViolation` is returned after calling `Validator.validateValue()`.
- If a method parameter, cross-parameter or return value constraint or a container element constraint hosted on a method parameter or return value, the object the method is executed on.
- If a constructor parameter or cross-parameter constraint or a container element constraint hosted on a constructor parameter, `null`.
- If a constructor return value constraint, the object the constructor has created.

The `getExecutableParameters()` returns the parameters provided to the method or constructor invocation or `null` if not validating a method or constructor parameters.

The `getExecutableReturnValue()` returns the return value of the method or constructor invocation or `null` if the method has no return value or if not validating a method or constructor return value.

The `getInvalidValue()` method returns the value (field, property, method/constructor parameter, method/constructor return value, container element or validated object) being passed to `isValid()`. For a cross-parameter constraint failure, an `Object[]` representing the method/constructor invocation arguments is returned. In case a constraint given on a container is subject to implicit application to the container element(s) (see Implicit unwrapping of containers), `getInvalidValue()` returns the invalid container element value.

`getConstraintDescriptor()` provides access to the failing constraint metadata (see ConstraintDescriptor).

The `getPropertyPath()` method returns the `Path` object representing the navigation path from the root object to the failing object.

`unwrap()` is provided as a way to access objects of a given type specific to a Jakarta Validation provider typically as a complement to the `ConstraintViolation` contract. Using this method makes your code non portable.

Listing 6.4: `Path` *and* `Node` *interfaces and* `ElementKind` *enum*

```
/**
 * Represents the navigation path from an object to another
 * in an object graph.
 * Each path element is represented by a {@code Node}.
 * <p>
 * The path corresponds to the succession of nodes
 * in the order they are returned by the {@code Iterator}.
 *
 * @author Emmanuel Bernard
 * @author Gunnar Morling
 * @author Guillaume Smet
 */
public interface Path extends Iterable<Path.Node> {

    /**
     * Returns a human-readable representation of this path.
     * <p>
     * Clients should not rely on any specific structure of the returned value. Instead they
     * should iterate through the path nodes and obtain any required information by calling
     * the methods on {@link Node} and its sub-types.
     *
     * @return a human-readable representation of this path
     * @since 2.0
     */
    @Override
    String toString();

    /**
     * Represents an element of a navigation path.
     */
    interface Node {

        /**
         * Returns the name of the element which the node represents:
         * <ul>
```

```
 *     <li>{@code null} if it is a leaf node which represents an entity / bean.
 *     In particular, the node representing the root object.</li>
 *     <li>The property name for a property.</li>
 *     <li>The method name for a method.</li>
 *     <li>The unqualified name of the type declaring the constructor
 *     for a constructor.</li>
 *     <li>The parameter named as defined by the {@link ParameterNameProvider}
 *     for a method or constructor parameter.</li>
 *     <li>The literal {@code <cross-parameter>} for a method or constructor
 *     cross-parameter.</li>
 *     <li>The literal {@code <return value>} for a method or constructor return
 *     value.</li>
 *     <li>The node name as defined by the {@link ValueExtractor} for a container
 *     element; specifically, the literal {@code <list element>} for elements
 *     stored in a list, the literal {@code <iterable element>} for elements
 *     stored in an {@code Iterable}, the literal {@code <map key>} for the keys
 *     stored in a {@code Map} and the literal {@code <map value>} for the values
 *     stored in a {@code Map}.
 * </ul>
 *
 * @return name of the element which the node represents
 */
String getName();

/**
 * @return {@code true} if the node represents an object contained in
 * a multi-valued container such as {@code Iterable} or {@code Map} or an array,
 * {@code false} otherwise
 */
boolean isInIterable();

/**
 * @return the index the node is placed in if contained in an array, a {@code List}
 *         or any other container supporting indexed access, {@code null} otherwise
 */
Integer getIndex();

/**
 * @return the key the node is placed in if contained in a {@code Map} or any
 *         other container supporting keyed access, {@code null} otherwise
 */
Object getKey();

/**
 * The kind of element represented by the node. The following relationship
 * between an {@link ElementKind} and its {@code Node} subtype exists:
 * <ul>
 *     <li>{@link ElementKind#BEAN}: {@link BeanNode}</li>
 *     <li>{@link ElementKind#PROPERTY}: {@link PropertyNode}</li>
 *     <li>{@link ElementKind#METHOD}: {@link MethodNode}</li>
 *     <li>{@link ElementKind#CONSTRUCTOR}: {@link ConstructorNode}</li>
 *     <li>{@link ElementKind#PARAMETER}: {@link ParameterNode}</li>
 *     <li>{@link ElementKind#CROSS_PARAMETER}: {@link CrossParameterNode}</li>
 *     <li>{@link ElementKind#RETURN_VALUE}: {@link ReturnValueNode}</li>
 *     <li>{@link ElementKind#CONTAINER_ELEMENT}: {@link ContainerElementNode}</li>
 * </ul>
 * <p>
 * This is useful to narrow down the {@code Node} type and access node specific
 * information:
 * <pre>
 * switch(node.getKind() {
 * case METHOD:
 *     name = node.getName();
 *     params = node.as(MethodNode.class).getParameterTypes();
 * case PARAMETER:
 *     index = node.as(ParameterNode.class).getParameterIndex();
 * [...]
 * }
 * </pre>
 *  @return the {@code ElementKind}
```

```
     *
     * @since 1.1
     */
    ElementKind getKind();

    /**
     * Narrows the type of this node down to the given type. The appropriate
     * type should be checked before by calling {@link #getKind()}.
     *
     * @param <T> the type to narrow down to
     * @param nodeType class object representing the descriptor type to narrow down to
     *                 to
     *
     * @return this node narrowed down to the given type.
     *
     * @throws ClassCastException if this node is not assignable to the type {@code T}
     * @since 1.1
     */
    <T extends Node> T as(Class<T> nodeType);

    /**
     * Returns a human-readable representation of this node.
     * <p>
     * Clients should not rely on any specific structure of the returned value. Instead
     * they should obtain any required information by calling the methods on this
     * interface and its sub-types.
     *
     * @return a human-readable representation of this node
     * @since 2.0
     */
    @Override
    String toString();
}

/**
 * Node representing a method.
 *
 * @since 1.1
 */
interface MethodNode extends Node {

    /**
     * @return the list of parameter types
     */
    List<Class<?>> getParameterTypes();
}

/**
 * Node representing a constructor.
 *
 * @since 1.1
 */
interface ConstructorNode extends Node {

    /**
     * @return the list of parameter types
     */
    List<Class<?>> getParameterTypes();
}

/**
 * Node representing the return value of a method or constructor.
 *
 * @since 1.1
 */
interface ReturnValueNode extends Node {
}

/**
 * Node representing a parameter of a method or constructor.
```

```
 *
 * @since 1.1
 */
interface ParameterNode extends Node {

    /**
     * @return the parameter index in the method or constructor definition
     */
    int getParameterIndex();
}

/**
 * Node representing the element holding cross-parameter constraints
 * of a method or constructor.
 *
 * @since 1.1
 */
interface CrossParameterNode extends Node {
}

/**
 * Node representing a bean.
 *
 * @since 1.1
 */
interface BeanNode extends Node {

    /**
     * @return the type of the container the node is placed in, if contained in a
     * container type such as {@code Optional}, {@code List} or {@code Map},
     * {@code null} otherwise
     *
     * @since 2.0
     */
    Class<?> getContainerClass();

    /**
     * @return the index of the type argument affected by the violated constraint, if
     * contained in a generic container type such as {@code Optional}, {@code List} or
     * {@code Map}.
     *
     * @since 2.0
     */
    Integer getTypeArgumentIndex();
}

/**
 * Node representing a property.
 *
 * @since 1.1
 */
interface PropertyNode extends Node {

    /**
     * @return the type of the container the node is placed in, if contained in a
     * container type such as {@code Optional}, {@code List} or {@code Map},
     * {@code null} otherwise
     *
     * @since 2.0
     */
    Class<?> getContainerClass();

    /**
     * @return the index of the type argument affected by the violated constraint, if
     * contained in a generic container type such as {@code Optional}, {@code List} or
     * {@code Map}, {@code null} otherwise
     *
     * @since 2.0
     */
    Integer getTypeArgumentIndex();
```

```
    }

    /**
     * Node representing an element in a generic container such as {@code Optional},
     * {@code List} or {@code Map}.
     *
     * @since 2.0
     */
    interface ContainerElementNode extends Node {

        /**
         * @return the type of the container the node is placed in
         */
        Class<?> getContainerClass();

        /**
         * @return the index of the type argument affected by the violated constraint
         */
        Integer getTypeArgumentIndex();
    }
}


/**
 * Enum of possible kinds of elements encountered in Jakarta Validation.
 * <p>
 * Mostly elements that can be constrained and described in the metadata
 * but also elements that can be part of a {@link Path} and represented
 * by a {@link Path.Node}
 *
 * @author Emmanuel Bernard
 * @author Gunnar Morling
 * @author Guillaume Smet
 *
 * @since 1.1
 */
public enum ElementKind {
    /**
     * A Java Bean or object.
     */
    BEAN,

    /**
     * A property of a Java Bean.
     */
    PROPERTY,

    /**
     * A method.
     */
    METHOD,

    /**
     * A constructor.
     */
    CONSTRUCTOR,

    /**
     * A parameter of a method or constructor.
     */
    PARAMETER,

    /**
     * Element holding cross-parameter constraints of a method or constructor.
     */
    CROSS_PARAMETER,

    /**
     * The return value of a method or constructor.
     */
```

```
    RETURN_VALUE,

    /**
     * An element stored in a container, e.g. a key or value of a {@code Map} or an element
     * of a {@code List}.
     *
     * @since 2.0
     */
    CONTAINER_ELEMENT
}
```

`Path` is an iterable of `Node` objects. `Node` offers the following methods:

- `getName()` returns the name of the element which the node represents:
  - `null` if it is a leaf node which represents an entity / bean. In particular, the node representing the root object.
  - The property name for a property.
  - The method name for a method.
  - The unqualified name of the type declaring the constructor for a constructor.
  - The parameter named as defined by the `ParameterNameProvider` (see Naming parameters) for a method or constructor parameter.
  - The literal `<cross-parameter>` for a method or constructor cross-parameter.
  - The literal `<return value>` for a method or constructor return value.
  - The name set by the applied value extractor for a container element constraint; specifically, when applying the default value extractor for iterable elements, list elements, map keys or map values, the literal `<iterable element>`, `<list element>`, `<map key>` or `<map value>`, respectively.
- `isInIterable()` returns `true` if the node represents an object contained in an array or in a multi-valued container such as `Iterable` or `Map`, `false` otherwise.
- `getIndex()` returns the index of the node if it is contained in an array, `List` or any other container supporting indexed access. Returns `null` otherwise.
- `getKey()` returns the key of the node if it is contained in a `Map` or any other container supporting keyed access. Returns `null` otherwise.
- `getKind()` returns the `ElementKind` corresponding to the actual node type. This can be used in conjunction with the method `as()` to narrow the type and access node specific methods
- `as(Class<? extends Node>)` returns the node instance narrowed to the type passed as a parameter or throws a `ClassCastException` if the type and node don't match.

Nodes are of the following possible types:

- `BeanNode`
- `PropertyNode`
- `MethodNode`
- `ConstructorNode`
- `ParameterNode`
- `CrossParameterNode`
- `ReturnValueNode`
- `ContainerElementNode`

It is possible to narrow a node instance to its precise type and extract node specific information by the use of `Node.getKind()` and `Node.as(Class<? extends Node>)`.

In particular, `MethodNode` and `ConstructorNode` host `getParameterTypes()` which return the method or constructor parameter list. Likewise `ParameterNode` hosts `getParameterIndex()` which returns the parameter index in the method or constructor parameter list.

`BeanNode`, `PropertyNode` and `ContainerElementNode` host `getContainerClass()` and `getTypeArgumentIndex()`. If the node represents an element that is contained in a container such as `Optional`, `List` or `Map`, the former returns the declared type of the container and, if the container is of a generic type, the latter returns the index of the affected type argument.

Example 6.2: Narrow a node to its specific type

```
Node node = [...];
switch ( node.getKind() ) {
case METHOD:
    MethodNode methodNode = node.as(MethodNode.class);
    methodName = methodNode.getName();
    params = methodNode.getParameterTypes().toArray(
```

```
                new Class<?>[methodNode.getParameterTypes().size()] );
            break;
        case CONSTRUCTOR:
            ConstructorNode constructorNode = node.as(ConstructorNode.class);
            methodName = constructorNode.getName();
            params = constructorNode.getParameterTypes().toArray(
                new Class<?>[constructorNode.getParameterTypes().size()] );
            break;
        case PARAMETER:
            arg = node.as(ParameterNode.class).getParameterIndex();
            break;
        case CONTAINER_ELEMENT:
            ContainerElementNode containerElementNode = node.as(ContainerElementNode.class);
            containerClass = containerElementNode.getContainerClass();
            typeArgumentIndex = containerElementNode.getTypeArgumentIndex();
            break;
        case CROSS_PARAMETER:
            [...]
        case RETURN_VALUE:
            [...]
        case PARAMETER:
            [...]
        case BEAN:
            [...]
        case PROPERTY:
            [...]
        }
```

`Path` objects are built according to the following rules:

- The runtime type is considered, not the static type. For example if a property is declared `Collection<String>` but its runtime type is `ArrayList<String>`, the property is considered an `ArrayList<String>`.
- If the failing object is the root object, a `BeanNode` with name set to `null` is added to the `Path`. The `ElementKind` of the node is `ElementKind.BEAN`.
- When an association is traversed:
  - a `PropertyNode` object whose `name` equals the name of the association property (field name or Java Bean property name) is added to `Path`. The `ElementKind` of the node is `ElementKind.PROPERTY`.
  - if the association is an array, a `List` or any other container whose value extractor invokes `ValueReceiver#indexedValue()` (see Value extractor definition), the following `Node` object added contains the index value in `getIndex()`
  - if the association is a `Map` or any other container whose value extractor invokes `ValueReceiver#keyedValue()`, the following `Node` object added (representing a given map entry) contains the key value in `getKey()`
  - for all `Iterable`, `Map` or other container whose value extractor invokes `ValueReceiver#indexedValue()`, `ValueReceiver#keyedValue()` or `ValueReceiver#iterableValue()`, the following `Node` object added is marked as `inIterable` (`isInIterable()`)
  - if the traversed object is of a container type (e.g. a `List` or `Map`), the following `Node` object added returns the declared type of the traversed container via `getContainerClass()` and the index of the affected type argument via `getTypeArgumentIndex()`
- When a nested container is traversed (e.g. when traversing into the elements of the lists in `Map<String, List<@Valid Address>>`):
  - if the value extractor of the outer container has provided a non-null node name, a `ContainerElementNode` object whose `name` equals that name is added to `Path`. The `ElementKind` of the node is `ElementKind.CONTAINER_ELEMENT`
  - if the container is a `List` or any other container whose value extractor invokes `ValueReceiver#indexedValue()`, the following `Node` object added contains the index value in `getIndex()`
  - if the container is a `Map` or any other container whose value extractor invokes `ValueReceiver#keyedValue()`, the following `Node` object added (representing a given map entry) contains the key value in `getKey()`
  - for all `Iterable`, `Map` or other container whose value extractor invokes `ValueReceiver#indexedValue()`, `ValueReceiver#keyedValue()` or `ValueReceiver#iterableValue()`, the following `Node` object added is marked as `inIterable` (`isInIterable()`)
  - the following `Node` object added returns the declared type of the traversed container via `getContainerClass()` and the index of the affected type argument via `getTypeArgumentIndex()`
- For a property level constraint (field and getter)
  - a `PropertyNode` object is added to `Path` whose `name` equals the name of the property (field name or Java Bean property name). The `ElementKind` of the node is `ElementKind.PROPERTY`.
  - the property path is considered complete
- For a class level constraint:
  - a `BeanNode` object is added to `Path` whose `name` is null. The `ElementKind` of the node is `ElementKind.BEAN`.

- the property path is considered complete

- For a method/constructor constraint (parameter, cross-parameter or return value constraint on a method or constructor):
  - a `MethodNode` respectively a `ConstructorNode` object is added to the `Path` which represents the validated method respectively constructor. The `name` of the node equals the validated method name or the validated constructor's unqualified class name, the `ElementKind` of the node is `ElementKind.METHOD` respectively `ElementKind.CONSTRUCTOR`.
  - if the constraint is on a parameter, a `ParameterNode` object is added to the `Path` which represents the validated parameter. The `name` of the node equals the parameter name as determined by the current parameter name provider (see Naming parameters). The `ElementKind` of the node is `ElementKind.PARAMETER`.
  - if the constraint is a cross-parameter constraint, a `CrossParameterNode` object is added to the `Path` which represents the validated cross-parameter element. The `name` of the node has the constant value `<cross-parameter>`. The `ElementKind` of the node is `ElementKind.CROSS_PARAMETER`.
  - if the constraint is on the return value, a `ReturnValueNode` object is added to the `Path` which represents the validated return value. The `name` of the node has the constant value `<return value>`. The `ElementKind` of the node is `ElementKind.RETURN_VALUE`.
  - the property path is considered complete
- If a parameter or the return value of a method or constructor is traversed:
  - a `MethodNode` respectively `ConstructorNode` object is added to the `Path` which represents the concerned method respectively constructor. The `name` of the node equals the concerned method name or the constructor's unqualified class name, the `ElementKind` of the node is `ElementKind.METHOD` or `ElementKind.CONSTRUCTOR`, respectively.
  - if a parameter is traversed, a `ParameterNode` object is added to the `Path` which represents the traversed parameter. The `name` of the node equals the parameter name as determined by the current parameter name provider. The `ElementKind` of the node is `ElementKind.PARAMETER`.
  - if a return value is traversed, a `ReturnValueNode` object is added to the `Path` which represents the traversed return value. The `name` of the node has the constant value `<return value>`. The `ElementKind` of the node is `ElementKind.RETURN_VALUE`.
  - if the parameter/return value is a `List` or an array, the following `Node` object added contains the index value in `getIndex()`.
  - if the parameter/return value is a `Map`, the following `Node` object added (representing a given map entry) contains the key value in `getKey()`.
  - for all `Iterable` or `Map`, the following `Node` object added is marked as `inIterable` (`isInIterable()`).

- For a container element constraint:
  - if the corresponding value extractor (see Value extractor definition) has specified a node name when calling one of the receiver methods, a `ContainerElementNode` object with that name is added to the `Path`. The `ElementKind` of the node is `ElementKind.CONTAINER_ELEMENT`. `getContainerClass()` returns the declared type of the container hosting the constraint. `getTypeArgumentIndex()` returns the index of the type argument hosting the constraint. If the constraint is given on a container and is subject to implicit application to the container's element(s) (see Implicit unwrapping of containers) and the applied value extractor is not tied to a type parameter, `getTypeArgumentIndex()` returns null.
  - if the corresponding value extractor has passed no node name to the called receiver method, no node is appended.
  - the property path is considered complete

If additional path nodes are added in a constraint validator implementation using the node builder API (see Constraint validation implementation), the following rules apply:

- if the default path ends with a `BeanNode`, this node is removed and the first added node (a `PropertyNode`) inherits its `inIterable`, `key` and `index` values. `inIterable`, `key` and `index` value must not be specified directly on this first node by the user.
- if the default path ends with a `CrossParameterNode`, this node is removed.
- then the additional nodes are appended to the (possibly amended) path generated by the Jakarta Validation engine as previously described:

- A `PropertyNode` is appended in case `addPropertyNode(String)` is invoked. The node name is equal to the name provided. The `ElementKind` of the node is `ElementKind.PROPERTY`.
- A `BeanNode` is appended in case `addBeanNode()` is invoked. The node name is `null`. The `ElementKind` of the node is `ElementKind.BEAN`.
- A `ParameterNode` is appended in case `addParameterNode(int)` is invoked. The node name is equal to the parameter name at the provided index. The name is determined by the current parameter name provider. The `ElementKind` of the node is `ElementKind.PARAMETER`. The previous node (removed) must be a `CrossParameterNode`.
- A `ContainerElementNode` is appended in case `addContainerElementNode(String, Class, Integer)` is invoked. The name, container type and type argument index of the node are equal to the values provided. The `ElementKind` of the node is `ElementKind.CONTAINER_ELEMENT`.
- If `inIterable()` is invoked, the node returns `true` for `isInIterable()`, `false` otherwise.
- If `inContainer(Class, Integer)` is invoked, the node returns the passed container type and type argument index from `getContainerClass()` and `getTypeArgumentIndex()`, respectively.
- If `atIndex(Integer)` is invoked, the node returns the provided integer for `getIndex()`, `null` otherwise.
- If `atKey(Object)` is invoked, the node returns the provided object for `getKey()`, `null` otherwise.

NOTE A given `Node` object derives its `inIterable`, `key` and `index` properties from the previous association, method parameter or

return value traversed. The same applies to `typeArgumentIndex` and `containerClass` if the given node type defines these properties.

NOTE From `getRootBean()`, `getPropertyPath()`, `getExecutableParameters()` and `getExecutableReturnValue()`, it is possible to rebuild the context of the failure.

NOTE `ConstraintViolation`s occurred during standard Jakarta Validation can be distinguished from violations occurred during method/constructor validation by analyzing the `ElementKind` of the `Node` of the first node in the violation's property path. In case of constructor or method validation, that `ElementKind` will be either `CONSTRUCTOR` or `METHOD`.

Let there be the following object definitions:

Example 6.3: Object model definition for examples

```
@SecurityChecking
public class Author {
    private String firstName;

    @NonEmpty(message="lastname must not be null")
    private String lastName;

    @Size(max=30)
    private String company;

    [...]

    @OldAndNewPasswordsDifferent
    @NewPasswordsIdentical
    public void renewPassword(
            String oldPassword, String newPassword, String retypedNewPassword) {
        [...]
    }
}

@AvailableInStore(groups={Availability.class})
public class Book {
    @NonEmpty(groups={FirstLevelCheck.class, Default.class})
    private String title;

    @Valid
    @NotNull
    private List<Author> authors;

    @Valid
    private Map<String, Review> reviewsPerSource;

    @Valid
    private Review pickedReview;

    private List<@NotBlank String> tags;

    private Map<Integer, List<@NotBlank String>> tagsByChapter;

    private List<@Valid Category> categories;

    private Map<Integer, List<@Valid Author>> authorsByChapter;

    [...]
}

public class Review {

    @Min(0)
    private int rating;
    [...]
}
```

```
public class Category {

    @Size(min=3)
    private String name;

    // [...]
}

public class Library {

    public Library(@NotNull String name, @NotNull String location) {
        [...]
    }

    public void addBook(@NotNull @Valid Book book) {
        [...]
    }

    public void addAllBooks(@NotNull List<@Valid Book> books) {
        [...]
    }

    @NotNull public String getLocation() {
        [...]
    }

    public Map<Author, @Valid Book> getMostPopularBookPerAuthor() {
        [...]
    }
}
```

Assuming a `Book` instance gets validated, the property paths to the different constraints would be as described in propertyPath examples:

Table 6.1: propertyPath examples

Constraint	propertyPath
`@AvailableInStore` on `Book`	BeanNode(name=null, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.BEAN)
`@NonEmpty` on `Book.title`	PropertyNode(name=title, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY)
`@NotNull` on `Book.authors`	PropertyNode(name=authors, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY)
`@SecurityChecking` on the fourth author, `Author`	PropertyNode(name=authors, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) BeanNode(name=null, inIterable=true, index=3, key=null, containerClass=List.class, typeArgumentIndex=0, kind=ElementKind.BEAN)
`@NonEmpty` on the fourth author, `Author.lastname`	PropertyNode(name=authors, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) PropertyNode(name=lastName, inIterable=true, index=3, key=null, containerClass=List.class, typeArgumentIndex=0, kind=ElementKind.PROPERTY)

Constraint	propertyPath
`@Size` on the first author, `Author.company`	PropertyNode(name=authors, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) PropertyNode(name=company, inIterable=true, index=0, key=null, containerClass=List.class, typeArgumentIndex=0, kind=ElementKind.PROPERTY)
`@Min` on the review associated to Consumer Report, `Review.rating`	PropertyNode(name=reviewsPerSource, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) PropertyNode(name=rating, inIterable=true, index=null, key="Consumer Report", containerClass=Map.class, typeArgumentIndex=1, kind=ElementKind.PROPERTY)
`@Min` on the picked review, `Review.rating`	PropertyNode(name=pickedReview, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) PropertyNode(name=rating, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY)
`@NotBlank` on the second tag, `Book.tags`	PropertyNode(name=tags, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) ContainerElementNode(name=<list element>, inIterable=true, index=1, key=null, containerClass=List.class, typeArgumentIndex=0, kind=ElementKind.CONTAINER_ELEMENT)
`@NotBlank` on the third tag of chapter 4, `Book.tagsByChapter`	PropertyNode(name=tagsByChapter, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) ContainerElementNode(name=<map value>, inIterable=true, index=null, key=4, containerClass=Map.class, typeArgumentIndex=1, kind=ElementKind.CONTAINER_ELEMENT) ContainerElementNode(name=<list element>, inIterable=true, index=2, key=null, containerClass=List.class, typeArgumentIndex=0, kind=ElementKind.CONTAINER_ELEMENT)
`@Size` on the name of the second category, `Category.name`	PropertyNode(name=categories, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) PropertyNode(name=name, inIterable=true, index=1, key=null, containerClass=List.class, typeArgumentIndex=0, kind=ElementKind.PROPERTY)
`@NonEmpty` on the last name of the third author of chapter 4, `Author.lastname`	PropertyNode(name=authorsByChapter, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY) ContainerElementNode(name=<map value>, inIterable=true, index=null, key=4, containerClass=Map.class, typeArgumentIndex=1, kind=ElementKind.CONTAINER_ELEMENT) PropertyNode(name=lastName, inIterable=true, index=2, key=null, containerClass=List.class, typeArgumentIndex=0, kind=ElementKind.PROPERTY)

Assuming the constructor and methods of the `Library` class are subject to method constraint validation and parameter names can be

obtained for them, the following property paths would exist for the different constraints:

Table 6.2: Property path examples for constrained methods or constructors

Constraint	propertyPath
`@NotNull` on the `location` parameter of the constructor	ConstructorNode(name=Library, inIterable=false, index=null, key=null, kind=ElementKind.CONSTRUCTOR, parameterTypes=[String.class,String.class]) ParameterNode(name=location, inIterable=false, index=null, key=null, kind=ElementKind.PARAMETER, parameterIndex=1)
`@NotNull` on the `book` parameter of the `addBook()` method	MethodNode(name=addBook, inIterable=false, index=null, key=null, kind=ElementKind.METHOD, parameterTypes=[Book.class]) ParameterNode(name=book, inIterable=false, index=null, key=null, kind=ElementKind.PARAMETER, parameterIndex=0)
`@NonEmpty` on `Book.title` during validation of `addBook()`	MethodNode(name=addBook, inIterable=false, index=null, key=null, kind=ElementKind.METHOD, parameterTypes=[Book.class]) ParameterNode(name=book, inIterable=false, index=null, key=null, kind=ElementKind.PARAMETER, parameterIndex=0) PropertyNode(name=title, inIterable=false, index=null, key=null, containerClass=null, typeArgumentIndex=null, kind=ElementKind.PROPERTY)
`@NonEmpty` on fourth book, `Book.title` during validation of `addAllBooks()`	MethodNode(name=addAllBooks, inIterable=false, index=null, key=null, kind=ElementKind.METHOD, parameterTypes=[List.class]) ParameterNode(name=books, inIterable=false, index=null, key=null, kind=ElementKind.PARAMETER, parameterIndex=0) PropertyNode(name=title, inIterable=true, index=3, key=null, containerClass=List.class, typeArgumentIndex=0, kind=ElementKind.PROPERTY)
`@NotNull` on the return value of the `getLocation()` method	MethodNode(name=getLocation, inIterable=false, index=null, key=null, kind=ElementKind.METHOD, parameterTypes=[]) ReturnValueNode(name=<return value>, inIterable=false, index=null, key=null, kind=ElementKind.RETURN_VALUE)
`@NonEmpty` on most popular book of author "John Doe", `Book.title` during validation of `getMostPopularBookPerAuthor()`	MethodNode(name=getMostPopularBookPerAuthor, inIterable=false, index=null, key=null, kind=ElementKind.METHOD, parameterTypes=[]) ReturnValueNode(name=<return value>, inIterable=false, index=null, key=null, kind=ElementKind.RETURN_VALUE) PropertyNode(name=title, inIterable=true, index=null, key=Author(firstName=John, lastName=Doe), containerClass=Map.class, typeArgumentIndex=1, kind=ElementKind.PROPERTY)
`@OldAndNewPasswordsDifferent` when executing `Author.renewPassword()` with `oldPassword`, `newPassword` and `retypedNewPassword` set to "foo". `@OldAndNewPasswordsDifferent` is a cross-parameter constraint.	MethodNode(name=renewPassword, inIterable=false, index=null, key=null, kind=ElementKind.METHOD, parameterTypes=[String.class, String.class, String.class]) CrossParameterNode(name=<cross-parameter>, inIterable=false, index=null, key=null, kind=ElementKind.CROSS_PARAMETER)

Constraint	propertyPath
`@NewPasswordsIdentical` when executing `Author.renewPassword()` with `oldPassword` as "foo", `newPassword` as "bar" and `retypedNewPassword` as "baz". `@NewPasswordsIdentical` is a cross-parameter constraint creating a constraint violation on the `retypedNewPassword` parameter.	MethodNode(name=renewPassword, inIterable=false, index=null, key=null, kind=ElementKind.METHOD, parameterTypes=[String.class, String.class, String.class]) ParameterNode(name=retypedNewPassword, inIterable=false, index=null, key=null, kind=ElementKind.PARAMETER, parameterIndex=2)

NOTE

Jakarta Validation implementations should ensure that a `ConstraintViolation` implementation is `Serializable` provided that the root bean, the leaf bean, the invalid value and keys in the `Path` object are `Serializable` objects.

If a user wishes to send `ConstraintViolation` remotely, it should make sure the object graph validated is itself `Serializable`.

6.2.1. Examples

These examples assume the following definition of `@NonEmpty`:

```
package com.acme.constraint;

@Documented
@NotNull
@Size(min = 1)
@ReportAsSingleViolation
@Constraint(validatedBy = NonEmpty.NonEmptyValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface NonEmpty {

    String message() default "{com.acme.constraint.NonEmpty.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        NonEmpty[] value();
    }

    class NonEmptyValidator implements ConstraintValidator<NonEmpty, String> {

        @Override
        public boolean isValid(String value, ConstraintValidatorContext context) {
            return true;
        }
    }
}
```

And the following class definitions:

```
public class Author {

    private String firstName;

    @NonEmpty(message = "lastname must not be null")
    private String lastName;

    @Size(max = 30)
    private String company;

    public String getFirstName() {
        return firstName;
```

```
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCompany() {
        return company;
    }

    public void setCompany(String company) {
        this.company = company;
    }
}


public class Book {

    @NonEmpty(groups = { FirstLevelCheck.class, Default.class })
    private String title;

    @Valid
    @NotNull
    private Author author;

    private List<@Size(min=3, max=30) String> tags;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }

    public List<String> getTags() {
        return tags;
    }

    public void setTags(List<String> tags) {
        this.tags = tags;
    }
}
```

When executing the following validation:

```
Author author = new Author();
author.setCompany( "ACME" );

List<String> tags = Arrays.asList( "a", "science fiction" );

Book book = new Book();
book.setTitle( "" );
book.setAuthor( author );
```

```
book.setTags( tags );

Validator validator = Validation.buildDefaultValidatorFactory().getValidator();
Set<ConstraintViolation<Book>> constraintViolations = validator.validate( book );
```

Then `constraintViolations` is a set of size 3. One of the entries represents the failure of `@NonEmpty` (or more precisely `@Size(min=1)` a composing constraint of `@NonEmpty`) on the `title` property.

The `ConstraintViolation` object for this failure passes the following assertions:

Example 6.4: Test assertions on `ContraintViolation` for `title`

```
//assuming an english locale, the interpolated message is returned
assert "may not be null or empty".equals( constraintViolation.getMessage() );
assert book == constraintViolation.getRootBean();
assert book == constraintViolation.getLeafBean();

//the offending value
assert book.getTitle().equals( constraintViolation.getInvalidValue() );

//the offending property
Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();
Node node = nodeIter.next();
assert "title".equals( node.getName() );
assert ElementKind.PROPERTY.equals( node.getKind() );

assert false == nodeIter.hasNext();
```

The second failure, `@NonEmpty` (or more precisely `@NotNull` a composing constraint of `@NonEmpty`) on the author's `lastname`, will produce the `ConstraintViolation` object satisfying the following assertions:

Example 6.5: Test assertions on `ContraintViolation` for `lastName`

```
assert "lastname must not be null".equals( constraintViolation.getMessage() );
assert book == constraintViolation.getRootBean();
assert author == constraintViolation.getLeafBean();

//the offending value
assert book.getAuthor().getLastName() == constraintViolation.getInvalidValue();

//the offending property
Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();

Node node = nodeIter.next();
assert "author".equals( node.getName() );
assert ElementKind.PROPERTY.equals( node.getKind() );

node = nodeIter.next();
assert "lastName".equals( node.getName() );
assert ElementKind.PROPERTY.equals( node.getKind() );

assert false == nodeIter.hasNext();
```

The third failure, `@Size` on one of the book's `tags`, will produce a `ConstraintViolation` object satisfying the following assertions:

Example 6.6: Test assertions on `ContraintViolation` for `tags`

```
assert "size must be between 3 and 30".equals( constraintViolation.getMessage() );
assert book == constraintViolation.getRootBean();
assert book == constraintViolation.getLeafBean();

//the offending value
assert book.getTags().get( 0 ) == constraintViolation.getInvalidValue();
```

```
//the offending property
Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();

Node node = nodeIter.next();
assert "tags".equals( node.getName() );
assert ElementKind.PROPERTY.equals( node.getKind() );

node = nodeIter.next();
assert "<list element>".equals( node.getName() );
assert ElementKind.CONTAINER_ELEMENT.equals( node.getKind() );

assert false == nodeIter.hasNext();
```

6.2.2. Examples for method and constructor constraint violations

The following examples assume the constraint, class and object definitions given in the previous section. Additionally the following class and object definitions are assumed:

```
public class Library {

    @PublicLibrary
    public Library() {
        [...]
    }

    public Library(@NotNull List<@Valid Book> books) {
        [...]
    }

    public void addBook(@NotNull @Valid Book book) {
        [...]
    }

    @Valid public Map<Author, Book> getMostPopularBookPerAuthor() {
        [...]
    }
}

public class User {

    @OldAndNewPasswordsDifferent
    public void renewPassword(String oldPassword, String newPassword, String retypedNewPassword);
}

Library library = new Library();
author.setLastName("Doe");
```

Assuming the following invocation of `addBook()` is subject to method parameter validation:

```
library.addBook(null);
```

Then one `ConstraintViolation` object would be returned by `ExecutableValidator.validateParameters()` which satisfies the following assertions:

```
//assuming an english locale, the interpolated message is returned
assert "must not be null".equals( constraintViolation.getMessage() );

assert library == constraintViolation.getRootBean();
assert Library.class == constraintViolation.getRootBeanClass();
assert library == constraintViolation.getLeafBean();
assert null == constraintViolation.getInvalidValue();

assert new Object[]{ null }.equals( constraintViolation.getExecutableParameters() );
assert null == constraintViolation.getExecutableReturnValue();

Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();
```

```
Node node = nodeIter.next();
assert "addBook".equals( node.getName() );
assert ElementKind.METHOD.equals( node.getKind() );

node = nodeIter.next();
//assuming the default parameter name provider is used and parameter names can
//be obtained
assert "book".equals( node.getName() );
assert ElementKind.PARAMETER.equals( node.getKind() );

assert false == nodeIter.hasNext();
```

Assuming the following invocation of `addBook()` is subject to method parameter validation:

```
library.addBook(book);
```

Then one `ConstraintViolation` object would be returned by `ExecutableValidator.validateParameters()` which satisfies the following assertions:

```
//assuming an english locale, the interpolated message is returned
assert "may not be null or empty".equals( constraintViolation.getMessage() );

assert library == constraintViolation.getRootBean();
assert Library.class == constraintViolation.getRootBeanClass();
assert book == constraintViolation.getLeafBean();
assert book.getTitle().equals( constraintViolation.getInvalidValue() );

assert new Object[]{ book }.equals( constraintViolation.getExecutableParameters() );
assert null == constraintViolation.getExecutableReturnValue();

Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();

Node node = nodeIter.next();
assert "addBook".equals( node.getName() );
assert ElementKind.METHOD.equals( node.getKind() );

node = nodeIter.next();
//assuming the default parameter name provider is used and parameter names can
//be obtained
assert "book".equals( node.getName() );
assert ElementKind.PARAMETER.equals( node.getKind() );

node = nodeIter.next();
assert "title".equals( node.getName() );
assert ElementKind.PROPERTY.equals( node.getKind() );

assert false == nodeIter.hasNext();
```

Assuming the following invocation of `User.renewPassword()` is subject to method parameter validation and the `@OldAndNewPasswordsDifferent` constraint is violated:

```
User user = [...];
user.renewPassword("foo", "foo", "foo");
```

Then one `ConstraintViolation` object would be returned by `ExecutableValidator.validateParameters()` which satisfies the following assertions:

```
assert user == constraintViolation.getRootBean();
assert User.class == constraintViolation.getRootBeanClass();
assert user == getLeafBean();
assert new Object[]{ "foo", "foo", "foo" }.equals( constraintViolation.getInvalidValue() );

assert new Object[]{ "foo", "foo", "foo" }.equals( constraintViolation.getExecutableParameters() );
assert null == constraintViolation.getExecutableReturnValue();

Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();

Node node = nodeIter.next();
```

```
assert "renewPassword".equals( node.getName() );
assert ElementKind.METHOD.equals( node.getKind() );

node = nodeIter.next();
assert "<cross-parameter>" == node.getName();
assert ElementKind.CROSS_PARAMETER.equals( node.getKind() );

assert false == nodeIter.hasNext();
```

Assuming the following invocation of the `Library` constructor accepting a list of books is subject to constructor parameter validation:

```
Library anotherLibrary = new Library(null);
```

Then one `ConstraintViolation` object would be returned by `ExecutableValidator.validateConstructorParameters()` which satisfies the following assertions:

```
//assuming an english locale, the interpolated message is returned
assert "must not be null".equals( constraintViolation.getMessage() );

assert null == constraintViolation.getRootBean();
assert Library.class == constraintViolation.getRootBeanClass();
assert null == constraintViolation.getLeafBean();
assert null == constraintViolation.getInvalidValue();

assert new Object[]{ null }.equals( constraintViolation.getExecutableParameters() );
assert null == constraintViolation.getExecutableReturnValue();

Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();

Node node = nodeIter.next();
assert "Library".equals( node.getName() );
assert ElementKind.CONSTRUCTOR.equals( node.getKind() );

node = nodeIter.next();
//assuming the default parameter name provider is used and parameter names can
//be obtained
assert "books".equals( node.getName() );
assert ElementKind.PARAMETER.equals( node.getKind() );

assert false == nodeIter.hasNext();
```

Assuming the following invocation of `getMostPopularBookPerAuthor()` is subject to method return value validation and returns a `Map` containing one entry with key `author` and value `book`:

```
Map<Author, Book> mostPopularBookPerAuthor = library.getMostPopularBookPerAuthor();
```

Then one `ConstraintViolation` object would be returned by `ExecutableValidator.validateReturnValue()` which satisfies the following assertions:

```
//assuming an english locale, the interpolated message is returned
assert "may not be null or empty".equals( constraintViolation.getMessage() );

assert library == constraintViolation.getRootBean();
assert Library.class == constraintViolation.getRootBeanClass();
assert book == constraintViolation.getLeafBean();
assert book.getTitle().equals( constraintViolation.getInvalidValue() );

assert null == constraintViolation.getExecutableParameters();
assert mostPopularBookPerAuthor == constraintViolation.getExecutableReturnValue();

Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();

Node node = nodeIter.next();
assert "getMostPopularBookPerAuthor".equals( node.getName() );
assert ElementKind.METHOD.equals( node.getKind() );

node = nodeIter.next();
assert "<return value>" == node.getName();
```

```
assert ElementKind.RETURN_VALUE.equals( node.getKind() );

node = nodeIter.next();
assert "title".equals( node.getName() );
assert ElementKind.PROPERTY.equals( node.getKind() );
assert author.equals( node.getKey() );
assert true == node.isInIterable();

assert false == nodeIter.hasNext();
```

Assuming the following invocation of the `Library` default constructor is subject to constructor return value validation and returns an instance which violates the `@PublicLibrary` constraint:

```
Library publicLibrary = new Library();
```

Then one `ConstraintViolation` object would be returned by `ExecutableValidator.validateConstructorReturnValue()` which satisfies the following assertions:

```
assert null == constraintViolation.getRootBean();
assert Library.class == constraintViolation.getRootBeanClass();
assert publicLibrary == constraintViolation.getLeafBean();
assert publicLibrary == constraintViolation.getInvalidValue();

assert null == constraintViolation.getExecutableParameters();
assert library == constraintViolation.getExecutableReturnValue();

Iterator<Node> nodeIter = constraintViolation.getPropertyPath().iterator();

Node node = nodeIter.next();
assert "Library".equals( node.getName() );
assert ElementKind.CONSTRUCTOR.equals( node.getKind() );

node = nodeIter.next();
assert "<return value>" == node.getName();
assert ElementKind.RETURN_VALUE.equals( node.getKind() );

assert false == nodeIter.hasNext();
```

6.3. Message interpolation

A message interpolator is responsible for transforming the so called message descriptor specified via the message attribute of the constraint into a fully expanded, human-readable error message.

6.3.1. Default message interpolation

Every conforming Jakarta Validation implementation includes a default message interpolator which has to comply with the algorithm defined here to interpolate message descriptors. As precondition for message interpolation the following applies:

- Each constraint defines a message descriptor via its `message` property.
- Every constraint definition defines a default message descriptor for that constraint.
- Messages can be overridden at constraint declaration time by setting the `message` property on the constraint.

The message descriptor is a string literal and may contain one or more message parameters or expressions. Message parameters and expressions are string literals enclosed in `{}` or `${}` respectively. The following character escaping rules apply:

- `\{` is considered as the literal `{` instead of being considered as the beginning of a message parameter
- `\}` is considered as the literal `}` instead of being considered as the end of a message parameter
- `\\` is considered as the literal `\` instead of being considered as the escaping character
- `\$` is considered as the literal `$` instead of being considered as the beginning of a message expression

Below are two examples using message parameters and expressions. The second is evaluated using Jakarta Expression Language as defined in Message expressions using Jakarta Expression Language.

Example 6.7: Message using parameters

```
Value must be between {min} and {max}
```

Example 6.8: Message using expressions

```
Must be greater than ${inclusive == true ? 'or equal to ' : ''}{value}
```

6.3.1.1. Default message interpolation algorithm

The default message interpolator uses the following steps:

1. Message parameters are extracted from the message string and used as keys to search the `ResourceBundle` named `ValidationMessages` (often materialized as the property file `/ValidationMessages.properties` and its locale variations) using the defined locale (see Locale for default message interpolation). If a property is found, the message parameter is replaced with the property value in the message string. Step 1 is applied recursively until no replacement is performed (i.e. a message parameter value can itself contain a message parameter).
2. Message parameters are extracted from the message string and used as keys to search the Jakarta Validation provider's built-in `ResourceBundle` using the defined locale (see Locale for default message interpolation). If a property is found, the message parameter is replaced with the property value in the message string. Contrary to step 1, step 2 is not processed recursively.
3. If step 2 triggers a replacement, then step 1 is applied again. Otherwise step 4 is performed.
4. Message parameters are extracted from the message string. Those matching the name of an attribute of the constraint are replaced by the value of that attribute in the constraint declaration. Parameter interpolation has precedence over message expressions. For example for the message descriptor `${value}`, trying to evaluate `{value}` as message parameter has precedence over evaluating `${value}` as message expression.
5. Message expressions are extracted from the message string and evaluated using Jakarta Expression Language. See also Message expressions using Jakarta Expression Language.

NOTE The proposed algorithm ensures that custom resource bundle always have priority over built-in resource bundle at all level of the recursive resolution. It also ensures that constraint declarations attributes values are not interpolated further.

NOTE The precedence of message parameter over expression interpolation ensures backwards compatibility to Jakarta Validation 1.0.

6.3.1.2. Locale for default message interpolation

The locale to be used for message interpolation is defined as following:

- if the locale is passed explicitly to the interpolator method via `interpolate(String, Context, Locale)`, this provided instance is used.
- otherwise, the default `Locale` as provided by `Locale.getDefault()` is used.

6.3.1.3. Message expressions using Jakarta Expression Language

The default message interpolation allows the use of Jakarta Expression Language. Expressions to be evaluated by Jakarta Expression Language need to be enclosed in `${}` within the message descriptor . The following properties and beans have to be made available in the Jakarta Expression Language context:

- the attribute values of the constraint declaration mapped to their attribute name
- the validated value mapped under the name `validatedValue`.
- a bean mapped to the name `formatter` exposing the vararg method `format(String format, Object... args)`. This method must behave like `java.util.Formatter.format(String format, Object... args)`. The locale used for formatting is defined by Locale for default message interpolation. The `formatter` bean allows to format property values, for example in the case of the validated value being 98.12345678, `${formatter.format('%1$.2f', validatedValue)}` would format it to 98.12 (two digits after the decimal point, where the use of '.' vs ',' would be locale specific).

If an exception occurs during message interpolation, e.g. due to invalid expressions or references to an unknown property, the message expression stays unchanged.

6.3.2. Custom message interpolation

A custom message interpolator may be provided (e.g., to interpolate contextual data, or to adjust the default `Locale` used). A message interpolator implements the `MessageInterpolator` interface.

Listing 6.5: `MessageInterpolator` *interface*

```
/**
 * Interpolates a given constraint violation message.
 * <p>
 * Implementations should be as tolerant as possible on syntax errors.
 * Implementations must be thread-safe.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface MessageInterpolator {

    /**
     * Interpolates the message template based on the constraint validation context.
     * <p>
     * The locale is defaulted according to the {@code MessageInterpolator}
     * implementation. See the implementation documentation for more detail.
     *
     * @param messageTemplate the message to interpolate
     * @param context contextual information related to the interpolation
     *
     * @return interpolated error message
     */
    String interpolate(String messageTemplate, Context context);

    /**
     * Interpolates the message template based on the constraint validation context.
     * The {@code Locale} used is provided as a parameter.
     *
     * @param messageTemplate the message to interpolate
     * @param context contextual information related to the interpolation
     * @param locale the locale targeted for the message
     *
     * @return interpolated error message
     */
    String interpolate(String messageTemplate, Context context,  Locale locale);

    /**
     * Information related to the interpolation context.
     */
    interface Context {
        /**
         * @return {@link ConstraintDescriptor} corresponding to the constraint being
         * validated
         */
        ConstraintDescriptor<?> getConstraintDescriptor();

        /**
         * @return value being validated
         */
        Object getValidatedValue();

        /**
         * Returns an instance of the specified type allowing access to
         * provider-specific APIs. If the Jakarta Validation provider
         * implementation does not support the specified class,
         * {@link ValidationException} is thrown.
         *
         * @param type the class of the object to be returned
         * @param <T> the type of the object to be returned
         * @return an instance of the specified class
         * @throws ValidationException if the provider does not support the call
         *
```

```
         * @since 1.1
         */
        <T> T unwrap(Class<T> type);
    }
}
```

`messageTemplate` is the value of the `message` attribute of the constraint declaration or provided to the `ConstraintValidatorContext` methods.

The `Context` object contains contextual information related to the interpolation.

`getConstraintDescriptor()` returns the `ConstraintDescriptor` object representing the metadata of the failing constraint (see Constraint metadata request APIs).

`getValidatedValue()` returns the value being validated.

`MessageInterpolator.interpolate(String, Context)` is invoked for each constraint violation report generated. The default `Locale` of custom message interpolators is implementation specific.

`MessageInterpolator.interpolate(String, Context, Locale)` can be invoked by a wrapping `MessageInterpolator` to enforce a specific `Locale` value by bypassing or overriding the default `Locale` strategy (see Use MessageInterpolator to use a specific Locale value).

A message interpolator implementation must be thread-safe.

The message interpolator is provided to the `ValidatorFactory` at construction time using `Configuration.messageInterpolator(MessageInterpolator)`. This message interpolator is shared by all `Validator` objects generated by this `ValidatorFactory`.

It is possible to override the `MessageInterpolator` implementation for a given `Validator` instance by invoking `ValidatorFactory.usingContext().messageInterpolator(messageInterpolator).getValidator()`.

It is recommended that `MessageInterpolator` implementations delegate final interpolation to the Jakarta Validation default `MessageInterpolator` to ensure standard Jakarta Validation interpolation rules are followed, The default implementation is accessible through `Configuration.getDefaultMessageInterpolator()`.

If the interpolation process leads to an exception, the exception is wrapped into a `ValidationException`.

6.3.3. Examples

These examples describe message interpolation based on the default message interpolator's built-in messages (see Standard ResourceBundle messages), and the `ValidationMessages.properties` file shown in table message interpolation. The current locale is assumed English.

```
//ValidationMessages.properties
myapp.creditcard.error=credit card number not valid
```

Table 6.3: message interpolation

Failing constraint declaration	interpolated message
`@NotNull`	must not be null
`@Max(30)`	must be less than or equal to 30
`@Size(min=5, max=15, message="Key must have \\{{min}\\} \\ \\{{max}\\} characters")`	Key must have {5} \ {15} characters
`@Digits(integer=9, fraction=2)`	numeric value out of bounds (<9 digits>.<2 digits> expected)
`@CreditCard(message={myapp.creditcard.error})`	credit card number not valid

Here is an approach to specify the `Locale` value to choose on a given `Validator` using a `Locale` aware `MessageInterpolator`. See Bootstrapping for more details on the APIs.

Example 6.9: Use MessageInterpolator to use a specific Locale value

```
/**
```

```
 * Delegates to a MessageInterpolator implementation but enforces a given Locale
 */
public class LocaleSpecificMessageInterpolator implements MessageInterpolator {
    private final MessageInterpolator defaultInterpolator;
    private final Locale defaultLocale;

    public LocaleSpecificMessageInterpolator(MessageInterpolator interpolator, Locale locale) {
        this.defaultLocale = locale;
        this.defaultInterpolator = interpolator;
    }

    /**
     * enforces the locale passed to the interpolator
     */
    @Override
    public String interpolate(String message,
                              Context context) {
        return defaultInterpolator.interpolate(message,
                                               context,
                                               this.defaultLocale);
    }

    // no real use, implemented for completeness
    @Override
    public String interpolate(String message,
                              Context context,
                              Locale locale) {
        return defaultInterpolator.interpolate(message, context, locale);
    }
}


Locale locale = getMyCurrentLocale();
MessageInterpolator interpolator = new LocaleSpecificMessageInterpolator(
                                     validatorFactory.getMessageInterpolator(),
                                     locale);

Validator validator = validatorFactory.usingContext()
                                      .messageInterpolator(interpolator)
                                      .getValidator();
```

Most of the time, however, the relevant `Locale` will be provided by your application framework transparently. This framework will implement its own version of `MessageInterpolator` and pass it during the `ValidatorFactory` configuration. The application will not have to set the `Locale` itself. This example shows how a container framework would implement `MessageInterpolator` to provide a user specific default locale.

Example 6.10: Contextual container possible `MessageInterpolator` *implementation*

```
public class ContextualMessageInterpolator implements MessageInterpolator {
    private final MessageInterpolator delegate;

    public ContextualMessageInterpolator(MessageInterpolator delegate) {
        this.delegate = delegate;
    }

    @Override
    public String interpolate(String message, Context context) {
        Locale locale = Container.getManager().getUserLocale();
        return this.delegate.interpolate(
                        message, context, locale );
    }

    @Override
    public String interpolate(String message, Context context, Locale locale) {
        return this.delegate.interpolate(message, context, locale);
    }
}
```

```
//Build the ValidatorFactory
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .messageInterpolator(
        new ContextualMessageInterpolator(
                configuration.getDefaultMessageInterpolator() ) )
    .buildValidatorFactory();

//The container uses the factory to validate constraints using the specific MessageInterpolator
Validator validator = factory.getValidator();
```

6.4. Triggering method validation

Jakarta Validation itself doesn't trigger the evaluation of method constraints. That is, just annotating any methods or constructors with parameter or return value constraints doesn't automatically enforce these constraints, just as annotating any fields or properties with bean constraints doesn't enforce these either.

Instead method constraints must be validated by invoking the appropriate methods on `jakarta.validation.executable.ExecutableValidator`. Typically this won't happen by manually calling these methods but rather automatically upon invocation of the constrained methods or constructors, using approaches and techniques such as Jakarta Context and Dependency Injection/Jakarta Enterprise Beans interceptors, aspect-oriented programming or dynamic proxies.

The validation of method / constructor constraints comprises the following steps:

- Intercept the method call to be validated
- Validate the parameter values provided by the method caller using `ExecutableValidator.validateParameters()` or `ExecutableValidator.validateConstructorParameters()`.
- If this validation yields a non-empty set of constraint violations, throw a `ConstraintViolationException` wrapping the violations. Otherwise proceed with the actual method invocation.
- Validate the result returned by the invoked method using `ExecutableValidator.validateReturnValue()` or `ExecutableValidator.validateConstructorReturnValue()`.
- If this validation yields a non-empty set of constraint violations, throw a `ConstraintViolationException` wrapping the violations. Otherwise return the invocation result to the method caller.

By throwing a `ConstraintViolationException` if either of the validation steps fails, it is ensured that the control flow

- only arrives at the method's body if the caller has satisfied the method's preconditions and
- only returns to the method caller if the method's postconditions are guaranteed.

By default, integrators intercept and validate methods either hosting a constraint or being marked for cascaded validation (`@Valid`) whether it be on the method itself or on any of its parameters. The `Default` group is used for validation out of the box.

Integrators are encouraged to use Jakarta Validation's metadata API to find whether or not a method or a constructor should be intercepted. This guarantees that XML descriptors as well as future mapping strategies are taken into account. Note that the metadata API does not take into account the fact that a method or constructor validation has been enabled or disabled by the techniques described in Method and constructor validation.

Here is an example of what such metadata usage would be:

Example 6.11: Using metadata API to figure out if method interception is required

```
//For methods

// is there any constrained method on this type
// assuming we don't validate on getter execution
public boolean interceptMethods(Class<?> type) {
    return validator.getConstraintsForClass( type ).getConstrainedMethods(MethodType.NON_GETTER).size() > 0;
}

// is this method constrained
public boolean interceptMethod(Class<?> type, Method method) {
    BeanDescriptor bean = validator.getConstraintsForClass( type );
    MethodDescriptor methodDescriptor = bean.getConstraintsForMethod(
        method.getName(), method.getParameterTypes() );
    return methodDescriptor != null;
}
```

```
// should method parameters be validated
public boolean requiresParametersValidation(Class<?> type, Method method) {
    BeanDescriptor bean = validator.getConstraintsForClass( type );
    MethodDescriptor methodDescriptor = bean.getConstraintsForMethod(
        method.getName(), method.getParameterTypes() );
    if ( methodDescriptor != null ) {
        return methodDescriptor.hasConstrainedParameters();
    }
    else {
        return false;
    }
}

// should method return value be validated?
public boolean requiresReturnValueValidation(Class<?> type, Method method) {
    BeanDescriptor bean = validator.getConstraintsForClass( type );
    MethodDescriptor methodDescriptor = bean.getConstraintsForMethod(
        method.getName(), method.getParameterTypes() );
    if ( methodDescriptor != null ) {
        return methodDescriptor.hasConstrainedReturnValue();
    }
    else {
        return false;
    }
}
```

Example 6.12: Using metadata API to figure out if constructor interception is required

```
//For constructors

// is there any constrained constructor on this type
public <T> boolean interceptConstructors(Class<T> type) {
    BeanDescriptor bean = validator.getConstraintsForClass( type );
    return bean.getConstrainedConstructors().size() > 0;
}

// is this constructor constrained
public <T> boolean interceptConstructor(Class<T> type, Constructor<T> ctor) {
    BeanDescriptor bean = validator.getConstraintsForClass( type );
    ConstructorDescriptor constructorDescriptor = bean.getConstraintsForConstructor(
        ctor.getParameterTypes() );
    return constructorDescriptor != null;
}

// should constructor parameters be validated
public <T> boolean requiresParametersValidation(Class<T> type, Constructor<T> ctor) {
    BeanDescriptor bean = validator.getConstraintsForClass( type );
    ConstructorDescriptor constructorDescriptor = bean.getConstraintsForConstructor(
        ctor.getParameterTypes() );
    if ( constructorDescriptor != null ) {
        return constructorDescriptor.hasConstrainedParameters();
    }
    else {
        return false;
    }
}

// should constructor return value be validated?
public <T> boolean requiresReturnValueValidation(Class<T> type, Constructor<T> ctor) {
    BeanDescriptor bean = validator.getConstraintsForClass( type );
    ConstructorDescriptor constructorDescriptor = bean.getConstraintsForConstructor(
        ctor.getName(),
        ctor.getParameterTypes()
    );
    if ( constructorDescriptor != null ) {
        return constructorDescriptor.hasConstrainedReturnValue();
    }
```

```
        else {
            return false;
        }
    }
```

NOTE Calls to the metadata API is likely only going to be needed during the initialization phase of the interception framework. Results can then be cached.

NOTE Only methods or constructors intercepted by the underlying interception technology can be validated.

The integration technology must put the validation interceptor as late as possible (if not last) in the interception stack. In particular, validation of parameters should be done after the security and transaction start logic. Likewise, return value validation should be done before the transaction stop logic. Putting the validation interceptor as late as possible in the stack ensures this.

NOTE

Why have the validation interceptor after other interceptors?

There are several reasons for delaying validation compared to other interceptors:

- You don't want to start business code before security has been cleared
- You might need transaction support in your validations
- You want transaction to fail if the return value is invalid
- Generally speaking, it makes more sense to apply technical layers around the more business focused constraints

6.5. Bootstrapping

The bootstrapping API aims at providing a `ValidatorFactory` object which is used to create `Validator` instances. The bootstrap process is decoupled from the provider implementation initialization: a bootstrap implementation must be able to bootstrap any Jakarta Validation provider implementation. The bootstrap sequence has been designed to achieve several goals:

- plug multiple implementations
- choose a specific implementation
- extensibility: an application using a specific provider implementation can use specific configurations
- share and reuse of metadata across `Validator`s
- leave as much freedom as possible to implementations
- provide integration mechanisms to Jakarta EE (starting from version 6) and other containers
- type safety

The main artifacts involved in the bootstrap process are:

- `Validation`: API entry point. Lets you optionally define the Jakarta Validation provider targeted as well as a provider resolution strategy. `Validation` generates `Configuration` objects and can bootstrap any provider implementation.
- `ValidationProvider`: contract between the bootstrap procedure and a Jakarta Validation provider implementation.
- `ValidationProviderResolver`: returns a list of all Jakarta Validation providers available in the execution context (generally the classpath).
- `Configuration`: collects the configuration details that will be used to build `ValidatorFactory`. A specific sub interface of `Configuration` must be provided by Jakarta Validation providers. This sub interface typically hosts provider specific configurations.
- `ValidatorFactory`: result of the bootstrap process. Build `Validator` instances from a given Jakarta Validation provider.
- `META-INF/validation.xml`: a configuration file Jakarta Validation users can use to customize the configuration of the default `ValidatorFactory`.

Let's first see the API in action through some examples before diving into the concrete definitions.

6.5.1. Examples

The most simple approach is to initialize the default Jakarta Validation provider or the one defined in the XML configuration file. The `ValidatorFactory` is then ready to provide `Validator` instances.

Example 6.13: Simple Jakarta Validation bootstrap sequence

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

//cache the factory somewhere
```

```
Validator validator = factory.getValidator();

//when the application shuts down, close ValidatorFactory
factory.close();
```

The `ValidatorFactory` object is thread-safe. Building `Validator` instances is typically a cheap operation. Building a `ValidatorFactory` is typically more expensive. Make sure to check your Jakarta Validation implementation documentation for more accurate details.

The second example shows how a container can customize aspects like message interpolation, constraint validator instantiation and others.

Example 6.14: Customize message resolution, traversable resolver etc.

```
//some customization from a container
ValidatorFactory factory = Validation
       .byDefaultProvider().configure()
          .messageInterpolator( new ContainerMessageInterpolator() )
          .constraintValidatorFactory( new ContainerComponentConstraintValidatorFactory() )
          .traversableResolver( new JPAAwareTraversableResolver() )
          .parameterNameProvider( new AnnotationBasedParameterNameProvider() )
          .clockProvider( new BatchJobClockProvider() )
          .addValueExtractor( new TableValueExtractor() )
          .addValueExtractor( new MultiMapValueExtractor() )
          .buildValidatorFactory();

//cache the factory somewhere
Validator validator = factory.getValidator();

//when the application shuts down, close ValidatorFactory
factory.close();
```

The third example shows how to bootstrap Jakarta Validation in an environment not following the traditional Java class loader strategies (such as tools or alternative service containers like OSGi). They can provide some alternative provider resolution strategy to discover Jakarta Validation providers.

Example 6.15: Customize the Jakarta Validation provider resolution mechanism

```
//osgi environment
ValidatorFactory factory = Validation
       .byDefaultProvider()
          .providerResolver( new OSGiServiceDiscoverer() )
          .configure()
             .buildValidatorFactory();

//cache the factory somewhere
Validator validator = factory.getValidator();

//when the bundle shuts down, close ValidatorFactory
factory.close();
```

The next example shows how a client can choose a specific Jakarta Validation provider and configure provider specific properties programmatically in a type-safe way.

Example 6.16: Use a specific provider and add specific configuration

```
ValidatorFactory factory = Validation
       .byProvider( ACMEProvider.class )  //chose a specific provider
       .configure()
          .messageInterpolator( new ContainerMessageInterpolator() ) //default configuration option
          .addConstraint(Address.class, customConstraintDescriptor) //ACME specific method
          .buildValidatorFactory();

//same initialization decomposing calls
ACMEConfiguration acmeConfiguration = Validation
```

```
        .byProvider( ACMEProvider.class )
        .configure();

ValidatorFactory factory = acmeConfiguration
          .messageInterpolator( new ContainerMessageInterpolator() ) //default configuration option
          .addConstraint(Address.class, customConstraintDescriptor) //ACME specific method
          .buildValidatorFactory();

/**
 * ACME specific validator configuration and configuration options
 */
public interface ACMEConfiguration extends Configuration<ACMEConfiguration> {
    /**
     * Programmatically add constraints. Specific to the ACME provider.
     */
    ACMEConfiguration addConstraint(Class<?> entity,
                                    ACMEConstraintDescriptor constraintDescriptor);
}

/**
 * ACME validation provider
 * Note how ACMEConfiguration and ACMEProvider are linked together
 * via the generic parameter.
 */
public class ACMEProvider implements ValidationProvider<ACMEConfiguration> {
    [...]
}
```

The last example shows how a `Validator` can use a specific `MessageInterpolator` implementation.

Example 6.17: Use a specific MessageInterpolator instance for a given Validator

```
ValidatorFactory factory = [...];
MessageInterpolator customInterpolator = new LocaleSpecificMessageInterpolator(
    locale,
    factory.getMessageInterpolator()
);

Validator localizedValidator =
    factory.usingContext()
                   .messageInterpolator(customInterpolator)
                   .getValidator();
```

In the same way, a custom `TraversableResolver` can be passed.

We will now explore the various interfaces, their constraints and usage. We will go from the `ValidatorFactory` to the `Validation` class walking up the bootstrap chain.

6.5.2. ValidatorFactory

`ValidatorFactory` objects build and provide initialized instances of `Validator` to Jakarta Validation clients. Each `Validator` instance is configured for a given context (message interpolator, traversable resolver etc.). Clients should cache `ValidatorFactory` objects and reuse them for optimal performances. The API is designed to allow implementors to share constraint metadata in `ValidatorFactory`. `ValidatorFactory` instances must be closed (by calling the `close()` method) by its creator when no longer in use.

`ValidatorFactory` implementations must be thread-safe. `ValidatorFactory` implementations can cache `Validator` instances if needed.

Listing 6.6: `ValidatorFactory` *interface*

```
/**
 * Factory returning initialized {@code Validator} instances.
 * <p>
 * Implementations are thread-safe and instances are typically cached and reused.
 *
 * @author Emmanuel Bernard
```

```
 * @author Gunnar Morling
 * @author Hardy Ferentschik
 * @author Guillaume Smet
 */
public interface ValidatorFactory extends AutoCloseable {

    /**
     * Returns an initialized {@link Validator} instance using the
     * factory defaults for message interpolator, traversable resolver
     * and constraint validator factory.
     * <p>
     * Validator instances can be pooled and shared by the implementation.
     *
     * @return an initialized {@code Validator} instance
     */
    Validator getValidator();

    /**
     * Defines a new validator context and returns a {@code Validator}
     * compliant this new context.
     *
     * @return a {@link ValidatorContext} instance
     */
    ValidatorContext usingContext();

    /**
     * Returns the {@link MessageInterpolator} instance configured at
     * initialization time for the {@code ValidatorFactory}.
     * This is the instance used by {@link #getValidator()}.
     *
     * @return {@code MessageInterpolator} instance
     */
    MessageInterpolator getMessageInterpolator();

    /**
     * Returns the {@link TraversableResolver} instance configured
     * at initialization time for the {@code ValidatorFactory}.
     * This is the instance used by {@link #getValidator()}.
     *
     * @return {@code TraversableResolver} instance
     */
    TraversableResolver getTraversableResolver();

    /**
     * Returns the {@link ConstraintValidatorFactory} instance
     * configured at initialization time for the
     * {@code ValidatorFactory}.
     * This is the instance used by {@link #getValidator()}.
     *
     * @return {@code ConstraintValidatorFactory} instance
     */
    ConstraintValidatorFactory getConstraintValidatorFactory();

    /**
     * Returns the {@link ParameterNameProvider} instance configured at
     * initialization time for the {@code ValidatorFactory}.
     * This is the instance used by #getValidator().
     *
     * @return {@code ParameterNameProvider} instance
     *
     * @since 1.1
     */
    ParameterNameProvider getParameterNameProvider();

    /**
     * Returns the {@link ClockProvider} instance configured at
     * initialization time for the {@code ValidatorFactory}.
     * This is the instance used by #getValidator().
     *
     * @return {@code ClockProvider} instance
```

```
     *
     * @since 2.0
     */
    ClockProvider getClockProvider();

    /**
     * Returns an instance of the specified type allowing access to
     * provider-specific APIs. If the Jakarta Validation provider
     * implementation does not support the specified class, a
     * {@code ValidationException} is thrown.
     *
     * @param type the class of the object to be returned
     * @param <T> the type of the object to be returned
     * @return an instance of the specified class
     * @throws ValidationException if the provider does not
     *         support the call.
     */
    public <T> T unwrap(Class<T> type);

    /**
     * Closes the {@code ValidatorFactory} instance.
     *
     * After the {@code ValidatorFactory} instance is closed, calling the following
     * methods is not allowed:
     * <ul>
     *     <li>methods of this {@code ValidatorFactory} instance</li>
     *     <li>methods of {@link Validator} instances created by this
     *     {@code ValidatorFactory}</li>
     * </ul>
     *
     * @since 1.1
     */
    @Override
    public void close();
}
```

A `ValidatorFactory` is provided by a `Configuration` object.

`unwrap()` is provided as a way to access objects of a given type specific to a Jakarta Validation provider typically as a complement to the `ValidatorFactory` contract. Using this method makes your code non portable.

Example 6.18: Using unwrap to access a provider specific contract

```
//if using the ACME provider
ACMEValidatorFactory acmeFactory = factory.unwrap(ACMEValidatorFactory.class);
acmeFactory.setSpecificConfiguration( [...] );
```

`close()` closes the `ValidatorFactory` instance which becomes unavailable and should be immediately discarded. This is also true of all the `Validator` instances it has spawned. The behavior is undefined and non portable if these instances are used after the `ValidatorFactory` has been closed.

`getMessageInterpolator()` returns the `MessageInterpolator` instance configured during the initialization of the `ValidatorFactory`. It is particularly useful to build a `Validator` specific `MessageInterpolator` wrapping the one from the `ValidatorFactory`.

`getTraversableResolver()` returns the `TraversableResolver` instance configured during the initialization of the `ValidatorFactory`. It is particularly useful to build a `Validator` specific `TraversableResolver` wrapping the one from the `ValidatorFactory`.

`getConstraintValidatorFactory()` returns the `ConstraintValidatorFactory` instance configured during the initialization of the `ValidatorFactory`. It is particularly useful to build a `Validator` specific `ConstraintValidatorFactory` wrapping the one from the `ValidatorFactory`.

`getParameterNameProvider()` returns the `ParameterNameProvider` instance configured during the initialization of the `ValidatorFactory`. It is particularly useful to build a `Validator` specific `ParameterNameProvider` wrapping the one from the `ValidatorFactory`.

`getClockProvider()` returns the `ClockProvider` instance configured during the initialization of the `ValidatorFactory`. It is particularly useful to build a `Validator` specific `ClockProvider` wrapping the one from the `ValidatorFactory`.

`ValidatorContext` returned by `usingContext()` can be used to customize the state in which the `Validator` must be initialized. This is used to customize the `MessageInterpolator`, the `TraversableResolver`, the `ParameterNameProvider`, the `ClockProvider` or the `ConstraintValidatorFactory`.

Listing 6.7: `ValidatorContext` *interface*

```
/**
 * Represents the context that is used to create {@link Validator}
 * instances.
 *
 * A client may use methods of the {@code ValidatorContext} returned by
 * {@link ValidatorFactory#usingContext()} to customize
 * the context used to create {@code Validator} instances
 * (for instance establish different message interpolators or
 * traversable resolvers).
 *
 * @author Emmanuel Bernard
 * @author Gunnar Morling
 * @author Guillaume Smet
 */
public interface ValidatorContext {

    /**
     * Defines the message interpolator implementation used by the
     * {@link Validator}.
     * <p>
     * If not set or if {@code null} is passed as a parameter,
     * the message interpolator of the {@link ValidatorFactory}
     * is used.
     *
     * @param messageInterpolator the {@link MessageInterpolator} used by the
     * {@code Validator}
     *
     * @return self following the chaining method pattern
     */
    ValidatorContext messageInterpolator(MessageInterpolator messageInterpolator);

    /**
     * Defines the traversable resolver implementation used by the
     * {@link Validator}.
     * <p>
     * If not set or if {@code null} is passed as a parameter,
     * the traversable resolver of the {@link ValidatorFactory} is used.
     *
     * @param traversableResolver the {@code TraversableResolver} used by the
     * {@code Validator}
     * @return self following the chaining method pattern
     */
    ValidatorContext traversableResolver(TraversableResolver traversableResolver);

    /**
     * Defines the constraint validator factory implementation used by the
     * {@link Validator}.
     * If not set or if {@code null} is passed as a parameter,
     * the constraint validator factory of the {@link ValidatorFactory} is used.
     *
     * @param factory the {@link ConstraintValidatorFactory} used by the {@code Validator}
     * @return self following the chaining method pattern
     */
    ValidatorContext constraintValidatorFactory(ConstraintValidatorFactory factory);

    /**
     * Defines the parameter name provider implementation used by the
     * {@link Validator}. If not set or if {@code null} is passed as a parameter,
     * the parameter name provider of the {@link ValidatorFactory} is used.
     *
     * @param parameterNameProvider parameter name provider implementation.
     * @return self following the chaining method pattern
     *
```

```
         * @since 1.1
         */
        ValidatorContext parameterNameProvider(ParameterNameProvider parameterNameProvider);

        /**
         * Defines the {@link ClockProvider} implementation used by the {@link Validator}.
         * If not set or if {@code null} is passed as a parameter,
         * the clock provider of the {@link ValidatorFactory} is used.
         *
         * @param clockProvider {@code ClockProvider} implementation
         * @return self following the chaining method pattern
         *
         * @since 2.0
         */
        ValidatorContext clockProvider(ClockProvider clockProvider);

        /**
         * Adds a value extractor to be used by the {@link Validator}. Has
         * priority over any extractor for the same type and type parameter
         * detected through the service loader, given in the XML configuration or
         * configured for the validator factory.
         *
         * @param extractor value extractor implementation
         * @return self following the chaining method pattern
         * @throws ValueExtractorDeclarationException if more than one extractor for
         *         the same type and type parameter is added
         * @since 2.0
         */
        ValidatorContext addValueExtractor(ValueExtractor<?> extractor);

        /**
         * Returns an initialized {@link Validator} instance respecting the defined state.
         * {@code Validator} instances can be pooled and shared by the implementation.
         *
         * @return contextualized {@code Validator}
         */
        Validator getValidator();
    }
```

The `MessageInterpolator`, the `TraversableResolver`, the `ConstraintValidatorFactory`, the `ParameterNameProvider` or the `ClockProvider` passed to the `ValidatorContext` are used instead of the `ValidatorFactory`'s `MessageInterpolator`, `TraversableResolver`, `ConstraintValidatorFactory`, `ParameterNameProvider` or `ClockProvider` instances. A `ValueExtractorDeclarationException` is raised if more than one extractor for the same type and type parameter is added via `addValueExtractor()`.

Example 6.19: Use of ValidatorFactory

```
ValidatorFactory factory = [...];
Validator validatorUsingDefaults = factory.getValidator();
Validator validatorUsingCustomTraversable = factory
                    .usingContext()
                    .traversableResolver( new JPATraversableResolver() )
                    .getValidator();
```

See Use MessageInterpolator to use a specific Locale value for an example using `ValidatorFactory.getMessageInterpolator()`.

6.5.3. Configuration

The responsibility of the `Configuration` is to collect configuration information, to determine the correct provider implementation and to delegate the `ValidatorFactory` creation to the selected provider. More concretely `Configuration` lets you define:

- the message interpolator instance
- the traversable resolver instance
- the constraint validator factory instance
- the parameter name provider instance

- the clock provider instance
- value extractor instances
- XML constraint mappings
- provider specific properties
- whether or not `META-INF/validation.xml` is considered

`Configuration` does provide a `MessageInterpolator` implementation following the default Jakarta Validation `MessageInterpolator` rules as defined in Default message interpolation. You can access it by calling `getDefaultMessageInterpolator()`. Such an implementation is useful to let a custom `MessageInterpolator` delegate to the standard `MessageInterpolator` (see Custom message interpolation and an example making use of `getDefaultMessageInterpolator()` in Contextual container possible `MessageInterpolator` implementation).

`Configuration` does provide a `TraversableResolver` implementation following the default Jakarta Validation `TraversableResolver` rules as defined in Traversable property. You can access it by calling `getDefaultTraversableResolver()`. Such an implementation is useful to let a custom `TraversableResolver` delegate to the standard `TraversableResolver`.

`Configuration` does provide a `ConstraintValidatorFactory` implementation following the default Jakarta Validation `ConstraintValidatorFactory` rules as defined in The ConstraintValidatorFactory. You can access it by calling `getDefaultConstraintValidatorFactory()`. Such an implementation is useful to let a custom `ConstraintValidatorFactory` delegate to the standard `ConstraintValidatorFactory`.

`Configuration` does provide a `ParameterNameProvider` implementation following the default Jakarta Validation `ParameterNameProvider` rules as defined in Naming parameters. You can access it by calling `getDefaultParameterNameProvider()`. Such an implementation is useful to let a custom `ParameterNameProvider` delegate to the standard `ParameterNameProvider`.

`Configuration` does provide a `ClockProvider` implementation following the default Jakarta Validation `ClockProvider` rules as defined in Implementation of temporal constraint validators. You can access it by calling `getDefaultClockProvider()`.

Via `getBootstrapConfiguration()`, `Configuration` also exposes data stored in `META-INF/validation.xml` (see XML configuration: META-INF/validation.xml). This is particularly useful for containers wishing to control the instance creation and lifecycle (more information at Bootstrapping considerations).

> **NOTE** `BootstrapConfiguration.getDefaultValidatedExecutableTypes()` and `BootstrapConfiguration.isExecutableValidationEnabled()` are not used by the Jakarta Validation engine but exposed here for interception technologies - see Method and constructor validation.

Via `addValueExtractor()`, additional value extractor implementations can be added to the configuration. A value extractor for a given type and type parameter takes precedence over any extractor for the same type and type parameter detected through the service loader or given in the XML configuration. A `ValueExtractorDeclarationException` is raised if more than one extractor for the same type and type parameter is added.

Using `addMapping()`, additional constraint mapping XML descriptors can be added to the configuration (see XML configuration: META-INF/validation.xml). The given input streams should support the `mark()` and `reset()` methods defined by `java.io.InputStream`. Streams not supporting the `mark()` and `reset()` methods will be wrapped with an `InputStream` implementation supporting these methods by the Jakarta Validation provider in order to allow the streams to be read several times.

Clients call `Configuration.buildValidatorFactory()` to retrieve the initialized `ValidatorFactory` instance. It is legal to invoke `buildValidatorFactory()` several times, e.g. in order to retrieve several `ValidatorFactory` instances with a slightly different configuration (see Using Configuration to create several validator factories).

Listing 6.8: `Configuration` *and* `BootstrapConfiguration` *interfaces*

```
/**
 * Receives configuration information, selects the appropriate
 * Jakarta Validation provider and builds the appropriate {@link ValidatorFactory}.
 * <p>
 * Usage:
 * <pre>
 * //provided by one of the Validation bootstrap methods
 * Configuration&lt;?&gt; configuration =
 *     ValidatorFactory = configuration
 *         .messageInterpolator( new CustomMessageInterpolator() )
 *         .buildValidatorFactory();
 * </pre>
 * <p>
 * By default, the configuration information is retrieved from
 * {@code META-INF/validation.xml}.
 * It is possible to override the configuration retrieved from the XML file
 * by using one or more of the {@code Configuration} methods.
```

```
 * <p>
 * The {@link ValidationProviderResolver} is specified at configuration time
 * (see {@link ValidationProvider}).
 * If none is explicitly requested, the default {@code ValidationProviderResolver} is used.
 * <p>
 * The provider is selected in the following way:
 * <ul>
 *     <li>if a specific provider is requested programmatically using
 *     {@link Validation#byProvider(Class)}, find the first provider implementing
 *     the provider class requested and use it</li>
 *     <li>if a specific provider is requested in {@code META-INF/validation.xml},
 *     find the first provider implementing the provider class requested and use it</li>
 *     <li>otherwise, use the first provider returned by the
 *     {@code ValidationProviderResolver}</li>
 * </ul>
 * <p>
 * Implementations are not meant to be thread-safe.
 *
 * @param <T> the type of a provider-specific specialization of this contract
 *
 * @author Emmanuel Bernard
 * @author Gunnar Morling
 * @author Hardy Ferentschik
 * @author Guillaume Smet
 */
public interface Configuration<T extends Configuration<T>> {

    /**
     * Ignores data from the {@code META-INF/validation.xml} file if this
     * method is called.
     * <p>
     * This method is typically useful for containers that parse
     * {@code META-INF/validation.xml} themselves and pass the information
     * via the {@code Configuration} methods.
     *
     * @return {@code this} following the chaining method pattern.
     */
    T ignoreXmlConfiguration();

    /**
     * Defines the message interpolator used. Has priority over the configuration
     * based message interpolator.
     * <p>
     * If {@code null} is passed, the default message interpolator is used
     * (defined in XML or the specification default).
     *
     * @param interpolator message interpolator implementation
     * @return {@code this} following the chaining method pattern
     */
    T messageInterpolator(MessageInterpolator interpolator);

    /**
     * Defines the traversable resolver used. Has priority over the configuration
     * based traversable resolver.
     * <p>
     * If {@code null} is passed, the default traversable resolver is used
     * (defined in XML or the specification default).
     *
     * @param resolver traversable resolver implementation
     * @return {@code this} following the chaining method pattern
     */
    T traversableResolver(TraversableResolver resolver);

    /**
     * Defines the constraint validator factory. Has priority over the configuration
     * based constraint factory.
     * <p>
     * If {@code null} is passed, the default constraint validator factory is used
     * (defined in XML or the specification default).
     *
```

```
     * @param constraintValidatorFactory constraint factory implementation
     * @return {@code this} following the chaining method pattern
     */
    T constraintValidatorFactory(ConstraintValidatorFactory constraintValidatorFactory);

    /**
     * Defines the parameter name provider. Has priority over the configuration
     * based provider.
     * <p>
     * If {@code null} is passed, the default parameter name provider is used
     * (defined in XML or the specification default).
     *
     * @param parameterNameProvider parameter name provider implementation
     * @return {@code this} following the chaining method pattern.
     *
     * @since 1.1
     */
    T parameterNameProvider(ParameterNameProvider parameterNameProvider);

    /**
     * Defines the clock provider. Has priority over the configuration
     * based provider.
     * <p>
     * If {@code null} is passed, the default clock provider is used
     * (defined in XML or the specification default).
     *
     * @param clockProvider clock provider implementation
     * @return {@code this} following the chaining method pattern.
     *
     * @since 2.0
     */
    T clockProvider(ClockProvider clockProvider);

    /**
     * Adds a value extractor. Has priority over any extractor for the same
     * type and type parameter detected through the service loader or given in
     * the XML configuration.
     *
     * @param extractor value extractor implementation
     * @return {@code this} following the chaining method pattern.
     * @throws ValueExtractorDeclarationException if more than one extractor for
     *         the same type and type parameter is added
     * @since 2.0
     */
    T addValueExtractor(ValueExtractor<?> extractor);

    /**
     * Add a stream describing constraint mapping in the Jakarta Validation XML
     * format.
     * <p>
     * The stream should be closed by the client API after the
     * {@link ValidatorFactory} has been built. The Jakarta Validation provider
     * must not close the stream.
     *
     * @param stream
     *        XML mapping stream; the given stream should support the
     *        mark/reset contract (see {@link InputStream#markSupported()});
     *        if it doesn't, it will be wrapped into a stream supporting the
     *        mark/reset contract by the Jakarta Validation provider
     *
     * @return {@code this} following the chaining method pattern
     * @throws IllegalArgumentException if {@code stream} is null
     */
    T addMapping(InputStream stream);

    /**
     * Adds a provider specific property. This property is equivalent to
     * XML configuration properties.
     * If the underlying provider does not know how to handle the property,
     * it must silently ignore it.
```

```
     * <p>
     * Note: Using this non type-safe method is generally not recommended.
     * <p>
     * It is more appropriate to use, if available, the type-safe equivalent provided
     * by a specific provider via its {@link Configuration} subclass.
     * <pre>
     * ValidatorFactory factory = Validation.byProvider(ACMEProvider.class)
     *     .configure()
     *         .providerSpecificProperty(ACMEState.FAST)
     *     .buildValidatorFactory();
     * </pre>
     * This method is typically used by containers parsing {@code META-INF/validation.xml}
     * themselves and injecting the state to the {@code Configuration} object.
     * <p>
     * If a property with a given name is defined both via this method and in the
     * XML configuration, the value set programmatically has priority.
     * <p>
     * If {@code null} is passed as a value, the value defined in XML is used. If no value
     * is defined in XML, the property is considered unset.
     *
     * @param name property name
     * @param value property value
     * @return {@code this} following the chaining method pattern
     * @throws IllegalArgumentException if {@code name} is null
     */
    T addProperty(String name, String value);

    /**
     * Returns an implementation of the {@link MessageInterpolator} interface
     * following the default {@code MessageInterpolator} defined in the
     * specification:
     * <ul>
     *     <li>use the {@code ValidationMessages} resource bundle to load keys</li>
     *     <li>use {@code Locale.getDefault()}</li>
     * </ul>
     *
     * @return default {@code MessageInterpolator} implementation compliant with the
     *         specification
     */
    MessageInterpolator getDefaultMessageInterpolator();

    /**
     * Returns an implementation of the {@link TraversableResolver} interface
     * following the default {@code TraversableResolver} defined in the
     * specification:
     * <ul>
     *     <li>if Java Persistence is available in the runtime environment,
     *     a property is considered reachable if Java Persistence considers
     *     the property as loaded</li>
     *     <li>if Java Persistence is not available in the runtime environment,
     *     all properties are considered reachable</li>
     *     <li>all properties are considered cascadable.</li>
     * </ul>
     *
     * @return default {@code TraversableResolver} implementation compliant with the
     *         specification
     */
    TraversableResolver getDefaultTraversableResolver();

    /**
     * Returns an implementation of the {@link ConstraintValidatorFactory} interface
     * following the default {@code ConstraintValidatorFactory} defined in the
     * specification:
     * <ul>
     *     <li>uses the public no-arg constructor of the {@link ConstraintValidator}</li>
     * </ul>
     *
     * @return default {@code ConstraintValidatorFactory} implementation compliant with the
     *         specification
     */
```

```
    ConstraintValidatorFactory getDefaultConstraintValidatorFactory();

    /**
     * Returns an implementation of the {@link ParameterNameProvider}
     * interface following the default {@code ParameterNameProvider}
     * defined in the specification:
     * <ul>
     *     <li>returns the actual parameter names as provided in the validated
     *     executable's definition, if the class file of the executable contains
     *     parameter name information</li>
     *     <li>
     *     otherwise returns names in the form {@code arg&lt;PARAMETER_INDEX&gt;},
     *     where {@code PARAMETER_INDEX} starts at 0 for the first parameter,
     *     e.g. {@code arg0}, {@code arg1} etc.</li>
     * </ul>
     *
     * @return default {@code ParameterNameProvider} implementation compliant with
     *         the specification
     *
     * @since 1.1
     */
    ParameterNameProvider getDefaultParameterNameProvider();

    /**
     * Returns an implementation of the {@link ClockProvider}
     * interface following the default {@code ClockProvider}
     * defined in the specification:
     * <ul>
     *     <li>returns a clock representing the current system time and default time
     *     zone.</li>
     * </ul>
     *
     * @return default {@code ClockProvider} implementation compliant with
     *         the specification
     *
     * @since 2.0
     */
    ClockProvider getDefaultClockProvider();

    /**
     * Returns configuration information stored in the {@code META-INF/validation.xml} file.
     * <p>
     * <b>Note</b>:
     * <br>
     * Implementations are encouraged to lazily build this object to delay parsing.
     *
     * @return returns an instance of {@link BootstrapConfiguration}; this method never
     *         returns {@code null}; if there is no {@code META-INF/validation.xml} the
     *         different getters of the returned instance will return {@code null}
     *         respectively an empty set or map
     *
     * @since 1.1
     */
    BootstrapConfiguration getBootstrapConfiguration();

    /**
     * Build a {@link ValidatorFactory} implementation.
     *
     * @return the {@code ValidatorFactory}
     * @throws ValidationException if the {@code ValidatorFactory} cannot be built
     */
    ValidatorFactory buildValidatorFactory();
}


/**
 * Represents the user specified default configuration in
 * {@code META-INF/validation.xml}.
 * <p>
 * Note that modifications to the returned objects do not have any effect.
```

```
 * Instead use the methods provided on {@link Configuration} in order to
 * apply modifications to the configuration.
 *
 * @author Emmanuel Bernard
 * @author Gunnar Morling
 * @author Hardy Ferentschik
 * @author Guillaume Smet
 * @since 1.1
 */
public interface BootstrapConfiguration {

    /**
     * Class name of the {@link ValidationProvider} implementation
     * or {@code null} if none is specified.
     *
     * @return validation provider class name or {@code null}
     */
    String getDefaultProviderClassName();

    /**
     * Class name of the {@link ConstraintValidatorFactory} implementation
     * or {@code null} if none is specified.
     *
     * @return constraint validator factory class name or {@code null}
     */
    String getConstraintValidatorFactoryClassName();

    /**
     * Class name of the {@link MessageInterpolator} implementation
     * or {@code null} if none is specified.
     *
     * @return message interpolator class name or {@code null}
     */
    String getMessageInterpolatorClassName();

    /**
     * Class name of the {@link TraversableResolver} implementation
     * or {@code null} if none is specified.
     *
     * @return traversable resolver class name or {@code null}
     */
    String getTraversableResolverClassName();

    /**
     * Class name of the {@link ParameterNameProvider} implementation
     * or {@code null} if none is specified.
     *
     * @return parameter name provider class name or {@code null}
     */
    String getParameterNameProviderClassName();

    /**
     * Class name of the {@link ClockProvider} implementation or
     * {@code null} if none is specified.
     *
     * @return clock provider class name or {@code null}
     *
     * @since 2.0
     */
    String getClockProviderClassName();

    /**
     * Returns the class names of {@link ValueExtractor}s.
     *
     * @return the value extractor class names or an empty set if none are specified
     * @since 2.0
     */
    Set<String> getValueExtractorClassNames();

    /**
```

```
     * Returns a set of resource paths pointing to XML constraint mapping files.
     * The set is empty if none are specified.
     *
     * @return set of constraint mapping resource paths
     */
    Set<String> getConstraintMappingResourcePaths();

    /**
     * Returns true if the validation execution is explicitly marked as enabled
     * or if it is left undefined.
     *
     * @return whether validation execution is globally enabled
     */
    boolean isExecutableValidationEnabled();

    /**
     * Returns the set of executable types that should be considered
     * unless explicitly overridden via {@link ValidateOnExecution}.
     * <p>
     * Returns a set containing {@link ExecutableType#CONSTRUCTORS} and
     * {@link ExecutableType#NON_GETTER_METHODS} if unspecified in the configuration.
     *
     * @return set of validated executable types
     */
    Set<ExecutableType> getDefaultValidatedExecutableTypes();

    /**
     * Returns properties as a map of string based key/value pairs.
     * The map is empty if no property has been specified.
     *
     * @return the properties map
     */
    Map<String, String> getProperties();
}
```

A Jakarta Validation provider must define a sub interface of `Configuration` uniquely identifying the provider. This subclass is linked to its provider via the `ValidationProvider` generic parameter. The `Configuration` sub interface typically hosts provider specific configuration methods.

To facilitate the use of provider specific configuration methods, `Configuration` uses generics: `Configuration<T extends Configuration<T>>` ; the generic return type `T` is returned by chaining methods. The provider specific sub interface must resolve the generic T as itself as shown in Example of provider specific Configuration sub interface.

Example 6.20: Example of provider specific Configuration sub interface

```
/**
 * Unique identifier of the ACME provider
 * also hosts some provider specific configuration methods
 */
public interface ACMEConfiguration
    extends Configuration<ACMEConfiguration> {

    /**
     * Enables constraints implementation dynamic reloading when using ACME
     * default to false
     */
    ACMEConfiguration enableDynamicReloading(boolean);

}
```

When `Configuration.buildValidatorFactory()` is called, the initialized `ValidatorFactory` is returned. More specifically, the requested Jakarta Validation provider is determined and the result of `validationProvider.buildValidatorFactory(ConfigurationState)` is returned. `ConfigurationState` gives access to the configuration artifacts defined in `META-INF/validation.xml` (unless XML configuration is ignored) and provided programmatically to `Configuration`. Generally speaking, programmatically defined elements have priority over XML defined configuration elements (read the `Configuration` JavaDoc and see XML configuration: META-INF/validation.xml for more information).

NOTE A typical implementation of `Configuration` also implements `ConfigurationState`, hence `this` can be passed to `buildValidatorFactory(ConfigurationState)`.

Streams represented in the XML configuration and opened by the `Configuration` implementation must be closed by the `Configuration` implementation after the `ValidatorFactory` creation (or if an exception occurs). Streams provided programmatically are the responsibility of the application.

Listing 6.9: `ConfigurationState` *interface*

```
package jakarta.validation.spi;


/**
 * Contract between a {@link Configuration} and a
 * {@link ValidationProvider} to create a {@link ValidatorFactory}.
 * <p>
 * The configuration artifacts defined in the XML configuration and provided to the
 * {@code Configuration} are merged and passed along via
 * {@code ConfigurationState}.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 * @author Gunnar Morling
 * @author Guillaume Smet
 */
public interface ConfigurationState {

    /**
     * Returns {@code true} if {@link Configuration#ignoreXmlConfiguration()} has been
     * called.
     * <p>
     * In this case, the {@link ValidatorFactory} must ignore
     * {@code META-INF/validation.xml}.
     *
     * @return {@code true} if {@code META-INF/validation.xml} should be ignored
     */
    boolean isIgnoreXmlConfiguration();

    /**
     * Returns the message interpolator of this configuration.
     * <p>
     * Message interpolator is defined in the following decreasing priority:
     * <ul>
     *     <li>set via the {@link Configuration} programmatic API</li>
     *     <li>defined in {@code META-INF/validation.xml} provided that
     *     {@code ignoreXmlConfiguration} is false. In this case the instance
     *     is created via its no-arg constructor.</li>
     *     <li>{@code null} if undefined.</li>
     * </ul>
     *
     * @return message interpolator instance or {@code null} if not defined
     */
    MessageInterpolator getMessageInterpolator();

    /**
     * Returns a set of configuration streams.
     * <p>
     * The streams are defined by:
     * <ul>
     *     <li>mapping XML streams passed programmatically in {@link Configuration}</li>
     *     <li>mapping XML streams located in the resources defined in
     *     {@code META-INF/validation.xml} (constraint-mapping element)</li>
     * </ul>
     * <p>
     * Streams represented in the XML configuration and opened by the
     * {@code Configuration} implementation must be closed by the
     * {@code Configuration} implementation after the {@link ValidatorFactory}
     * creation (or if an exception occurs). All streams are guaranteed to
```

```
     * adhere to the mark/reset contract (see {@link InputStream#markSupported()}
     * by the Jakarta Validation provider.
     *
     * @return set of input stream
     */
    Set<InputStream> getMappingStreams();

    /**
     * Returns a set of value extractors.
     * <p>
     * The extractors are retrieved from the following sources in decreasing
     * order:
     * <ul>
     *     <li>extractors passed programmatically to {@link Configuration}</li>
     *     <li>extractors defined in {@code META-INF/validation.xml} provided
     *     that {@code ignoredXmlConfiguration} is {@code false}</li>
     *     <li>extractors loaded through the Java service loader</li>
     * </ul>
     * An extractor for a given type and type parameter passed in
     * programmatically takes precedence over any extractor for the same type
     * and type parameter defined in {@code META-INF/validation.xml} or loaded
     * through the service loader. Extractors defined in
     * {@code META-INF/validation.xml} take precedence over any extractor for
     * the same type and type parameter loaded through the service loader.
     * <p>
     * Extractors defined in {@code META-INF/validation.xml} or loaded through
     * the service loader are instantiated using their no-arg constructor.
     *
     * @return set of value extractors; may be empty but never {@code null}
     *
     * @since 2.0
     */
    Set<ValueExtractor<?>> getValueExtractors();

    /**
     * Returns the constraint validator factory of this configuration.
     * <p>
     * The {@link ConstraintValidatorFactory} implementation is defined in the following
     * decreasing priority:
     * <ul>
     *     <li>set via the {@link Configuration} programmatic API</li>
     *     <li>defined in {@code META-INF/validation.xml} provided that
     *     {@code ignoredXmlConfiguration} is {@code false}. In this case the instance
     *     is created via its no-arg constructor.</li>
     *     <li>{@code null} if undefined.</li>
     * </ul>
     *
     * @return factory instance or {@code null} if not defined
     */
    ConstraintValidatorFactory getConstraintValidatorFactory();

    /**
     * Returns the traversable resolver for this configuration.
     * <p>
     * {@link TraversableResolver} is defined in the following decreasing priority:
     * <ul>
     *     <li>set via the {@link Configuration} programmatic API</li>
     *     <li>defined in {@code META-INF/validation.xml} provided that
     *     {@code ignoredXmlConfiguration} is {@code false}. In this case the
     *     instance is created via its no-arg constructor.</li>
     *     <li>{@code null} if undefined.</li>
     * </ul>
     *
     * @return traversable resolver instance or {@code null} if not defined
     */
    TraversableResolver getTraversableResolver();

    /**
     * Returns the parameter name provider for this configuration.
     * <p>
```

```
         * {@link ParameterNameProvider} is defined in the following decreasing priority:
         * <ul>
         *     <li>set via the {@link Configuration} programmatic API</li>
         *     <li>defined in {@code META-INF/validation.xml} provided that
         *     {@code ignoreXmlConfiguration} is {@code false}. In this case the instance
         *     is created via its no-arg constructor.</li>
         *     <li>{@code null} if undefined.</li>
         * </ul>
         *
         * @return parameter name provider instance or {@code null} if not defined
         *
         * @since 1.1
         */
        ParameterNameProvider getParameterNameProvider();

        /**
         * Returns the clock provider for this configuration.
         * <p>
         * {@link ClockProvider} is defined in the following decreasing priority:
         * <ul>
         *     <li>set via the {@link Configuration} programmatic API</li>
         *     <li>defined in {@code META-INF/validation.xml} provided that
         *     {@code ignoreXmlConfiguration} is {@code false}. In this case the instance
         *     is created via its no-arg constructor.</li>
         *     <li>{@code null} if undefined.</li>
         * </ul>
         *
         * @return clock provider instance or {@code null} if not defined
         *
         * @since 2.0
         */
        ClockProvider getClockProvider();

        /**
         * Returns a map of non type-safe custom properties.
         * <p>
         * Properties defined via:
         * <ul>
         *     <li>{@link Configuration#addProperty(String, String)}</li>
         *     <li>{@code META-INF/validation.xml} provided that
         *     {@code ignoreXmlConfiguration} is {@code false}.</li>
         * </ul>
         * <p>
         * If a property is defined both programmatically and in XML,
         * the value defined programmatically has priority.
         *
         * @return {@code Map} whose key is the property key and the value
         *         the property value
         */
        Map<String, String> getProperties();
    }
```

The requested provider implementation is resolved according to the following rules in the following order:

- Use the provider implementation requested if `Configuration` has been created from `Validation.byProvider(Class)`.
- Use the provider implementation described in the XML configuration (under `validation-config.default-provider` see XML configuration: META-INF/validation.xml) if defined: the value of this element is the fully qualified class name of the `ValidationProvider` implementation uniquely identifying the provider.
- Use the first provider implementation returned by `validationProviderResolver.getValidationProviders()`.

The `ValidationProviderResolver` is specified when `Configuration` instances are created (see `ValidationProvider`). If no `ValidationProviderResolver` instance has been specified, the default `ValidationProviderResolver` is used.

`Configuration` instances are provided to the Jakarta Validation client through the `Validation` methods. `Configuration` instances are created by `ValidationProvider`.

If a problem occurs while building the `ValidatorFactory`, a `ValidationException` is raised. This can be due to various reasons including:

- malformed XML configuration
- malformed XML mapping
- inability to find the provider (or a provider)
- inability to instantiate extension classes provided in the XML configuration
- inconsistent XML mapping (entity declared more than once, incorrect field etc.)
- invalid constraint declaration or definition

Other exception causes may occur.

Here is an example of `Configuration` use.

Example 6.21: Use Configuration

```
Configuration<?> configuration = [...];
ValidatorFactory factory = configuration
              .messageInterpolator( new WBMessageInterpolator() )
              .traversableResolver( new JPAAwareTraversableResolver() )
              .buildValidatorFactory();
```

The following shows an example of setting up a `Configuration`, retrieving a validator factory from it, subsequently altering the configuration and then retrieving another factory:

Example 6.22: Using Configuration to create several validator factories

```
Configuration<?> configuration = [...];
ValidatorFactory factory1 = configuration
              .messageInterpolator( new WBMessageInterpolator() )
              .buildValidatorFactory();

ValidatorFactory factory2 = configuration
              .traversableResolver( new JPAAwareTraversableResolver() )
              .buildValidatorFactory();
```

Here, `factory1` is set up using a custom message interpolator, while `factory2` is set up using the same message interpolator and additionally using a custom traversable resolver.

6.5.4. ValidationProvider and ValidationProviderResolver

`ValidationProvider` is the contract between the bootstrap process and a specific Jakarta Validation provider. `ValidationProviderResolver` implements the discovery mechanism for Jakarta Validation provider implementations. Any Jakarta Validation client can implement such a discovery mechanism but it is typically implemented by containers having specific class loader structures and restrictions.

6.5.4.1. ValidationProviderResolver

`ValidationProviderResolver` returns the list of Jakarta Validation providers available at runtime and more specifically a `ValidationProvider` instance for each provider available in the context. This service can be customized by implementing `ValidationProviderResolver`. Implementations must be thread-safe.

Listing 6.10: `ValidationProviderResolver` *interface*

```
/**
 * Determines the list of Jakarta Validation providers available in the runtime environment
 * <p>
 * Jakarta Validation providers are identified by the presence of
 * {@code META-INF/services/jakarta.validation.spi.ValidationProvider}
 * files following the Service Provider pattern described
 * <a href="http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Service_Provider">here</a>.
 * <p>
 * Each {@code META-INF/services/jakarta.validation.spi.ValidationProvider} file contains the
 * list of {@link ValidationProvider} implementations each of them representing a provider.
 * <p>
 * Implementations must be thread-safe.
 *
```

```
 * @author Emmanuel Bernard
 */
public interface ValidationProviderResolver {

    /**
     * Returns a list of {@link ValidationProvider} available in the runtime environment.
     *
     * @return list of validation providers
     */
    List<ValidationProvider<?>> getValidationProviders();
}
```

By default, providers are resolved using the Service Provider pattern described in http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html#Service%20Provider. Jakarta Validation providers must supply a service provider configuration file by creating a text file `jakarta.validation.spi.ValidationProvider` and placing it in the `META-INF/services` directory of one of its jar files. The content of the file contains the name of the provider implementation class of the `jakarta.validation.spi.ValidationProvider` interface.

Jakarta Validation 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 according to the guidelines in the JAR file specification.

The default `ValidationProviderResolver` implementation will locate all the Jakarta Validation providers by their provider configuration files visible in the classpath. The default `ValidationProviderResolver` implementation is recommended and custom `ValidationProviderResolver` implementations should be rarely used. A typical use of a custom resolution is resolving providers in a class loader constrained container like OSGi or in a tool environment (IDE).

The default `ValidationProviderResolver` can be accessed via `BootStrapState.getDefaultValidationProviderResolver()`. This method is typically used by the Jakarta Validation provider `Configuration` implementation.

6.5.4.2. ValidationProvider

`ValidationProvider` represents the SPI (Service Provider Interface) defining the contract between the provider discovery and initialization mechanism, and the provider. A `ValidationProvider` does:

- Provide a generic `Configuration` implementation (i.e. not tied to a given provider).
- Provide a provider specific `Configuration` implementation. This `Configuration` will specifically build `ValidatorFactory` instances of the provider it comes from.
- Build a `ValidatorFactory` object from the configuration provided by `ConfigurationState`.

Listing 6.11: `ValidationProvider` *interface*

```
package jakarta.validation.spi;

/**
 * Contract between the validation bootstrap mechanism and the provider engine.
 * <p>
 * Implementations must have a public no-arg constructor. The construction of a provider
 * should be as "lightweight" as possible.
 *
 * @param <T> the provider specific Configuration subclass which typically host provider's
 * additional configuration methods
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ValidationProvider<T extends Configuration<T>> {

    /**
     * Returns a {@link Configuration} instance implementing {@code T},
     * the {@code Configuration} sub-interface.
     * The returned {@code Configuration} instance must use the current provider
     * ({@code this}) to build the {@code ValidatorFactory} instance.
     *
     * @param state bootstrap state
     * @return specific {@code Configuration} implementation
     */
    T createSpecializedConfiguration(BootstrapState state);
```

```
    /**
     * Returns a {@link Configuration} instance. This instance is not bound to
     * use the current provider. The choice of provider follows the algorithm described
     * in {@code Configuration}
     * <p>
     * The {@link ValidationProviderResolver} used by {@code Configuration}
     * is provided by {@code state}.
     * If null, the default {@code ValidationProviderResolver} is used.
     *
     * @param state bootstrap state
     * @return non specialized Configuration implementation
     */
    Configuration<?> createGenericConfiguration(BootstrapState state);

    /**
     * Build a {@link ValidatorFactory} using the current provider implementation.
     * <p>
     * The {@code ValidatorFactory} is assembled and follows the configuration passed
     * via {@link ConfigurationState}.
     * <p>
     * The returned {@code ValidatorFactory} is properly initialized and ready for use.
     *
     * @param configurationState the configuration descriptor
     * @return the instantiated {@code ValidatorFactory}
     * @throws ValidationException if the {@code ValidatorFactory} cannot be built
     */
    ValidatorFactory buildValidatorFactory(ConfigurationState configurationState);
}
```

Listing 6.12: `BootstrapState` *interface*

```
package jakarta.validation.spi;

/**
 * Defines the state used to bootstrap the {@link Configuration}.
 *
 * @author Emmanuel Bernard
 * @author Sebastian Thomschke
 */
public interface BootstrapState {

    /**
     * User defined {@code ValidationProviderResolver} strategy
     * instance or {@code null} if undefined.
     *
     * @return ValidationProviderResolver instance or null
     */
    ValidationProviderResolver getValidationProviderResolver();

    /**
     * Specification default {@code ValidationProviderResolver}
     * strategy instance.
     *
     * @return default implementation of ValidationProviderResolver
     */
    ValidationProviderResolver getDefaultValidationProviderResolver();
}
```

A client can request a specific Jakarta Validation provider by using `<T extends Configuration<T>, U extends ValidationProvider<T>>` `Validation.byProvider(Class<U>)` or by defining the provider in the XML configuration file. The key uniquely identifying a Jakarta Validation provider is the `ValidationProvider` implementation specific to this provider.

A `ValidationProvider` implementation is linked (via its generic parameter) to a specific sub interface of `Configuration`. The Jakarta Validation bootstrap API makes use of this link to return the specific `Configuration` subinterface implementation in a type-safe way when a specific provider is requested. The sub interface does not have to add any new methods but is the natural holder for provider specific configuration methods.

Example 6.23: Example of provider specific Configuration sub interface

```
/**
 * Unique identifier of the ACME provider
 * also hosts some provider specific configuration methods
 */
public interface ACMEConfiguration
    extends Configuration<ACMEConfiguration> {

    /**
     * Enables constraints implementation dynamic reloading when using ACME
     * default to false
     */
    ACMEConfiguration enableDynamicReloading(boolean);

}

/**
 * ACME validation provider
 * Note how ACMEConfiguration and ACMEProvider are linked together
 * via the generic parameter.
 */
public class ACMEProvider implements ValidationProvider<ACMEConfiguration> {
    [...]
}
```

NOTE `Configuration` references itself in the generic definition. Methods of `Configuration` will return the `ACMEConfiguration` making the API easy to use even for vendor specific extensions.

The provider discovery mechanism uses the following algorithm:

- Retrieve available providers using `ValidationProviderResolver.getValidationProviders()`.
- The first `ValidationProvider` matching the requested provider is returned. Providers are evaluated in the order they are returned by `ValidationProviderResolver`. A provider instance is considered matching if it is assignable to the requested provider class.

When the default Jakarta Validation provider is requested, the first `ValidationProvider` returned by the `ValidationProviderResolver` strategy is returned.

Every Jakarta Validation provider must provide a `ValidationProvider` implementation containing a public no-arg constructor and add the corresponding `META-INF/services/jakarta.validation.spi.ValidationProvider` file descriptor in one of its jars.

If a problem occurs while building the `ValidatorFactory`, a `ValidationException` is raised. This can be due to various reasons including:

- malformed XML mapping
- inability to find the provider (or a provider)
- inability to instantiate extension classes provided in the XML configuration
- inconsistent XML mapping (entity declared more than once, incorrect field etc.)
- invalid constraint declaration or definition

6.5.5. Validation

The `Validation` class is the entry point used to bootstrap Jakarta Validation providers. The first entry point, `buildDefaultValidatorFactory()`, is considered to be the default `ValidatorFactory` and is equivalent to the `ValidatorFactory` returned by `Validation.byDefaultProvider().configure().buildValidatorFactory()`.

Example 6.24: Validation methods available

```
 * This class is the entry point for Jakarta Validation.
 * <p>
 * There are three ways to bootstrap it:
 * <ul>
 *     <li>The easiest approach is to build the default {@link ValidatorFactory}.
 *     <pre>
 * ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
 * </pre>
```

```
 *     In this case, the default validation provider resolver
 *     will be used to locate available providers.
 *     <p>
 *     The chosen provider is defined as followed:
 *     <ul>
 *         <li>if the XML configuration defines a provider, this provider is used</li>
 *         <li>if the XML configuration does not define a provider or if no XML
 *         configuration is present the first provider returned by the
 *         {@link ValidationProviderResolver} instance is used.</li>
 *     </ul>
 *     </li>
 *     <li>
 *     The second bootstrap approach allows to choose a custom
 *     {@code ValidationProviderResolver}. The chosen
 *     {@link ValidationProvider} is then determined in the same way
 *     as in the default bootstrapping case (see above).
 *     <pre>
 * Configuration&lt;?&gt; configuration = Validation
 *    .byDefaultProvider()
 *    .providerResolver( new MyResolverStrategy() )
 *    .configure();
 * ValidatorFactory factory = configuration.buildValidatorFactory();
 * </pre>
 *     </li>
 *     <li>
 *     The third approach allows you to specify explicitly and in
 *     a type safe fashion the expected provider.
 *     <p>
 *     Optionally you can choose a custom {@code ValidationProviderResolver}.
 *     <pre>
 * ACMEConfiguration configuration = Validation
 *    .byProvider(ACMEProvider.class)
 *    .providerResolver( new MyResolverStrategy() )  // optionally set the provider resolver
 *    .configure();
 * ValidatorFactory factory = configuration.buildValidatorFactory();
 * </pre>
 *     </li>
 * </ul>
 * <p>
 * Note:
 * <ul>
 *     <li>
 *     The {@code ValidatorFactory} object built by the bootstrap process should be cached
 *     and shared amongst {@code Validator} consumers.
 *     </li>
 *     <li>This class is thread-safe.</li>
 * </ul>
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public final class Validation {

    private Validation() {
    }

    /**
     * Builds and returns a {@link ValidatorFactory} instance based on the
     * default Jakarta Validation provider and following the XML configuration.
     * <p>
     * The provider list is resolved using the default validation provider resolver
     * logic.
     * <p>
     * The code is semantically equivalent to
     * {@code Validation.byDefaultProvider().configure().buildValidatorFactory()}.
     *
     * @return {@code ValidatorFactory} instance
     *
     * @throws NoProviderFoundException if no Jakarta Validation provider was found
     * @throws ValidationException if a Jakarta Validation provider was found but the
```

```
     * {@code ValidatorFactory} cannot be built
        [...]
    public static ValidatorFactory buildDefaultValidatorFactory() {
        return byDefaultProvider().configure().buildValidatorFactory();
    }

    /**
     * Builds a {@link Configuration}. The provider list is resolved
     * using the strategy provided to the bootstrap state.
     * <pre>
     * Configuration&lt;?&gt; configuration = Validation
     *    .byDefaultProvider()
     *    .providerResolver( new MyResolverStrategy() )
     *    .configure();
     * ValidatorFactory factory = configuration.buildValidatorFactory();
     * </pre>
     * The provider can be specified in the XML configuration. If the XML
     * configuration does not exist or if no provider is specified,
     * the first available provider will be returned.
     *
     * @return instance building a generic {@code Configuration}
     *         compliant with the bootstrap state provided
        [...]
    public static GenericBootstrap byDefaultProvider() {
        return new GenericBootstrapImpl();
    }

    /**
     * Builds a {@link Configuration} for a particular provider implementation.
     * <p>
     * Optionally overrides the provider resolution strategy used to determine the provider.
     * <p>
     * Used by applications targeting a specific provider programmatically.
     * <pre>
     * ACMEConfiguration configuration =
     *     Validation.byProvider(ACMEProvider.class)
     *             .providerResolver( new MyResolverStrategy() )
     *             .configure();
     * </pre>,
     * where {@code ACMEConfiguration} is the
     * {@code Configuration} sub interface uniquely identifying the
     * ACME Jakarta Validation provider. and {@code ACMEProvider} is the
     * {@link ValidationProvider} implementation of the ACME provider.
     *
     * @param providerType the {@code ValidationProvider} implementation type
     * @param <T> the type of the {@code Configuration} corresponding to this
     *        {@code ValidationProvider}
     * @param <U> the type of the {@code ValidationProvider} implementation
     *
     * @return instance building a provider specific {@code Configuration}
     *         sub interface implementation
     */
        [...]
    }

    [...]
}
```

The second entry point lets the client provide a custom `ValidationProviderResolver` instance. This instance is passed to `GenericBootstrap`. `GenericBootstrap` builds a generic `Configuration` using the first `ValidationProvider` returned by `ValidationProviderResolution` and calling `ValidationProvider.createGenericConfiguration(BootstrapState state)`. `BootstrapState` holds the `ValidationProviderResolution` instance passed to `GenericBootstrap` and will be used by the `Configuration` instance when resolving the provider to use. Note that `ValidationProvider.createGenericConfiguration` returns a `Configuration` object not bound to any particular provider.

Listing 6.13: `GenericBootstrap` *interface*

```
package jakarta.validation.bootstrap;
```

```
/**
 * Defines the state used to bootstrap Jakarta Validation and
 * creates a provider agnostic {@link Configuration}.
 *
 * @author Emmanuel Bernard
 */
public interface GenericBootstrap {

    /**
     * Defines the provider resolution strategy.
     * This resolver returns the list of providers evaluated
     * to build the {@link Configuration}.
     * <p>
     * If no resolver is defined, the default {@link ValidationProviderResolver}
     * implementation is used.
     *
     * @param resolver the {@code ValidationProviderResolver} to use for bootstrapping
     * @return {@code this} following the chaining method pattern
     */
    GenericBootstrap providerResolver(ValidationProviderResolver resolver);

    /**
     * Returns a generic {@link Configuration} implementation.
     * At this stage the provider used to build the {@link ValidatorFactory}
     * is not defined.
     * <p>
     * The {@code Configuration} implementation is provided by the first provider
     * returned by the {@link ValidationProviderResolver} strategy.
     *
     * @return a {@code Configuration} implementation compliant with the bootstrap state
     * @throws NoProviderFoundException if no Jakarta Validation provider was found
     * @throws ValidationException if a Jakarta Validation provider was found but the
     *         {@code Configuration} object cannot be built; this is generally due to an
     *         issue with the {@code ValidationProviderResolver}
     */
    Configuration<?> configure();
}
```

The last entry point lets the client define the specific Jakarta Validation provider requested as well as a custom `ValidationProviderResolver` implementation if needed. The entry point method, `Validation.byProvider(Class<U> providerType)`, takes the provider specific `ValidationProvider` implementation type and returns a `ProviderSpecificBootstrap` object that guarantees to return an instance of the specific `Configuration` sub interface. Thanks to the use of generics, the client API does not have to cast to the `Configuration` sub interface.

A `ProviderSpecificBootstrap` object can optionally receive a `ValidationProviderResolver` instance.

Listing 6.14: `ProviderSpecificBootstrap` *interface*

```
package jakarta.validation.bootstrap;

/**
 * Defines the state used to bootstrap Jakarta Validation and
 * creates a provider specific {@link Configuration}
 * of type {@code T}.
 * <p>
 * The specific {@code Configuration} is linked to the provider via the generic
 * parameter of the {@link ValidationProvider} implementation.
 * <p>
 * The requested provider is the first provider instance assignable to
 * the requested provider type (known when {@link ProviderSpecificBootstrap} is built).
 * The list of providers evaluated is returned by {@link ValidationProviderResolver}.
 * If no {@code ValidationProviderResolver} is defined, the
 * default {@code ValidationProviderResolver} strategy is used.
 *
 * @param <T> the provider specific {@link Configuration} type
 *
 * @author Emmanuel Bernard
```

```
 */
public interface ProviderSpecificBootstrap<T extends Configuration<T>> {

    /**
     * Optionally defines the provider resolver implementation used.
     * If not defined, use the default {@link ValidationProviderResolver}
     *
     * @param resolver {@code ValidationProviderResolver} implementation used
     *
     * @return {@code this} following the chaining method pattern
     */
    public ProviderSpecificBootstrap<T> providerResolver(
            ValidationProviderResolver resolver);

    /**
     * Determines the provider implementation suitable for {@code T} and delegates
     * the creation of this specific {@link Configuration} subclass to the provider.
     *
     * @return {@code Configuration} sub interface implementation
     *
     * @throws ValidationException if the {@code Configuration} object cannot be built;
     *         this is generally due to an issue with the {@code ValidationProviderResolver}
     */
    public T configure();
}
```

`ProviderSpecificBootstrap.configure()` must return the result of `ValidationProvider.createSpecializedConfiguration(BootstrapState state)`. The state parameter holds the `ValidationProviderResolver` passed to `ProviderSpecificBootstrap`. The validation provider instance used is the one assignable to the type passed as a parameter in `Validation.byProvider(Class)`. The validation provider is selected according to the algorithm described in ValidationProvider.

The `Validation` implementation must not contain any non private attribute or method aside from the three public static bootstrap methods:

- `public static ValidatorFactory buildDefaultValidatorFactory()`
- `public static GenericBootstrap byDefaultProvider()`
- `public static <T extends Configuration<T>, U extends ValidationProvider<T>> ProviderSpecificBootstrap<T> byProvider(Class<U> providerType)`

The bootstrap API is designed to allow complete portability among Jakarta Validation provider implementations. The bootstrap implementation must ensure it can bootstrap third party providers.

When bootstrapping a Jakarta Validation provider, if the `ValidationProviderResolver` either fails or if the expected provider is not found, a `ValidationException` is raised.

6.5.6. XML configuration: META-INF/validation.xml

Unless explicitly ignored by calling `Configuration.ignoreXMLConfiguration()`, a `Configuration` takes into account the configuration available in `META-INF/validation.xml`. This configuration file is optional but can be used by applications to refine some of the Jakarta Validation behavior. If more than one `META-INF/validation.xml` file is found in the classpath, a `ValidationException` is raised.

Unless stated otherwise, XML based configuration settings are overridden by values explicitly set via the `Configuration` API. For example, the `MessageInterpolator` defined via `Configuration.messageInterpolator(MessageInterpolator)` has priority over the `message-interpolator` definition.

`default-provider`: represents the class name of the provider specific `ValidationProvider` implementation class. If defined, the specific provider is used (unless a specific provider has been chosen via the programmatic approach).

`message-interpolator`: represents the fully qualified class name of the `MessageInterpolator` implementation. When defined in XML, the implementation must have a public no-arg constructor.

`traversable-resolver`: represents the fully qualified class name of the `TraversableResolver` implementation. When defined in XML, the implementation must have a public no-arg constructor.

`constraint-validator-factory`: represents the fully qualified class name of the `ConstraintValidatorFactory` implementation. When defined in XML, the implementation must have a public no-arg constructor.

`parameter-name-provider`: represents the fully qualified class name of the `ParameterNameProvider` implementation. When defined in XML,

the implementation must have a public no-arg constructor.

`clock-provider`: represents the fully qualified class name of the `ClockProvider` implementation. When defined in XML, the implementation must have a public no-arg constructor.

`value-extractor`: represents the fully qualified class name of a `ValueExtractor` implementation. `value-extractor` can be given several times for declaring multiple extractors. When defined in XML, the implementation must have a public no-arg constructor. An extractor for a given type and type parameter configured via XML takes precedence over any extractor for the same type and type parameter detected through the service loader or provided by the Jakarta Validation implementation itself. If more than one value extractor for the same type and type parameter is configured via XML, a `ValueExtractorDeclarationException` is raised.

`executable-validation`: allows to disable executable validation entirely via its attribute `enabled="false"` and optionally contains `default-validated-executable-types`. `enabled` defaults to `true`.

`default-validated-executable-types`: declared under `executable-validation`, contains the list of `executable-type` that are considered by default by the integration technology validating executables upon execution.

`constraint-mapping`: represents the resource path of an XML mapping file. More than one `constraint-mapping` element can be present. Mappings provided via `Configuration.addMapping(InputStream)` are added to the list of mappings described via `constraint-mapping`.

`property`: represents a key/value pair property providing room to provider specific configurations. Vendors should use vendor namespaces for properties (e.g., `com.acme.validation.logging`). Entries that make use of the namespace `jakarta.validation` and its subnamespaces must not be used for vendor-specific information. The namespace `jakarta.validation` is reserved for use by this specification. Properties defined via `Configuration.addProperty(String, String)` are added to the properties defined via `property`. If a property with the same name are defined in both XML and via the programmatic API, the value provided via programmatic API has priority.

All these top level elements are optional.

If a public no-arg constructor is missing on any of the classes referenced by the relevant XML elements, a `ValidationException` is raised during the `Configuration.buildValidatorFactory()` call.

Example 6.25: Example of META-INF/validation.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
        xmlns="https://jakarta.ee/xml/ns/validation/configuration"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/configuration
            https://jakarta.ee/xml/ns/validation/configuration/validation-configuration-3.0.xsd"
        version="3.0">
    <default-provider>com.acme.ACMEProvider</default-provider>
    <message-interpolator>com.acme.ACMEAwareMessageInterpolator</message-interpolator>

    <executable-validation>
        <default-validated-executable-types>
            <executable-type>NONE</executable-type>
        </default-validated-executable-types>
    </executable-validation>

    <constraint-mapping>META-INF/validation/order-constraints.xml</constraint-mapping>
    <constraint-mapping>META-INF/validation/catalog-constraints.xml</constraint-mapping>
    <constraint-mapping>META-INF/validation/customer-constraints.xml</constraint-mapping>

    <property name="com.acme.validation.logging">WARN</property>
    <property name="com.acme.validation.safetyChecking">failOnError</property>

</validation-config>
```

The XML schema is described in Configuration schema.

6.5.7. Bootstrapping considerations

The Jakarta Validation bootstrap API can be used directly by any application or made available through a container or other framework. In all cases, the following rules apply:

- `ValidatorFactory` is a thread-safe object that should be built once per deployment unit.
- `ValidatorFactory` should be closed when it is no longer needed (e.g. when the unit is undeployed or the server stopped).

- `Validator` is a thread-safe and lightweight object which can be cached by the `ValidatorFactory` instance.

7. Constraint metadata request APIs

The Jakarta Validation specification provides a way to query the constraint repository. This API is expected to be used for tooling support as well as integration with other frameworks, libraries and other specifications. The Jakarta Validation specification aims to provide both a validation engine and a metadata repository for object constraints. Frameworks (EE or SE) in need for constraint definition, validation and metadata will be able to rely on the Jakarta Validation specification for these services avoiding any unnecessary duplication work from an application and infrastructure point of view.

7.1. Validator

The main API to access all metadata related to a given object is `Validator` (see Bootstrapping for more information on how to retrieve a `Validator` instance).

A `Validator` instance hosts the method to access to the metadata repository for a given class. It is recommended to leave the caching of `Validator` instances to the `ValidatorFactory`. `Validator` implementations are thread-safe.

Example 7.1: Validator interface (metadata request API)

```
/**
 * Validates bean instances. Implementations of this interface must be thread-safe.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 * @author Gunnar Morling
 */
public interface Validator {

    [...] //See 5.1
    /**
     * Returns the descriptor object describing bean constraints.
     * <p>
     * The returned object (and associated objects including
     * {@link ConstraintDescriptor}s) are immutable.
     *
     * @param clazz class or interface type evaluated
     * @return the bean descriptor for the specified class
     * @throws IllegalArgumentException if clazz is {@code null}
     * @throws ValidationException if a non recoverable error happens
     *         during the metadata discovery or if some
     *         constraints are invalid.
     */
    BeanDescriptor getConstraintsForClass(Class<?> clazz);

}
```

`getConstraintsForClass()` returns a `BeanDescriptor` object describing the bean level constraints (see Object validation) and providing access to the property level constraints metadata. An `IllegalArgumentException` is raised if the `clazz` parameter is null.

If a constraint definition or declaration hosted by the requested class (or any of its superclasses and interfaces according to the constraint propagation rules) is invalid, a `ValidationException` is raised. This can be a subclass of `ValidationException` like `ConstraintDefinitionException`, `ConstraintDeclarationException` or `UnexpectedTypeException`.

All descriptor types accessible via `getConstraintsForClass()` and introduced in the following sections are located in the package `jakarta.validation.metadata`.

7.2. ElementDescriptor

`ElementDescriptor` is the root interface describing elements hosting constraints. It is used to describe the list of constraints for a given element (whether it be a class, property, method etc.).

Listing 7.1: `ElementDescriptor` *interface and* `Scope` *enum*

```
package jakarta.validation.metadata;
```

```
/**
 * Describes a validated element (class, property, method etc.).
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 * @author Gunnar Morling
 */
public interface ElementDescriptor {

    /**
     * @return returns {@code true} if at least one constraint declaration is present
     *         for this element in the class hierarchy, {@code false} otherwise
     */
    boolean hasConstraints();

    /**
     * @return the statically defined returned type
     */
    Class<?> getElementClass();

    /**
     * Returns all constraint descriptors for this element in the class hierarchy
     * or an empty {@code Set} if none are present.
     *
     * @return {@code Set} of constraint descriptors for this element
     */
    Set<ConstraintDescriptor<?>> getConstraintDescriptors();

    /**
     * Finds constraints and potentially restricts them to certain criteria.
     *
     * @return {@code ConstraintFinder} object
     */
    ConstraintFinder findConstraints();

    /**
     * Declares restrictions on retrieved constraints.
     * Restrictions are cumulative.
     * <p>
     * A {@code ConstraintFinder} is not thread-safe. The set of matching
     * {@link ConstraintDescriptor} is.
     */
    interface ConstraintFinder {

        /**
         * Restricts to the constraints matching a given set of groups for this element.
         * <p>
         * This method respects group conversion, group sequences
         * and group inheritance (including class-level {@link Default} group
         * overriding) but does not return {@link ConstraintDescriptor}s
         * in any particular order.
         * Specifically, ordering of the group sequence is not respected.
         *
         * @param groups groups targeted
         * @return {@code this} following the chaining method pattern
         */
        ConstraintFinder unorderedAndMatchingGroups(Class<?>... groups);

        /**
         * Restricts to the constraints matching the provided scope for this element.
         *
         * Defaults to {@link Scope#HIERARCHY}
         *
         * @param scope expected scope
         * @return {@code this} following the chaining method pattern
         */
        ConstraintFinder lookingAt(Scope scope);

        /**
         * Restricts to the constraints hosted on the listed {@code types}
```

```
             * for a given element.
             * <p>
             * Defaults to all possible types of the element.
             * <p>
             * Typically used to restrict to fields ({@code FIELD})
             * or getters ({@code METHOD}).
             *
             * @param types targeted types
             *
             * @return {@code this} following the chaining method pattern
             */
            ConstraintFinder declaredOn(ElementType... types);

            /**
             * Retrieves the constraint descriptors following the defined
             * restrictions and hosted on the element described by
             * {@link ElementDescriptor}.
             *
             * @return matching constraint descriptors
             */
            Set<ConstraintDescriptor<?>> getConstraintDescriptors();

            /**
             * Returns {@code true} if at least one constraint declaration
             * matching the restrictions is present on the element,
             * {@code false} otherwise.
             *
             * @return {@code true} if there is at least one constraint
             */
            boolean hasConstraints();
        }
    }
```

Listing 7.2: Scope *enum*

```
    package jakarta.validation.metadata;

    /**
     * Scope looked at when discovering constraints.
     *
     * @author Emmanuel Bernard
     */
    public enum Scope {

        /**
         * Look for constraints declared on the current class element
         * and ignore inheritance and elements with the same name in
         * the class hierarchy.
         */
        LOCAL_ELEMENT,

        /**
         * Look for constraints declared on all elements of the class hierarchy
         * with the same name.
         */
        HIERARCHY
    }
```

`getElementClass()` returns

- the object type when invoked on `BeanDescriptor`,
- the type of a property or parameter when invoked on `PropertyDescriptor` or `ParameterDescriptor` respectively,
- `Object[].class` when invoked on `CrossParameterDescriptor`,
- the return type when invoked on `ConstructorDescriptor`, `MethodDescriptor` or `ReturnValueDescriptor`,
- the container element type when invoked on `ContainerElementTypeDescriptor` (e.g. when invoked on a descriptor representing the

container element type of `List<String>`, `String.class` will be returned).

`getConstraintDescriptors()` returns all the `ConstraintDescriptor`s (see ConstraintDescriptor) hosted on the given element in the class hierarchy, each `ConstraintDescriptor` describing one of the constraints declared on the given element.

`hasConstraints()` returns `true` if the given element in the class hierarchy holds at least one constraint declaration.

If you need to query the metadata API in a more fine grained way for example by restricting the constraints to the ones described on fields or on getters or by restricting to a given set of groups, you can use the `ConstraintFinder` fluent API by calling `findConstraints()`.

`unorderedAndMatchingGroups()` restricts the results to the `ConstraintDescriptor`s (see ConstraintDescriptor) matching the given groups. Order is not respected but group inheritance and inheritance via sequence (including the `Default` group overriding at the class level) are honored.

`declaredOn()` lets you restrict the list of element types constraints are hosted on. This is particularly useful to retrieve property constraints only hosted on fields (`ElementType.FIELD`) or only hosted on getters (`ElementType.METHOD`).

`lookingAt()` lets you restrict which constraints are considered. Either constraints belonging to the element but hosted on the class represented by the given descriptor (`Scope.LOCAL_ELEMENT`), or constraints belonging to the element but hosted anywhere in the class hierarchy (`Scope.HIERARCHY`).

Here is an example restricting the list of constraints on getters, matching the default group and declared physically on the `name` getter of `Customer` (and not any of the getters on the super classes).

Example 7.2: Using the fluent API to restrict matching constraints

```
public class User {

    @Size(max=50)
    String getName() {
        [...]
    }

    [...]
}

public class Customer extends User {

    @NotNull
    String getName() {
        [...]
    }
}

PropertyDescriptor pd =
    validator.getConstraintsForClass(Customer.class).getConstraintsForProperty("name");
Set<ConstraintDescriptor<?>> constraints =
    pd.findConstraints()
        .declaredOn(ElementType.METHOD)
        .unorderedAndMatchingGroups(Default.class)
        .lookingAt(Scope.LOCAL_ELEMENT)
        .getConstraintDescriptors();

assert 1 == constraints.size();

constraints = pd.getConstraintDescriptors();
//equivalent to pd.findConstraints().getConstraintDescriptors();
assert 2 == constraints.size();
```

The following example shows how the fluent API is used to retrieve parameter, cross-parameter and return value constraints, taking into account locally declared constraints as well as constraints declared in the inheritance hierarchy.

Example 7.3: Using the fluent API to select method and constructor constraints

```
public class User {

    public User(@Size(max=50) String name) {
        [...]
```

```
    }

    @PasswordParametersMatch
    @NotNull
    public String resetPassword(
        @NotNull @Size(min=8) String password,
        @NotNull @Size(min=8) String confirmation) {
        [...]
    }
}

public class Customer extends User {

    public Customer(@NotNull String name) {
        [...]
    }

    @Size(min=8)
    public String resetPassword(String password, String confirmation) {
        [...]
    }
}

MethodDescriptor methodDescriptor = validator
    .getConstraintsForClass( Customer.class )
    .getConstraintsForMethod( "resetPassword", String.class, String.class );

//one cross-parameter constraint
assert 1 == methodDescriptor.getCrossParameterDescriptor().getConstraintDescriptors().size();

//one local return value constraint
assert 1 == methodDescriptor.getReturnValueDescriptor()
    .findConstraints()
    .lookingAt( Scope.LOCAL_ELEMENT )
    .getConstraintDescriptors()
    .size();

//two return value constraints in the complete hierarchy
assert 2 == methodDescriptor.getReturnValueDescriptor()
    .findConstraints()
    .lookingAt( Scope.HIERARCHY )
    .getConstraintDescriptors()
    .size();

//two parameter constraints, defined on overridden method
assert 2 == methodDescriptor.getParameterDescriptors()
    .get( 0 )
    .getConstraintDescriptors()
    .size();

ConstructorDescriptor constructorDescriptor = validator
    .getConstraintsForClass( Customer.class )
    .getConstraintsForConstructor( String.class );

//one parameter constraint; constraints from super constructor don't apply
assert 1 == constructorDescriptor.getParameterDescriptors()
    .get( 0 )
    .findConstraints()
    .lookingAt( Scope.HIERARCHY )
    .getConstraintDescriptors()
    .size();
```

7.3. BeanDescriptor

The `BeanDescriptor` interface describes a constrained Java Bean. This interface is returned by `Validator.getConstraintsForClass(Class<?>)`.

Listing 7.3: `BeanDescriptor` *interface*

```
package jakarta.validation.metadata;


/**
 * Describes a constrained Java Bean and the constraints associated to it. All
 * objects returned by the methods of this descriptor (and associated objects
 * including {@link ConstraintDescriptor}s) are immutable.
 *
 * @author Emmanuel Bernard
 * @author Gunnar Morling
 */
public interface BeanDescriptor extends ElementDescriptor {

    /**
     * Returns {@code true} if the bean involves validation:
     * <ul>
     *     <li>a constraint is hosted on the bean itself</li>
     *     <li>a constraint is hosted on one of the bean properties</li>
     *     <li>or a bean property is marked for cascaded validation ({@link Valid})</li>
     * </ul>
     * <p>
     * Constrained methods and constructors are ignored.
     *
     * @return {@code true} if the bean involves validation, {@code false} otherwise
     */
    boolean isBeanConstrained();

    /**
     * Returns the property descriptor for a given property.
     * <p>
     * Returns {@code null} if the property does not exist or has no
     * constraint nor is marked as cascaded (see {@link #getConstrainedProperties()})
     * Properties of super types are considered.
     *
     * @param propertyName property evaluated
     * @return the property descriptor for a given property
     * @throws IllegalArgumentException if {@code propertyName} is {@code null}
     */
    PropertyDescriptor getConstraintsForProperty(String propertyName);

    /**
     * Returns a set of property descriptors having at least one constraint defined
     * or marked as cascaded ({@link Valid}).
     * <p>
     * If no property matches, an empty set is returned.
     * Properties of super types are considered.
     *
     * @return the set of {@link PropertyDescriptor}s for the constraint properties; if
     *         there are no constraint properties, the empty set is returned
     */
    Set<PropertyDescriptor> getConstrainedProperties();

    /**
     * Returns a method descriptor for the given method.
     * <p>
     * Returns {@code null} if no method with the given name and parameter types
     * exists or the specified method neither has parameter or return value constraints nor a
     * parameter or return value marked for cascaded validation.
     * Methods of super types are considered.
     *
     * @param methodName the name of the method
     * @param parameterTypes the parameter types of the method
     * @return a method descriptor for the given method
     * @throws IllegalArgumentException if {@code methodName} is {@code null}
     *
     * @since 1.1
     */
```

```
    MethodDescriptor getConstraintsForMethod(String methodName, Class<?>... parameterTypes);

    /**
     * Returns a set with descriptors for the constrained methods of the bean
     * represented by this descriptor.
     * <p>
     * Constrained methods have at least one parameter or return value constraint
     * or at least one parameter or return value marked for cascaded validation.
     * Methods of super types are considered.
     * <p>
     * Only methods matching the given method type(s) are considered.
     *
     * @param methodType method type to consider
     * @param methodTypes remaining optional method types to consider
     * @return a set with descriptors for the constrained methods of this bean;
     *         will be empty if this bean has no constrained methods of the considered
     *         method type(s) but never {@code null}
     *
     * @since 1.1
     */
    Set<MethodDescriptor> getConstrainedMethods(MethodType methodType,
                                                MethodType... methodTypes);

    /**
     * Returns a constructor descriptor for the given constructor.
     * <p>
     * Returns {@code null} if no constructor with the given parameter types
     * exists or the specified constructor neither has parameter or return value
     * constraints nor a parameter or return value marked for cascaded
     * validation.
     *
     * @param parameterTypes the parameter types of the constructor
     * @return a constructor descriptor for the given constructor
     *
     * @since 1.1
     */
    ConstructorDescriptor getConstraintsForConstructor(Class<?>... parameterTypes);

    /**
     * Returns a set with descriptors for the constrained constructors of the
     * bean represented by this descriptor.
     * <p>
     * Constrained constructors have at least one parameter or return value constraint
     * or at least one parameter or return value marked for cascaded validation.
     *
     * @return a set with descriptors for the constrained constructor of this
     *         bean; will be empty if this bean has no constrained constructor
     *         but never {@code null}
     *
     * @since 1.1
     */
    Set<ConstructorDescriptor> getConstrainedConstructors();
}
```

Listing 7.4: `MethodType` *enum*

```
package jakarta.validation.metadata;

/**
 * Represents the type of a method: getter or non getter.
 *
 * @author Emmanuel Bernard
 * @since 1.1
 */
public enum MethodType {

    /**
     * A method following the getter pattern. A getter according to the
```

```
     * JavaBeans specification is a method whose:
     * <ul>
     *     <li>name starts with get, has a return type but no parameter</li>
     *     <li>name starts with is, has a return type and is returning {@code boolean}.</li>
     * </ul>
     */
    GETTER,

    /**
     * A method that does not follow the getter pattern. A getter according to the
     * JavaBeans specification is a method whose:
     * <ul>
     *     <li>name starts with get, has a return type but no parameter</li>
     *     <li>name starts with is, has a return type and is returning {@code boolean}.</li>
     * </ul>
     */
    NON_GETTER
}
```

`isBeanConstrained()` returns `true` if the given class (and superclasses and interfaces) has at least one class-level or property-level constraint or validation cascade. If the method returns false, the Jakarta Validation engine can safely ignore the bean as it will not be impacted by validation.

`getConstraintsForProperty()` returns a `PropertyDescriptor` object describing the property level constraints (See Field and property validation). The property is uniquely identified by its name as per the JavaBeans convention: field level and getter level constraints of the given name are all returned. An `IllegalArgumentException` is raised if the `propertyName` parameter is null.

`getConstrainedProperties()` returns the `PropertyDescriptor`s of the bean properties having at least one constraint or being cascaded (`@Valid` annotation).

`getConstraintsForMethod()` returns a `MethodDescriptor` object describing the method constraints of the given method. The method is uniquely identified by its name and the types of its parameters.

`getConstrainedMethods()` returns the `MethodDescriptor`s of the methods matching the `MethodType`s provided as parameter and having at least one constraint or cascaded parameter or return value.

`getConstraintsForConstructor()` returns a `ConstructorDescriptor` object describing the method constraints of the given constructor. The constructor is uniquely identified by its name and the types of its parameters.

`getConstrainedConstructors()` returns the `ConstructorDescriptor`s of the constructors having at least one constraint or cascaded parameter or return value.

7.4. CascadableDescriptor

The `CascadableDescriptor` interface describes a cascadable element, i.e. an element which can be marked with `@Valid` in order to perform a cascaded validation of the element as described in Graph validation.

Listing 7.5: `CascadableDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Represents a cascadable element.
 *
 * @author Gunnar Morling
 * @since 1.1
 */
public interface CascadableDescriptor {

    /**
     * Whether this element is marked for cascaded validation or not.
     *
     * @return {@code true}, if this element is marked for cascaded validation,
     *         {@code false} otherwise
     */
    boolean isCascaded();

    /**
```

```
     * Returns the group conversions configured for this element.
     *
     * @return a set containing this element's group conversions; an empty set
     *         may be returned if no conversions are configured but never
     *         {@code null}
     */
    Set<GroupConversionDescriptor> getGroupConversions();
}
```

The `isCascaded()` method returns `true` if the element is marked for cascaded validation.

The method `getGroupConversions()` returns a set with the group conversions declared for the cascadable element. An empty set will be returned if no group conversions are configured.

7.5. GroupConversionDescriptor

The `GroupConversionDescriptor` interface describes a group conversion rule configured for a cascadable element as described in Group conversion. It is returned by `CascadableDescriptor.getGroupConversions()`.

Listing 7.6: `GroupConversionDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * A group conversion rule to be applied during cascaded validation. Two group
 * conversion descriptors are considered equal if they have the same
 * {@code from} and {@code to} group respectively.
 *
 * @author Gunnar Morling
 * @see ConvertGroup
 * @since 1.1
 */
public interface GroupConversionDescriptor {

    /**
     * Returns the source group of this conversion rule.
     *
     * @return the source group of this conversion rule
     */
    Class<?> getFrom();

    /**
     * Returns the target group of this conversion rule.
     *
     * @return the target group of this conversion rule
     */
    Class<?> getTo();
}
```

The `getFrom()` method returns the source of a group conversion rule.

The `getTo()` method returns the target of a group conversion rule.

7.6. PropertyDescriptor

The `PropertyDescriptor` interface describes a constrained property of a Java Bean.

This interface is returned by `BeanDescriptor.getConstraintsForProperty(String)` or `BeanDescriptor.getConstrainedProperties()`. Constraints declared on the attribute and the getter of the same name according to the JavaBeans rules are returned by this descriptor.

Listing 7.7: `PropertyDescriptor` *interface*

```
package jakarta.validation.metadata;
```

```
/**
 * Describes a Java Bean property hosting validation constraints.
 *
 * Constraints placed on the attribute and the getter of a given property
 * are all referenced.
 *
 * @author Emmanuel Bernard
 */
public interface PropertyDescriptor extends ElementDescriptor, CascadableDescriptor, ContainerDescriptor {

    /**
     * Name of the property according to the Java Bean specification.
     *
     * @return property name
     */
    String getPropertyName();
}
```

`getPropertyName()` returns the property name as described in ConstraintViolation.

7.7. ExecutableDescriptor, MethodDescriptor and ConstructorDescriptor

The `ExecutableDescriptor` interface describes a constrained method or constructor of a Java type.

Listing 7.8: `ExecutableDescriptor` *interface*

```
package jakarta.validation.metadata;


/**
 * Provides the common functionality of {@link MethodDescriptor} and
 * {@link ConstructorDescriptor}.
 *
 * @author Gunnar Morling
 *
 * @since 1.1
 */
public interface ExecutableDescriptor extends ElementDescriptor {

    /**
     * Returns the method name in case this descriptor represents a method or
     * the non-qualified name of the declaring class in case this descriptor
     * represents a constructor.
     *
     * @return the name of the executable represented by this descriptor
     */
    String getName();

    /**
     * Returns a list of descriptors representing this executable's
     * parameters, in the order of their declaration, including synthetic
     * parameters.
     *
     * @return a list of descriptors representing this executable's
     *         parameters; an empty list will be returned if this executable has
     *         no parameters, but never {@code null}
     */
    List<ParameterDescriptor> getParameterDescriptors();

    /**
     * Returns a descriptor containing the cross-parameter constraints
     * of this executable.
     *
     * @return a descriptor containing the cross-parameter constraints of
     *         this executable
     */
    CrossParameterDescriptor getCrossParameterDescriptor();
```

```
/**
 * Returns a descriptor for this executable's return value.
 * <p>
 * An executable without return value will return a descriptor
 * representing {@code void}. This descriptor will have no constraint
 * associated.
 *
 * @return a descriptor for this executable's return value
 */
ReturnValueDescriptor getReturnValueDescriptor();

/**
 * Returns {@code true} if the executable parameters are constrained either:
 * <ul>
 *     <li>because of a constraint on at least one of the parameters</li>
 *     <li>because of a cascade on at least one of the parameters (via
 *     {@link Valid})</li>
 *     <li>because of at least one cross-parameter constraint</li>
 * </ul>
 * <p>
 * Also returns {@code false} if there is no parameter.
 *
 * @return {@code true} if the executable parameters are constrained
 */
boolean hasConstrainedParameters();

/**
 * Returns {@code true} if the executable return value is constrained
 * either:
 * <ul>
 *     <li>because of a constraint on the return value</li>
 *     <li>because validation is cascaded on the return value (via
 *     {@link Valid})</li>
 * </ul>
 * <p>
 * Also returns {@code false} if there is no return value.
 *
 * @return {@code true} if the executable return value is constrained
 */
boolean hasConstrainedReturnValue();

/**
 * Returns {@code false}.
 * <p>
 * An executable per se does not host constraints, use
 * {@link #getParameterDescriptors()}, {@link #getCrossParameterDescriptor()}
 * and {@link #getReturnValueDescriptor()} to discover constraints.
 *
 * @return {@code false}
 */
@Override
boolean hasConstraints();

/**
 * Returns an empty {@code Set}.
 * <p>
 * An executable per se does not host constraints, use
 * {@link #getParameterDescriptors()}, {@link #getCrossParameterDescriptor()}
 * and {@link #getReturnValueDescriptor()} to discover constraints.
 *
 * @return an empty {@code Set}
 */
@Override
Set<ConstraintDescriptor<?>> getConstraintDescriptors();

/**
 * Returns a finder that will always return an empty {@code Set}.
 * <p>
 * An executable per se does not host constraints, use
```

```
     * {@link #getParameterDescriptors()}, {@link #getCrossParameterDescriptor()}
     * and {@link #getReturnValueDescriptor()} to discover constraints.
     *
     * @return {@code ConstraintFinder} object
     */
    @Override
    ConstraintFinder findConstraints();
}
```

`getName()` returns the name of the represented method (e.g. "placeOrder") respectively the non-qualified name of the declaring class of the represented constructor (e.g. "OrderService").

`getParameterDescriptors()` returns a list of `ParameterDescriptor`s representing the method's or constructor's parameters in order of their declaration, including synthetic parameters. An empty list will be returned in case the method or constructor has no parameters.

`getCrossParameterDescriptor()` returns a descriptor containing cross-parameter constraints of the method or constructor. If no cross-parameter constraint is present, the descriptor will return an empty set of constraint descriptors.

`getReturnValueDescriptor()` returns a descriptor for the method's or constructor's return value. A descriptor representing the special class `void`, without any constraint descriptors, will be returned for executables which have no return value.

`hasConstrainedParameters()` returns `true` if any of the parameters is constrained or cascaded or if the represented executable has at least one cross-parameter constraint. Returns `false` if there is no parameter.

`hasConstrainedReturnValue()` returns `true` if the return value is constrained or cascaded. Returns `false` if there is no return value.

The methods `hasConstraints()`, `getConstraintDescriptors()` and `findConstraints()` defined on `ElementDescriptor` are redefined to clarify that executables do not host constraints directly and thus will always return `false` or an empty set of constraints, respectively. Constraint descriptors for individual parameters can be obtained from the corresponding `ParameterDescriptor` object, constraint descriptors for cross-parameter constraints can be obtained from the corresponding `CrossParameterDescriptor` object and constraint descriptors for the return value can be obtained from `ReturnValueDescriptor`.

The interfaces `MethodDescriptor` and `ConstructorDescriptor` are derived from `ExecutableDescriptor` and allow to distinguish between descriptors representing methods and descriptors representing constructors.

Listing 7.9: `MethodDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Describes a validated method.
 *
 * @author Gunnar Morling
 * @author Emmanuel Bernard
 * @since 1.1
 */
public interface MethodDescriptor extends ExecutableDescriptor {
}
```

Listing 7.10: `ConstructorDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Describes a validated constructor.
 *
 * @author Gunnar Morling
 * @author Emmanuel Bernard
 * @since 1.1
 */
public interface ConstructorDescriptor extends ExecutableDescriptor {
}
```

`MethodDescriptor` objects are returned by `BeanDescriptor.getConstraintsForMethod(String, Class<?>…)` and

`BeanDescriptor.getConstrainedMethods(MethodType, MethodType…)`, while `ConstructorDescriptor` objects are returned by `BeanDescriptor.getConstraintsForConstructor(Class<?>…)` and `BeanDescriptor.getConstrainedConstructors()`.

None of the metadata API methods honor the XML configuration around executable validation nor the presence of `@ValidateOnExecution`. In other words, all constrained methods and constructors will be returned by the metadata API regardless of these settings.

7.8. ParameterDescriptor

The `ParameterDescriptor` interface describes a constrained parameter of a method or constructor.

This interface is returned by `MethodDescriptor.getParameterDescriptors()` and `ConstructorDescriptor.getParameterDescriptors()`.

Listing 7.11: `ParameterDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Describes a validated method or constructor parameter.
 *
 * @author Gunnar Morling
 * @since 1.1
 */
public interface ParameterDescriptor extends ElementDescriptor, CascadableDescriptor,
        ContainerDescriptor {

    /**
     * Returns this parameter's index within the parameter array of the method
     * or constructor holding it.
     *
     * @return this parameter's index
     */
    int getIndex();

    /**
     * Returns this parameter's name as retrieved by the current parameter name
     * resolver.
     *
     * @return this parameter's name
     */
    String getName();
}
```

`getIndex()` returns the index of the represented parameter within the parameter array of the method or constructor holding it.

`getName()` returns the name of the represented parameter.

7.9. CrossParameterDescriptor

The `CrossParameterDescriptor` interface describes an element containing all cross-parameter constraints of a method or constructor.

This interface is returned by `MethodDescriptor.getCrossParameterDescriptor()` and `ConstructorDescriptor.getCrossParameterDescriptor()`.

Listing 7.12: `CrossParameterDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Describes an element holding cross-parameter constraints of a method or constructor
 *
 * @author Emmanuel Bernard
 * @since 1.1
 */
public interface CrossParameterDescriptor extends ElementDescriptor {

    /**
```

```
     * @return {@code Object[].class} - the type of the parameter array
     */
    @Override
    Class<?> getElementClass();
}
```

`getElementClass()` returns `Object[]`.

7.10. ReturnValueDescriptor

The `ReturnValueDescriptor` interface describes the return value of a method or constructor.

This interface is returned by `MethodDescriptor.getReturnValueDescriptor()` and `ConstructorDescriptor.getReturnValueDescriptor()`.

Listing 7.13: `ReturnValueDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Describes a validated return value of a method or constructor.
 *
 * @author Gunnar Morling
 * @since 1.1
 */
public interface ReturnValueDescriptor extends ElementDescriptor, CascadableDescriptor, ContainerDescriptor {
}
```

7.11. ContainerDescriptor and ContainerElementTypeDescriptor

The `ContainerDescriptor` interface describes those elements that can be of a container type, e.g. `List` or `Map`, and as such may host container element constraints (see Container element constraints) or have container element types that are marked with `@Valid`.

`ContainerDescriptor` is extended by `PropertyDescriptor`, `ParameterDescriptor`, `ReturnValueDescriptor` and `ContainerElementTypeDescriptor`.

Listing 7.14: `ContainerDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Represents an element that might be a container, thus allowing container element
 * constraints.
 *
 * @author Guillaume Smet
 * @since 2.0
 */
public interface ContainerDescriptor {

    /**
     * If this element is of a container type, e.g. {@code List} or {@code Map}, a set of
     * descriptors of those container element types is returned, which are constrained or
     * marked with {@link Valid}. A container element type is constrained, if it hosts at
     * least one constraint.
     * <p>
     * In the context of properties and method return values, container element types of
     * super-types are considered.
     *
     * @return the set of descriptors representing the container element types that are
     * constrained or are marked with {@code Valid}. An empty set will be returned if this
     * element is not of a container type or is of a container type but there are no
     * container element types hosting constraints or marked with {@code Valid}.
     */
    Set<ContainerElementTypeDescriptor> getConstrainedContainerElementTypes();
}
```

If a given element is of a container type, `getConstrainedContainerElementTypes()` returns a set with descriptors representing those container element types that either host at least one constraint or are marked with `@Valid`. The returned set will be empty if the given element is not of a container type or is of a container type but has no element types that are constrained or marked with `@Valid`. In the context of properties and method return values, container element types of super-types are considered.

The `ContainerElementTypeDescriptor` interface describes the potential container element constraints applied to one element type of a container.

This interface is returned by `ContainerDescriptor.getConstrainedContainerElementTypes()`.

Listing 7.15: `ContainerElementTypeDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Describes a validated container element type, e.g. the element type of {@code List} if it
 * hosts at least one constraint or is marked with {@link Valid}.
 *
 * @author Guillaume Smet
 * @since 2.0
 */
public interface ContainerElementTypeDescriptor extends ElementDescriptor, CascadableDescriptor,
ContainerDescriptor {

    /**
     * Returns the index of the type argument corresponding to this container element type.
     * @return the index of the type argument corresponding to this container element type
     */
    Integer getTypeArgumentIndex();

    /**
     * Returns the container class hosting this container element type.
     * @return the container class hosting this container element type
     */
    Class<?> getContainerClass();
}
```

`getTypeArgumentIndex()` returns the index of the type argument corresponding to this descriptor instance. `getContainerClass()` returns the type of the container declaring the container element type represented by this descriptor instance.

7.12. ConstraintDescriptor

A `ConstraintDescriptor` object describes a given constraint declaration (i.e. a constraint annotation).

Listing 7.16: `ConstraintDescriptor` *interface*

```
package jakarta.validation.metadata;

/**
 * Describes a single constraint and its composing constraints.
 *
 * @param <T> the constraint's annotation type
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConstraintDescriptor<T extends Annotation> {

    /**
     * Returns the annotation describing the constraint declaration.
     * If a composing constraint, attribute values are reflecting
     * the overridden attributes of the composing constraint
     *
     * @return the annotation for this constraint
     */
    T getAnnotation();
```

```
/**
 * The non-interpolated error message
 *
 * @return the non-interpolated error message
 *
 * @since 1.1
 */
String getMessageTemplate();

/**
 * The set of groups the constraint is applied on.
 * If the constraint declares no group, a set with only the {@link Default}
 * group is returned.
 *
 * @return the groups the constraint is applied on
 */
Set<Class<?>> getGroups();

/**
 * The set of payload the constraint hosts.
 *
 * @return payload classes hosted on the constraint or an empty set if none
 */
Set<Class<? extends Payload>> getPayload();

/**
 * The {@link ConstraintTarget} value of {@code validationAppliesTo} if the constraint
 * hosts it or {@code null} otherwise.
 *
 * @return the {@code ConstraintTarget} value or {@code null}
 *
 * @since 1.1
 */
ConstraintTarget getValidationAppliesTo();

/**
 * List of the constraint validation implementation classes.
 *
 * @return list of the constraint validation implementation classes
 */
List<Class<? extends ConstraintValidator<T, ?>>> getConstraintValidatorClasses();

/**
 * Returns a map containing the annotation attribute names as keys and the
 * annotation attribute values as value.
 * <p>
 * If this constraint is used as part of a composed constraint, attribute
 * values are reflecting the overridden attribute of the composing constraint.
 *
 * @return a map containing the annotation attribute names as keys
 *         and the annotation attribute values as value
 */
Map<String, Object> getAttributes();

/**
 * Returns the value of the annotation attribute.
 * <p>
 * Serves as a shortcut to {@code (V) getAttributes().get(attrName);}.
 * <p>
 * If this constraint is used as part of a composed constraint, attribute
 * values are reflecting the overridden attribute of the composing constraint.
 *
 * @param name the name of the constraint attribute to retrieve
 * @param type the class of the attribute to be returned
 * @return the value of a requested attribute, or {@code null} if such attribute is not defined.
 * @param <V> the type of the attribute to be returned
 * @throws ClassCastException if the actual attribute value cannot be cast to the requested type.
 * @see #getAttributes()
 * @since 4.0
```

```
         */
        <V> V getAttribute(String name, Class<V> type);

        /**
         * Return a set of composing {@link ConstraintDescriptor}s where each
         * descriptor describes a composing constraint. {@code ConstraintDescriptor}
         * instances of composing constraints reflect overridden attribute values in
         * {@link #getAttributes()}  and {@link #getAnnotation()}.
         *
         * @return a set of {@code ConstraintDescriptor} objects or an empty set
         *         in case there are no composing constraints
         */
        Set<ConstraintDescriptor<?>> getComposingConstraints();

        /**
         * @return {@code true} if the constraint is annotated with {@link ReportAsSingleViolation}
         */
        boolean isReportAsSingleViolation();

        /**
         * @return a {@link ValidateUnwrappedValue} describing the unwrapping behavior as given
         * via the {@link Unwrapping} constraint payloads.
         *
         * @since 2.0
         */
        ValidateUnwrappedValue getValueUnwrapping();

        /**
         * Returns an instance of the specified type allowing access to provider-specific APIs.
         * <p>
         * If the Jakarta Validation provider implementation does not support the specified class,
         * a {@link ValidationException} is thrown.
         *
         * @param type the class of the object to be returned
         * @param <U> the type of the object to be returned
         * @return an instance of the specified class
         * @throws ValidationException if the provider does not support the call
         *
         * @since 2.0
         */
        <U> U unwrap(Class<U> type);
    }
```

`getAnnotation()` returns the annotation instance (or an annotation instance representing the given constraint declaration). If `ConstraintDescriptor` represents a composing annotation (see Constraint composition), the returned annotation must reflect parameter overriding. In other words, the annotation parameter values are the overridden values.

`getAttributes()` returns a map containing the annotation attribute names as a key, and the annotation attribute values as a value (this API is anticipated to be simpler to use by tools than reflection over the annotation instance). If `ConstraintDescriptor()` represents a composing annotation (see Constraint composition), the returned `Map` must reflect attribute overriding.

`getMessageTemplate()` returns the non-interpolated error message.

`getGroups()` returns the groups the constraint is supposed to be applied upon. If no group is set on the constraint declaration, the `Default` group is returned. The groups of a composing constraint are the groups of the composed constraint.

`getPayload()` returns the payloads associated to the constraint or an empty set if none. The payload from the main constraint annotation is inherited by the composing annotations. Any payload definition on a composing annotation is ignored.

`getValidationAppliesTo()` returns the `ConstraintTarget` returned by `validationAppliesTo` if the constraint hosts the attribute or `null` otherwise. The constraint target from the main constraint annotation is inherited by the composing annotation. Any constraint target definition on a composing annotation is ignored.

`isReportAsSingleViolation()` returns `true` if the constraint is annotated with `@ReportAsSingleViolation`.

`getComposingConstraints()` return a set of composing `ConstraintDescriptor`s where each descriptor describes a composing constraint.

`getConstraintValidatorClasses()` returns the `ConstraintValidator` classes associated with the constraint.

`getValueUnwrapping()` returns a `ValidateUnwrappedValue` instance describing the unwrapping behavior.

Listing 7.17: `ValidateUnwrappedValue` *enum*

```
package jakarta.validation.metadata;

/**
 * The unwrapping behavior that can be applied to a specific constraint.
 *
 * @author Guillaume Smet
 * @since 2.0
 */
public enum ValidateUnwrappedValue {

    /**
     * No specific unwrapping behavior has been defined for this constraint and the default
     * behavior applies: if there is exactly one maximally-specific type-compliant
     * {@link ValueExtractor} and this extractor is marked with {@link UnwrapByDefault}, this
     * extractor is applied and the constraint is applied to the value(s) wrapped by the
     * annotated container. Otherwise, no value extractor is applied.
     */
    DEFAULT,

    /**
     * The value is unwrapped before validation, i.e. the constraint is applied to the
     * value(s) wrapped by the annotated container.
     */
    UNWRAP,

    /**
     * The value is not unwrapped before validation, i.e. the constraint is applied to the
     * annotated element.
     */
    SKIP;
}
```

`unwrap()` is provided as a way to access objects of a given type specific to a Jakarta Validation provider, exposing functionality complementary to the `ConstraintDescriptor` contract. Using this method makes your code non portable.

7.13. Example

Assuming the following constraint definitions

```
package com.acme.constraint;

@Documented
@Constraint(validatedBy = ValidInterval.Validator.class)
@Target({ METHOD, ANNOTATION_TYPE, CONSTRUCTOR })
@Retention(RUNTIME)
@Repeatable( List.class )
public @interface ValidInterval {

    String message() default "{com.acme.constraint.ValidInterval.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    int startParameter();

    int endParameter();

    @Target({ METHOD, ANNOTATION_TYPE, CONSTRUCTOR })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        ValidInterval[] value();
    }
```

```
    @SupportedValidationTarget(PARAMETERS)
    class Validator implements ConstraintValidator<ValidInterval, Object[]> {

        private int start;
        private int end;

        @Override
        public void initialize(ValidInterval constraintAnnotation) {
            this.start = constraintAnnotation.startParameter();
            this.end = constraintAnnotation.endParameter();
        }

        @Override
        public boolean isValid(Object[] value, ConstraintValidatorContext context) {
            return Integer.parseInt( String.valueOf( value[start] ) ) <
                    Integer.parseInt( String.valueOf( value[end] ) );
        }
    }
}

@Documented
@Constraint(validatedBy = ValidAddress.Validator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable( List.class )
public @interface ValidAddress {

    String message() default "{com.acme.constraint.ValidAddress.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        ValidAddress[] value();
    }

    class Validator implements ConstraintValidator<ValidAddress, Address> {

        @Override
        public boolean isValid(Address value, ConstraintValidatorContext context) {
            // [...]
        }
    }
}
```

and the following class definitions

```
public class Author {

    private String firstName;

    @NotEmpty(message = "lastname must not be null")
    private String lastName;

    @Size(max = 30)
    private String company;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
```

```
    public String getLastName() {
        return lastName;
    }

    public String getCompany() {
        return company;
    }

    public void setCompany(String company) {
        this.company = company;
    }
}

public class Book {

    public interface FirstLevelCheck {
    }

    public interface SecondLevelCheck {
    }

    private String title;
    private String description;

    @Valid
    @NotNull
    private Author author;

    @Valid
    public Book(
            String title,
            @Size(max = 30) String description,
            @Valid @ConvertGroup(from = Default.class, to = SecondLevelCheck.class) Author author) {
        // [...]
    }

    public Book() {
        // [...]
    }

    @NotEmpty(groups = { FirstLevelCheck.class, Default.class })
    @Size(max = 30)
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }

    public String getDescription() {
        return description;
    }

    public void setAuthor(String description) {
        this.description = description;
    }

    @ValidInterval(startParameter = 1, endParameter = 2)
    public void addChapter(String title, int startPage, int endPage) {
        // [...]
```

```
    }
}

public class Account {
    // [...]
}

public class Address {
    // [...]
}

public abstract class Roles implements Set<String> {
    // [...]
}

public interface LegalEntity {

    Iterable<@NotNull String> getRoles();
}

public interface Person extends LegalEntity {

    @Override
    Set<@NotEmpty String> getRoles();

    Map<@NotNull String, @Valid Account> getAccounts();
}

public interface Employee extends Person {

    @Override
    Set<@NotBlank String> getRoles();

    Map<String, List<@NotNull @Valid Address>> getAddresses();
}

public class EmployeeImpl implements Employee {

    @Override
    public Roles getRoles() {
        // [...]
    }

    @Override
    public Map<String, List<@ValidAddress Address>> getAddresses() {
        // [...]
    }

    @Override
    public Map<String, Account> getAccounts() {
        // [...]
    }
}
```

The following assertions are true.

```
BeanDescriptor bookDescriptor = validator.getConstraintsForClass(Book.class);

assert ! bookDescriptor.hasConstraints();

assert bookDescriptor.isBeanConstrained();
assert bookDescriptor.getConstrainedMethods( MethodType.NON_GETTER ).size() > 0;

assert bookDescriptor.getConstraintDescriptors().size() == 0; //no bean-level constraint

//more specifically "author", "title" and "keywordsPerChapter"
assert bookDescriptor.getConstrainedProperties().size() == 2;

//not a property
assert bookDescriptor.getConstraintsForProperty( "doesNotExist" ) == null;
```

```
//property with no constraint
assert bookDescriptor.getConstraintsForProperty( "description" ) == null;

PropertyDescriptor propertyDescriptor = bookDescriptor.getConstraintsForProperty( "title" );
assert propertyDescriptor.getConstraintDescriptors().size() == 2;
assert "title".equals( propertyDescriptor.getPropertyName() );

//assuming the implementation returns the @NotEmpty constraint first
ConstraintDescriptor<?> constraintDescriptor = propertyDescriptor.getConstraintDescriptors()
        .iterator().next();
assert constraintDescriptor.getAnnotation().annotationType().equals( NotEmpty.class );
assert constraintDescriptor.getGroups().size() == 2; //FirstLevelCheck and Default
assert constraintDescriptor.getComposingConstraints().size() == 2;
assert constraintDescriptor.isReportAsSingleViolation() == true;

//@NotEmpty cannot be null
boolean notNullPresence = false;
for ( ConstraintDescriptor<?> composingDescriptor :
            constraintDescriptor.getComposingConstraints() ) {
    if ( composingDescriptor.getAnnotation().annotationType().equals( NotNull.class ) ) {
        notNullPresence = true;
    }
}
assert notNullPresence;

//assuming the implementation returns the Size constraint second
constraintDescriptor = propertyDescriptor.getConstraintDescriptors().iterator().next();
assert constraintDescriptor.getAnnotation().annotationType().equals( Size.class );
assert constraintDescriptor.getAttributes().get( "max" ) == Integer.valueOf( 30 );
assert constraintDescriptor.getGroups().size() == 1;

propertyDescriptor = bookDescriptor.getConstraintsForProperty( "author" );
assert propertyDescriptor.getConstraintDescriptors().size() == 1;
assert propertyDescriptor.isCascaded();

//getTitle() and addChapter()
assert bookDescriptor.getConstrainedMethods( MethodType.GETTER, MethodType.NON_GETTER ).size() ==
        2;

//the constructor accepting title, description and author
assert bookDescriptor.getConstrainedConstructors().size() == 1;

ConstructorDescriptor constructorDescriptor = bookDescriptor.getConstraintsForConstructor(
        String.class, String.class, Author.class
);
assert constructorDescriptor.getName().equals( "Book" );
assert constructorDescriptor.getElementClass() == Book.class;
assert constructorDescriptor.hasConstrainedParameters() == true;

//return value is marked for cascaded validation
assert constructorDescriptor.hasConstrainedReturnValue() == true;

//constraints are retrieved via the sub-descriptors for parameters etc.
assert constructorDescriptor.hasConstraints() == false;

//one descriptor for each parameter
assert constructorDescriptor.getParameterDescriptors().size() == 3;

//"description" parameter
ParameterDescriptor parameterDescriptor = constructorDescriptor.getParameterDescriptors()
        .get( 1 );

//assuming the default parameter name provider is used and parameter names can
//be obtained
assert parameterDescriptor.getName().equals( "description" );
assert parameterDescriptor.getElementClass() == String.class;
assert parameterDescriptor.getIndex() == 1;
assert parameterDescriptor.hasConstraints() == true;
```

```
Set<ConstraintDescriptor<?>> parameterConstraints =
        parameterDescriptor.getConstraintDescriptors();
assert parameterConstraints.iterator().next().getAnnotation().annotationType() == Size.class;

//"author" parameter
parameterDescriptor = constructorDescriptor.getParameterDescriptors().get( 2 );
assert parameterDescriptor.hasConstraints() == false;
assert parameterDescriptor.isCascaded() == true;

//group conversion on "author" parameter
GroupConversionDescriptor groupConversion =
        parameterDescriptor.getGroupConversions().iterator().next();
assert groupConversion.getFrom() == Default.class;
assert groupConversion.getTo() == SecondLevelCheck.class;

//constructor return value
ReturnValueDescriptor returnValueDescriptor = constructorDescriptor.getReturnValueDescriptor();
assert returnValueDescriptor.hasConstraints() == false;
assert returnValueDescriptor.isCascaded() == true;

//a getter is also a method which is constrained on its return value
MethodDescriptor methodDescriptor = bookDescriptor.getConstraintsForMethod( "getTitle" );
assert methodDescriptor.getName().equals( "getTitle" );
assert methodDescriptor.getElementClass() == String.class;
assert methodDescriptor.hasConstrainedParameters() == false;
assert methodDescriptor.hasConstrainedReturnValue() == true;
assert methodDescriptor.hasConstraints() == false;

returnValueDescriptor = methodDescriptor.getReturnValueDescriptor();
assert returnValueDescriptor.getElementClass() == String.class;
assert returnValueDescriptor.getConstraintDescriptors().size() == 2;
assert returnValueDescriptor.isCascaded() == false;

//void method which has a cross-parameter constraint
methodDescriptor = bookDescriptor.getConstraintsForMethod(
        "addChapter", String.class, int.class, int.class
);
assert methodDescriptor.getElementClass() == void.class;
assert methodDescriptor.hasConstrainedParameters() == true;
assert methodDescriptor.hasConstrainedReturnValue() == false;

//cross-parameter constraints accessible via separate descriptor
assert methodDescriptor.hasConstraints() == false;

assert methodDescriptor.getReturnValueDescriptor().getElementClass() == void.class;

//cross-parameter descriptor
CrossParameterDescriptor crossParameterDescriptor =
        methodDescriptor.getCrossParameterDescriptor();
assert crossParameterDescriptor.getElementClass() == Object[].class;
assert crossParameterDescriptor.hasConstraints() == true;

ConstraintDescriptor<?> crossParameterConstraint =
        crossParameterDescriptor.getConstraintDescriptors().iterator().next();
assert crossParameterConstraint.getAnnotation().annotationType() == ValidInterval.class;

// no constrained container element types for title
assert bookDescriptor.getConstraintsForProperty( "title" )
        .getConstrainedContainerElementTypes().size() == 0;

BeanDescriptor employeeImplDescriptor = validator.getConstraintsForClass( Employee.class );

// container element constraints for property "roles"
PropertyDescriptor rolesDescriptor = employeeImplDescriptor.getConstraintsForProperty( "roles" );
assert rolesDescriptor != null;

Set<ContainerElementTypeDescriptor> constrainedContainerElementTypes = rolesDescriptor
        .getConstrainedContainerElementTypes();
// the container element types of Set and Iterable; Roles does not declare any container element types itself
assert constrainedContainerElementTypes.size() == 2;
```

```
Iterator<ContainerElementTypeDescriptor> it = constrainedContainerElementTypes.iterator();

// assuming that the descriptor for Set is returned first
ContainerElementTypeDescriptor containerElementTypeDescriptor = it.next();
assert containerElementTypeDescriptor.getContainerClass() == Set.class;
assert containerElementTypeDescriptor.getTypeArgumentIndex() == 0;
assert containerElementTypeDescriptor.getElementClass() == String.class;
// @NotEmpty and @NotBlank
assert containerElementTypeDescriptor.getConstraintDescriptors().size() == 2;

// assuming that the descriptor for Iterable is returned next
containerElementTypeDescriptor = it.next();
assert containerElementTypeDescriptor.getContainerClass() == Iterable.class;
assert containerElementTypeDescriptor.getTypeArgumentIndex() == 0;
assert containerElementTypeDescriptor.getElementClass() == String.class;
// @NotNull
assert containerElementTypeDescriptor.getConstraintDescriptors().size() == 1;

// container element constraints for property "accounts"
PropertyDescriptor accountsDescriptor = employeeImplDescriptor
        .getConstraintsForProperty( "accounts" );
constrainedContainerElementTypes = accountsDescriptor.getConstrainedContainerElementTypes();
// the map key type and the map value type
assert constrainedContainerElementTypes.size() == 2;

it = constrainedContainerElementTypes.iterator();

// assuming that the descriptor for the map key is returned first
containerElementTypeDescriptor = it.next();
assert containerElementTypeDescriptor.getContainerClass() == Map.class;
assert containerElementTypeDescriptor.getTypeArgumentIndex() == 0;
assert containerElementTypeDescriptor.getElementClass() == String.class;
// @NotNull
assert containerElementTypeDescriptor.getConstraintDescriptors().size() == 1;
assert containerElementTypeDescriptor.isCascaded() == false;

// assuming that the descriptor for the map value is returned next
containerElementTypeDescriptor = it.next();
assert containerElementTypeDescriptor.getContainerClass() == Map.class;
assert containerElementTypeDescriptor.getTypeArgumentIndex() == 1;
assert containerElementTypeDescriptor.getElementClass() == Account.class;
assert containerElementTypeDescriptor.getConstraintDescriptors().size() == 0;
assert containerElementTypeDescriptor.isCascaded() == true;

// container element constraints for property "addresses"
PropertyDescriptor addressesDescriptor = employeeImplDescriptor
        .getConstraintsForProperty( "addresses" );
constrainedContainerElementTypes = addressesDescriptor.getConstrainedContainerElementTypes();
// the map value type
assert constrainedContainerElementTypes.size() == 1;

it = constrainedContainerElementTypes.iterator();

containerElementTypeDescriptor = it.next();
assert containerElementTypeDescriptor.getContainerClass() == Map.class;
assert containerElementTypeDescriptor.getTypeArgumentIndex() == 1;
assert containerElementTypeDescriptor.getElementClass() == List.class;
// No constraints nor @Valid on List itself
assert containerElementTypeDescriptor.getConstraintDescriptors().size() == 0;
assert containerElementTypeDescriptor.isCascaded() == false;

// container element type of the nested List container
constrainedContainerElementTypes = containerElementTypeDescriptor.getConstrainedContainerElementTypes();
assert constrainedContainerElementTypes.size() == 1;
it = constrainedContainerElementTypes.iterator();

containerElementTypeDescriptor = it.next();
assert containerElementTypeDescriptor.getContainerClass() == List.class;
assert containerElementTypeDescriptor.getTypeArgumentIndex() == 0;
```

```
assert containerElementTypeDescriptor.getElementClass() == Address.class;
// @NotNull and @ValidAddress
assert containerElementTypeDescriptor.getConstraintDescriptors().size() == 2;
assert containerElementTypeDescriptor.isCascaded() == true;
```

8. Built-in Constraint definitions

The specification defines a small set of built-in constraints. Their usage is encouraged both in regular constraint declarations and as composing constraints. Using this set of constraints will enhance portability of your constraints across constraint-consuming frameworks relying on the metadata API (such as client side validation frameworks or database schema generation frameworks).

Built-in annotations are annotated with an empty `@Constraint` annotation to avoid any dependency between the specification API and a specific implementation. Each Jakarta Validation provider must recognize built-in constraint annotations as valid constraint definitions and provide compliant constraint implementations for each. The built-in constraint validation implementation is having a lower priority than an XML mapping definition. In other words `ConstraintValidator` implementations for built-in constraints can be overridden by using the XML mapping (see Overriding constraint definitions in XML).

All built-in constraints are in the `jakarta.validation.constraints` package. Here is the list of constraints and their declaration.

8.1. `@Null` constraint

Listing 8.1: `@Null` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be {@code null}.
 * Accepts any type.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Null {

    String message() default "{jakarta.validation.constraints.Null.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@link Null} annotations on the same element.
     *
     * @see jakarta.validation.constraints.Null
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Null[] value();
    }
}
```

8.2. `@NotNull` constraint

Listing 8.2: `@NotNull` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must not be {@code null}.
 * Accepts any type.
 *
 * @author Emmanuel Bernard
 */
```

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface NotNull {

    String message() default "{jakarta.validation.constraints.NotNull.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@link NotNull} annotations on the same element.
     *
     * @see jakarta.validation.constraints.NotNull
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        NotNull[] value();
    }
}
```

8.3. `@AssertTrue` constraint

Listing 8.3: `@AssertTrue` *constraint*

```
package jakarta.validation.constraints;

/**
 * The annotated element must be true.
 * Supported types are {@code boolean} and {@code Boolean}.
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface AssertTrue {

    String message() default "{jakarta.validation.constraints.AssertTrue.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@link AssertTrue} annotations on the same element.
     *
     * @see AssertTrue
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        AssertTrue[] value();
    }
}
```

8.4. `@AssertFalse` constraint

Listing 8.4: `@AssertFalse` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be false.
 * Supported types are {@code boolean} and {@code Boolean}.
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface AssertFalse {

    String message() default "{jakarta.validation.constraints.AssertFalse.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@link AssertFalse} annotations on the same element.
     *
     * @see jakarta.validation.constraints.AssertFalse
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        AssertFalse[] value();
    }
}
```

8.5. `@Min` constraint

Listing 8.5: `@Min` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a number whose value must be higher or
 * equal to the specified minimum.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code CharSequence}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, and their respective
 *     wrappers</li>
 * </ul>
 * Note that {@code double} and {@code float} are not supported due to rounding errors
 * (some providers might provide some approximative support).
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
```

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Min {

    String message() default "{jakarta.validation.constraints.Min.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * @return value the element must be higher or equal to
     */
    long value();

    /**
     * Defines several {@link Min} annotations on the same element.
     *
     * @see Min
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Min[] value();
    }
}
```

8.6. `@Max` constraint

Listing 8.6: `@Max` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a number whose value must be lower or
 * equal to the specified maximum.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code CharSequence}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, and their respective
 *     wrappers</li>
 * </ul>
 * Note that {@code double} and {@code float} are not supported due to rounding errors
 * (some providers might provide some approximative support).
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Max {

    String message() default "{jakarta.validation.constraints.Max.message}";

    Class<?>[] groups() default { };
```

```
    Class<? extends Payload>[] payload() default { };

    /**
     * @return value the element must be lower or equal to
     */
    long value();

    /**
     * Defines several {@link Max} annotations on the same element.
     *
     * @see Max
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Max[] value();
    }
}
```

8.7. `@DecimalMin` constraint

Listing 8.7: `@DecimalMin` *constraint*

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a number whose value must be higher or
 * equal to the specified minimum.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code CharSequence}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, and their respective
 *     wrappers</li>
 * </ul>
 * Note that {@code double} and {@code float} are not supported due to rounding errors
 * (some providers might provide some approximative support).
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface DecimalMin {

    String message() default "{jakarta.validation.constraints.DecimalMin.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * The {@code String} representation of the min value according to the
     * {@code BigDecimal} string representation.
     *
     * @return value the element must be higher or equal to
     */
    String value();
```

```
    /**
     * Specifies whether the specified minimum is inclusive or exclusive.
     * By default, it is inclusive.
     *
     * @return {@code true} if the value must be higher or equal to the specified minimum,
     *         {@code false} if the value must be higher
     *
     * @since 1.1
     */
    boolean inclusive() default true;

    /**
     * Defines several {@link DecimalMin} annotations on the same element.
     *
     * @see DecimalMin
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        DecimalMin[] value();
    }
}
```

8.8. `@DecimalMax` constraint

Listing 8.8: `@DecimalMax` *constraint*

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a number whose value must be lower or
 * equal to the specified maximum.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code CharSequence}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, and their respective
 *     wrappers</li>
 * </ul>
 * Note that {@code double} and {@code float} are not supported due to rounding errors
 * (some providers might provide some approximative support).
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface DecimalMax {

    String message() default "{jakarta.validation.constraints.DecimalMax.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * The {@code String} representation of the max value according to the
     * {@code BigDecimal} string representation.
```

```
     *
     * @return value the element must be lower or equal to
     */
    String value();

    /**
     * Specifies whether the specified maximum is inclusive or exclusive.
     * By default, it is inclusive.
     *
     * @return {@code true} if the value must be lower or equal to the specified maximum,
     *         {@code false} if the value must be lower
     *
     * @since 1.1
     */
    boolean inclusive() default true;

    /**
     * Defines several {@link DecimalMax} annotations on the same element.
     *
     * @see DecimalMax
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        DecimalMax[] value();
    }
}
```

8.9. `@Negative` constraint

Listing 8.9: `@Negative` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a strictly negative number (i.e. 0 is considered as an
 * invalid value).
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, {@code float},
 *     {@code double} and their respective wrappers</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Gunnar Morling
 * @since 2.0
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Negative {

    String message() default "{jakarta.validation.constraints.Negative.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
```

```
     * Defines several {@link Negative} constraints on the same element.
     *
     * @see Negative
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Negative[] value();
    }
}
```

8.10. `@NegativeOrZero` constraint

Listing 8.10: `@NegativeOrZero` *constraint*

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a negative number or 0.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, {@code float},
 *     {@code double} and their respective wrappers</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Gunnar Morling
 * @since 2.0
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface NegativeOrZero {

    String message() default "{jakarta.validation.constraints.NegativeOrZero.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@link NegativeOrZero} constraints on the same element.
     *
     * @see NegativeOrZero
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        NegativeOrZero[] value();
    }
}
```

8.11. `@Positive` constraint

Listing 8.11: `@Positive` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a strictly positive number (i.e. 0 is considered as an
 * invalid value).
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, {@code float},
 *     {@code double} and their respective wrappers</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Gunnar Morling
 * @since 2.0
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Positive {

    String message() default "{jakarta.validation.constraints.Positive.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@link Positive} constraints on the same element.
     *
     * @see Positive
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Positive[] value();
    }
}
```

8.12. `@PositiveOrZero` constraint

Listing 8.12: `@PositiveOrZero` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a positive number or 0.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, {@code float},
 *     {@code double} and their respective wrappers</li>
 * </ul>
```

```
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Gunnar Morling
 * @since 2.0
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface PositiveOrZero {

    String message() default "{jakarta.validation.constraints.PositiveOrZero.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@link PositiveOrZero} constraints on the same element.
     *
     * @see PositiveOrZero
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        PositiveOrZero[] value();
    }
}
```

8.13. `@Size` constraint

Listing 8.13: `@Size` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element size must be between the specified boundaries (included).
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code CharSequence} (length of character sequence is evaluated)</li>
 *     <li>{@code Collection} (collection size is evaluated)</li>
 *     <li>{@code Map} (map size is evaluated)</li>
 *     <li>Array (array length is evaluated)</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Size {

    String message() default "{jakarta.validation.constraints.Size.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };
```

```
    /**
     * @return size the element must be higher or equal to
     */
    int min() default 0;

    /**
     * @return size the element must be lower or equal to
     */
    int max() default Integer.MAX_VALUE;

    /**
     * Defines several {@link Size} annotations on the same element.
     *
     * @see Size
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Size[] value();
    }
}
```

8.14. `@Digits` constraint

Listing 8.14: `@Digits` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be a number within accepted range.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code BigDecimal}</li>
 *     <li>{@code BigInteger}</li>
 *     <li>{@code CharSequence}</li>
 *     <li>{@code byte}, {@code short}, {@code int}, {@code long}, and their respective
 *     wrapper types</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Digits {

    String message() default "{jakarta.validation.constraints.Digits.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * @return maximum number of integral digits accepted for this number
     */
    int integer();

    /**
     * @return maximum number of fractional digits accepted for this number
     */
```

```
    int fraction();

    /**
     * Defines several {@link Digits} annotations on the same element.
     *
     * @see Digits
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Digits[] value();
    }
}
```

8.15. `@Past` constraint

Listing 8.15: `@Past` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be an instant, date or time in the past.
 * <p>
 * <i>Now</i> is defined by the {@link ClockProvider} attached to the {@link Validator} or
 * {@link ValidatorFactory}. The default {@code clockProvider} defines the current time
 * according to the virtual machine, applying the current default time zone if needed.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code java.util.Date}</li>
 *     <li>{@code java.util.Calendar}</li>
 *     <li>{@code java.time.Instant}</li>
 *     <li>{@code java.time.LocalDate}</li>
 *     <li>{@code java.time.LocalDateTime}</li>
 *     <li>{@code java.time.LocalTime}</li>
 *     <li>{@code java.time.MonthDay}</li>
 *     <li>{@code java.time.OffsetDateTime}</li>
 *     <li>{@code java.time.OffsetTime}</li>
 *     <li>{@code java.time.Year}</li>
 *     <li>{@code java.time.YearMonth}</li>
 *     <li>{@code java.time.ZonedDateTime}</li>
 *     <li>{@code java.time.chrono.HijrahDate}</li>
 *     <li>{@code java.time.chrono.JapaneseDate}</li>
 *     <li>{@code java.time.chrono.MinguoDate}</li>
 *     <li>{@code java.time.chrono.ThaiBuddhistDate}</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Past {

    String message() default "{jakarta.validation.constraints.Past.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
```

```
     * Defines several {@link Past} annotations on the same element.
     *
     * @see Past
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Past[] value();
    }
}
```

8.16. `@PastOrPresent` constraint

Listing 8.16: `@PastOrPresent` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be an instant, date or time in the past or in the present.
 * <p>
 * <i>Now</i> is defined by the {@link ClockProvider} attached to the {@link Validator} or
 * {@link ValidatorFactory}. The default {@code clockProvider} defines the current time
 * according to the virtual machine, applying the current default time zone if needed.
 * <p>
 * The notion of present is defined relatively to the type on which the constraint is
 * used. For instance, if the constraint is on a {@link Year}, present would mean the whole
 * current year.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code java.util.Date}</li>
 *     <li>{@code java.util.Calendar}</li>
 *     <li>{@code java.time.Instant}</li>
 *     <li>{@code java.time.LocalDate}</li>
 *     <li>{@code java.time.LocalDateTime}</li>
 *     <li>{@code java.time.LocalTime}</li>
 *     <li>{@code java.time.MonthDay}</li>
 *     <li>{@code java.time.OffsetDateTime}</li>
 *     <li>{@code java.time.OffsetTime}</li>
 *     <li>{@code java.time.Year}</li>
 *     <li>{@code java.time.YearMonth}</li>
 *     <li>{@code java.time.ZonedDateTime}</li>
 *     <li>{@code java.time.chrono.HijrahDate}</li>
 *     <li>{@code java.time.chrono.JapaneseDate}</li>
 *     <li>{@code java.time.chrono.MinguoDate}</li>
 *     <li>{@code java.time.chrono.ThaiBuddhistDate}</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Guillaume Smet
 * @since 2.0
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface PastOrPresent {

    String message() default "{jakarta.validation.constraints.PastOrPresent.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };
```

```
    /**
     * Defines several {@link PastOrPresent} annotations on the same element.
     *
     * @see PastOrPresent
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        PastOrPresent[] value();
    }
}
```

8.17. `@Future` constraint

Listing 8.17: `@Future` *constraint*

```
package jakarta.validation.constraints;

/**
 * The annotated element must be an instant, date or time in the future.
 * <p>
 * <i>Now</i> is defined by the {@link ClockProvider} attached to the {@link Validator} or
 * {@link ValidatorFactory}. The default {@code clockProvider} defines the current time
 * according to the virtual machine, applying the current default time zone if needed.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code java.util.Date}</li>
 *     <li>{@code java.util.Calendar}</li>
 *     <li>{@code java.time.Instant}</li>
 *     <li>{@code java.time.LocalDate}</li>
 *     <li>{@code java.time.LocalDateTime}</li>
 *     <li>{@code java.time.LocalTime}</li>
 *     <li>{@code java.time.MonthDay}</li>
 *     <li>{@code java.time.OffsetDateTime}</li>
 *     <li>{@code java.time.OffsetTime}</li>
 *     <li>{@code java.time.Year}</li>
 *     <li>{@code java.time.YearMonth}</li>
 *     <li>{@code java.time.ZonedDateTime}</li>
 *     <li>{@code java.time.chrono.HijrahDate}</li>
 *     <li>{@code java.time.chrono.JapaneseDate}</li>
 *     <li>{@code java.time.chrono.MinguoDate}</li>
 *     <li>{@code java.time.chrono.ThaiBuddhistDate}</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Future {

    String message() default "{jakarta.validation.constraints.Future.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@link Future} annotations on the same element.
```

```
     *
     * @see Future
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        Future[] value();
    }
}
```

8.18. `@FutureOrPresent` constraint

Listing 8.18: `@FutureOrPresent` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must be an instant, date or time in the present or in the future.
 * <p>
 * <i>Now</i> is defined by the {@link ClockProvider} attached to the {@link Validator} or
 * {@link ValidatorFactory}. The default {@code clockProvider} defines the current time
 * according to the virtual machine, applying the current default time zone if needed.
 * <p>
 * The notion of present here is defined relatively to the type on which the constraint is
 * used. For instance, if the constraint is on a {@link Year}, present would mean the whole
 * current year.
 * <p>
 * Supported types are:
 * <ul>
 *     <li>{@code java.util.Date}</li>
 *     <li>{@code java.util.Calendar}</li>
 *     <li>{@code java.time.Instant}</li>
 *     <li>{@code java.time.LocalDate}</li>
 *     <li>{@code java.time.LocalDateTime}</li>
 *     <li>{@code java.time.LocalTime}</li>
 *     <li>{@code java.time.MonthDay}</li>
 *     <li>{@code java.time.OffsetDateTime}</li>
 *     <li>{@code java.time.OffsetTime}</li>
 *     <li>{@code java.time.Year}</li>
 *     <li>{@code java.time.YearMonth}</li>
 *     <li>{@code java.time.ZonedDateTime}</li>
 *     <li>{@code java.time.chrono.HijrahDate}</li>
 *     <li>{@code java.time.chrono.JapaneseDate}</li>
 *     <li>{@code java.time.chrono.MinguoDate}</li>
 *     <li>{@code java.time.chrono.ThaiBuddhistDate}</li>
 * </ul>
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Guillaume Smet
 * @since 2.0
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface FutureOrPresent {

    String message() default "{jakarta.validation.constraints.FutureOrPresent.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };
```

```
    /**
     * Defines several {@link FutureOrPresent} annotations on the same element.
     *
     * @see FutureOrPresent
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    @interface List {

        FutureOrPresent[] value();
    }
}
```

8.19. `@Pattern` constraint

Listing 8.19: `@Pattern` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated {@code CharSequence} must match the specified regular expression.
 * The regular expression follows the Java regular expression conventions
 * see {@link java.util.regex.Pattern}.
 * <p>
 * Accepts {@code CharSequence}. {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Pattern {

    /**
     * @return the regular expression to match
     */
    String regexp();

    /**
     * @return array of {@code Flag}s considered when resolving the regular expression
     */
    Flag[] flags() default { };

    /**
     * @return the error message template
     */
    String message() default "{jakarta.validation.constraints.Pattern.message}";

    /**
     * @return the groups the constraint belongs to
     */
    Class<?>[] groups() default { };

    /**
     * @return the payload associated to the constraint
     */
    Class<? extends Payload>[] payload() default { };

    /**
     * Possible Regexp flags.
     */
    public static enum Flag {

        /**
```

```
         * Enables Unix lines mode.
         *
         * @see java.util.regex.Pattern#UNIX_LINES
         */
        UNIX_LINES( java.util.regex.Pattern.UNIX_LINES ),

        /**
         * Enables case-insensitive matching.
         *
         * @see java.util.regex.Pattern#CASE_INSENSITIVE
         */
        CASE_INSENSITIVE( java.util.regex.Pattern.CASE_INSENSITIVE ),

        /**
         * Permits whitespace and comments in pattern.
         *
         * @see java.util.regex.Pattern#COMMENTS
         */
        COMMENTS( java.util.regex.Pattern.COMMENTS ),

        /**
         * Enables multiline mode.
         *
         * @see java.util.regex.Pattern#MULTILINE
         */
        MULTILINE( java.util.regex.Pattern.MULTILINE ),

        /**
         * Enables dotall mode.
         *
         * @see java.util.regex.Pattern#DOTALL
         */
        DOTALL( java.util.regex.Pattern.DOTALL ),

        /**
         * Enables Unicode-aware case folding.
         *
         * @see java.util.regex.Pattern#UNICODE_CASE
         */
        UNICODE_CASE( java.util.regex.Pattern.UNICODE_CASE ),

        /**
         * Enables canonical equivalence.
         *
         * @see java.util.regex.Pattern#CANON_EQ
         */
        CANON_EQ( java.util.regex.Pattern.CANON_EQ );

        //JDK flag value
        private final int value;

        private Flag(int value) {
            this.value = value;
        }

        /**
         * @return flag value as defined in {@link java.util.regex.Pattern}
         */
        public int getValue() {
            return value;
        }
    }

    /**
     * Defines several {@link Pattern} annotations on the same element.
     *
     * @see Pattern
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
```

```
    @Documented
    @interface List {

        Pattern[] value();
    }
}
```

8.20. `@NotEmpty` constraint

Listing 8.20: `@NotEmpty` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must not be {@code null} nor empty.
 * <p>
 * Supported types are:
 * <ul>
 * <li>{@code CharSequence} (length of character sequence is evaluated)</li>
 * <li>{@code Collection} (collection size is evaluated)</li>
 * <li>{@code Map} (map size is evaluated)</li>
 * <li>Array (array length is evaluated)</li>
 * </ul>
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 *
 * @since 2.0
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface NotEmpty {

    String message() default "{jakarta.validation.constraints.NotEmpty.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@code @NotEmpty} constraints on the same element.
     *
     * @see NotEmpty
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    public @interface List {
        NotEmpty[] value();
    }
}
```

8.21. `@NotBlank` constraint

Listing 8.21: `@NotBlank` constraint

```
package jakarta.validation.constraints;

/**
 * The annotated element must not be {@code null} and must contain at least one
 * non-whitespace character. Accepts {@code CharSequence}.
```

```
 *
 * @author Hardy Ferentschik
 * @since 2.0
 *
 * @see Character#isWhitespace(char)
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface NotBlank {

    String message() default "{jakarta.validation.constraints.NotBlank.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * Defines several {@code @NotBlank} constraints on the same element.
     *
     * @see NotBlank
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    public @interface List {
        NotBlank[] value();
    }
}
```

8.22. `@Email` constraint

Listing 8.22: `@Email` constraint

```
package jakarta.validation.constraints;

/**
 * The string has to be a well-formed email address. Exact semantics of what makes up a valid
 * email address are left to Jakarta Validation providers. Accepts {@code CharSequence}.
 * <p>
 * {@code null} elements are considered valid.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 *
 * @since 2.0
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
@Repeatable(List.class)
@Documented
@Constraint
public @interface Email {

    String message() default "{jakarta.validation.constraints.Email.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

    /**
     * @return an additional regular expression the annotated element must match. The default
     * is any string ('.*')
     */
    String regexp() default ".*";
```

```
    /**
     * @return used in combination with {@link #regexp()} in order to specify a regular
     * expression option
     */
    Pattern.Flag[] flags() default { };

    /**
     * Defines several {@code @Email} constraints on the same element.
     *
     * @see Email
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
    @Retention(RUNTIME)
    @Documented
    public @interface List {
        Email[] value();
    }
}
```

9. XML deployment descriptor

Two kinds of XML descriptors are used by Jakarta Validation. The first one describes the Jakarta Validation configuration provided as `META-INF/validation.xml`. The second one describes constraints declarations and closely matches the annotations declaration approach. If an XML descriptor does not validate against the corresponding XSD file, a `ValidationException` is raised.

9.1. Constraint definition and declaration

Jakarta Validation lets you declare constraints via XML rather than annotations. You can either ignore constraints declared via annotations or consider XML as adding additional constraints on top of annotation constraints. While it is not possible to define a new constraint via XML, you can redefine the list of `ConstraintValidator` classes associated to a given constraint definition.

There is no distinction between an annotation based constraint declaration and an XML based constraint declaration: they are considered equivalent and should be treated as such by the Jakarta Validation provider. The rest of the specification only refers to annotations as validation metadata: it should be read as annotation or their XML descriptor equivalent.

Specifically when exploring metadata, the Jakarta Validation provider must ensure that an annotation instance corresponding to the XML declaration is provided via `ConstraintDescriptor.getAnnnotation()`. The annotation elements as well as `ConstraintValidator.getAttributes()` must reflect the values described in the XML declaration (see Converting the string representation of a value). Likewise, `ConstraintDescriptor.getConstraintValidatorClasses()` must reflect XML based constraint definition overriding (see Overriding constraint definitions in XML).

A given class must not be described more than once among all the XML mapping descriptors. A given field or getter must not be described more than once on a given class description. A given constraint definition must not be overridden more than once among all the XML mapping descriptors. If any of these rules is violated in a given validation deployment, a `ValidationException` is raised during the creation of the `ValidatorFactory`.

The schema is provided in XML Schema.

9.1.1. Constraint declaration in XML

If `default-package` is set, all unqualified class names (including annotations) are considered part of the package described by `default-package`.

A given JavaBean is described by the `bean` element. The name of the class is mandatory. By default, all constraint declarations expressed via annotations are ignored for classes described in XML. You can force Jakarta Validation to consider both annotations and XML constraint declarations by using `ignore-annotations="false"` on `bean`.

NOTE The `ignore-annotations` setting is not inherited from nor by the class hierarchy. In other words, it applies to the current bean only.

If the name of the class does refer to a class not present in the classpath, a `ValidationException` is raised.

Example 9.1: Example of bean XML declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
    xmlns="https://jakarta.ee/xml/ns/validation/mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        https://jakarta.ee/xml/ns/validation/mapping
        https://jakarta.ee/xml/ns/validation/validation-mapping-4.0.xsd"
    version="4.0">

    <default-package>com.acme.app.domain</default-package>

    <bean class="Customer" ignore-annotations="false">
        [...]
    </bean>
    <bean class="com.acme.common.model.Address">
        [...]
    </bean>
</constraint-mappings>
```

9.1.1.1. Class-level overriding

Class level annotations are described via the `class` element. If `ignore-annotations` is declared, Jakarta Validation must honor the explicit value for this element. If not declared, the default value defined in the encapsulating `bean` element is considered.

When `ignore-annotations` is true, class-level Jakarta Validation annotations are ignored for this class (including the `@GroupSequence`). When `ignore-annotations` is false:

- Constraints declared in XML and constraints declared in annotations are added and form the list of class-level declared constraints.
- `@GroupSequence` is considered unless `group-sequence` element is explicitly used.

Example 9.2: Example of class-level declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
    xmlns="https://jakarta.ee/xml/ns/validation/mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        https://jakarta.ee/xml/ns/validation/mapping
        https://jakarta.ee/xml/ns/validation/validation-mapping-4.0.xsd"
    version="4.0">
    <default-package>com.acme.app.domain</default-package>
    <bean class="Customer" ignore-annotations="false">
        <class ignore-annotations="true">
            [...]
        </class>
    </bean>
    <bean class="com.acme.common.model.Address">
        <class>
            [...]
        </class>
    </bean>
</constraint-mappings>
```

9.1.1.2. Field-level overriding

Field level annotations are described via the `field` element. The `name` attribute corresponds to the name of the field considered. If `ignore-annotations` is declared, Jakarta Validation must honor the explicit value for this element. If not declared, the default value defined in the encapsulating `bean` element is considered.

When `ignore-annotations` is true, field-level Jakarta Validation annotations on the targeted field are ignored (including `@Valid` and `@ConvertGroup`). When `ignore-annotations` is false:

- Constraints declared in XML and constraints declared in annotations are added and form the list of field-level declared constraints.
- `@Valid` is considered unless the `valid` element is explicitly used. Note that the only way to disable cascading on a field marked as `@Valid` is to use `ignore-annotations=true`.
- Group conversions declared in XML and via the `@ConvertGroup` annotation are added and form the list of applied conversions. Note that the rules for the declaration of group conversions as outlined in Group conversion apply, in particular it is not legal to declare several conversions for the same source group.

If the name of the field does not correspond to a field in the given bean a `ValidationException` is raised.

Example 9.3: Field-level declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/mapping/validation-mapping-4.0.xsd"
        version="4.0">
    <default-package>com.acme.app.domain</default-package>
    <bean class="Customer" ignore-annotations="false">
        <field name="firstName">
            [...]
        </field>
```

```
        <field name="orders">
            <valid/>
            [...]
        </field>
    </bean>
</constraint-mappings>
```

9.1.1.3. Property-level overriding

Property-level annotations are described via the `getter` element. The `name` attribute corresponds to the name of the property considered as defined in Field and property validation (for example a getter `String getAge()` would have `<getter name="age"/>` as a corresponding descriptor). If `ignore-annotations` is declared, Jakarta Validation must honor the explicit value for this element. If not declared, the default value defined in the encapsulating `bean` element is considered.

When `ignore-annotations` is true, property-level Jakarta Validation annotations on the targeted property are ignored (including `@Valid` and `@ConvertGroup`). When `ignore-annotations` is false:

- Constraints declared in XML and constraints declared in annotations are added and form the list of property-level declared constraints.
- `@Valid` is considered unless the `valid` element is explicitly used. Note that the only way to disable cascading on a property marked as `@Valid` is to use `ignore-annotations=true`.
- Group conversions declared in XML and via the `@ConvertGroup` annotation are added and form the list of applied conversions. Note that the rules for the declaration of group conversions as outlined in Group conversion apply, in particular it is not legal to declare several conversions for the same source group.

If the name of the property does not correspond to a property in the given bean a `ValidationException` is raised.

Example 9.4: Property-level declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/mapping/validation-mapping-4.0.xsd"
        version="4.0">
    <default-package>com.acme.app.domain</default-package>
    <bean class="Customer" ignore-annotations="false">
        <getter name="firstName">
            [...]
        </getter>
        <getter name="orders">
            <valid/>
            [...]
        </getter>
    </bean>
</constraint-mappings>
```

9.1.1.4. Constructor-level overriding

Constructor-level annotations are described via the `constructor` element.

To identify a constructor to be configured, zero or more `parameter` elements are used, matching the number and types of parameters of the configured constructor. When configuring the default constructor, no `parameter` element is to be used. The parameter types are specified using their fully qualified name using the syntax described in the documentation of `java.lang.Class.getName()`.

Let's look at some examples:

- `"java.lang.String"` must be specified for a parameter of type `java.lang.String`
- `"long"` must be specified for a parameter of type `long`
- `"[Ljava.lang.Object;"` must be specified for a parameter of type `java.lang.Object[]`

Varargs parameters are specified using the corresponding array type, e.g. a parameter `String...` must be specified as `"[Ljava.lang.String;"`.

If the `default-package` element is configured for the mapping file, any unqualified class names will be resolved using the given default

package.

NOTE You must declare all parameters even if they are not reconfigured to ensure the right constructor is identified.

If no constructor with the specified parameter types exists in the given bean a `ValidationException` is raised.

The optional `return-value` element is used to change the configuration of a constructor's return value if required.

The optional `cross-parameter` element is used to change the configuration of a constructor's cross-parameter constraints if required.

The constraints applying for a constructor's parameters and its return value are specified by adding `constraint` elements to the `parameter` and `return-value` elements respectively. Whether or not to perform cascaded validation is controlled using the `valid` element . Group conversion rules for cascaded validation are specified using the `convert-group` element.

The cross-parameter constraints applied on a constructor parameter list are specified by adding `constraint` elements to the `cross-parameter` element.

If `ignore-annotations` is declared on the `parameter`, `cross-parameter` or `return-value` element, Jakarta Validation must honor the explicit value for this element. Otherwise, if `ignore-annotations` is declared for the `constructor` element, Jakarta Validation must honor this value. Otherwise, the default value declared in the encapsulating `bean` element is considered.

When `ignore-annotations` is true, Jakarta Validation annotations on the targeted constructor or parameter are ignored (including `@Valid` and `@ConvertGroup`). When `ignore-annotations` is false:

- Constraints declared in XML and constraints declared in annotations are added and form the list of declared parameter, cross-parameter or return value constraints respectively.
- `@Valid` is considered unless the `valid` element is explicitly used. Note that the only way to disable cascading on a constructor parameter or return value marked as `@Valid` is to use `ignore-annotations=true`. This does not apply to cross-parameter elements as cascading does not make sense in this situation.
- Group conversions declared in XML and via the `@ConvertGroup` annotation are added and form the list of applied conversions. Note that the rules for the declaration of group conversions as outlined in Group conversion apply, in particular it is not legal to declare several conversions for the same source group. This does not apply to cross-parameter elements as cascading does not make sense in this situation.

Example 9.5: Constructor-level declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/mapping/validation-mapping-4.0.xsd"
        version="4.0">
    <default-package>com.acme.app.domain</default-package>
    <bean class="Customer" ignore-annotations="false">
        <constructor ignore-annotations="true">
            <parameter type="java.lang.String">
                [...]
            </parameter>
            <parameter type="int">
                <valid/>
                [...]
            </parameter>
            <parameter type="long" ignore-annotations="false"/>
            <cross-parameter ignore-annotations="false">
                [...]
            </cross-parameter>
            <return-value>
                <valid/>
                [...]
            </return-value>
            [...]
        </constructor>

    </bean>
</constraint-mappings>
```

9.1.1.5. Method-level overriding

Method-level annotations are described via the `method` element.

To identify a method to be configured, zero or more `parameter` elements are used, matching the number and types of parameters of the configured method. The parameter types are specified using their fully qualified name using the syntax described in the documentation of `java.lang.Class.getName()`.

Let's look at some examples:

- `"java.lang.String"` must be specified for a parameter of type `java.lang.String`
- `"long"` must be specified for a parameter of type `long`
- `"[Ljava.lang.Object;"` must be specified for a parameter of type `java.lang.Object[]`

Varargs parameters are specified using the corresponding array type, e.g. a parameter `String...` must be specified as `"[Ljava.lang.String;"`.

If the `default-package` element is configured for the mapping file, any unqualified class names will be resolved using the given default package.

NOTE You must declare all parameters even if they are not reconfigured to ensure the right method is identified.

NOTE A given getter method representing a JavaBeans property may either be configured using the `getter` or the `method` element, but not both. If a `getter` element and a `method` element referring to the same method are detected by the Jakarta Validation provider, a `ValidationException` is raised.

If no method with the specified name and parameter types exists in the given bean a `ValidationException` is raised.

The optional `return-value` element is used to change the configuration of a method's return value if required.

The optional `cross-parameter` element is used to change the configuration of a method's cross-parameter constraints if required.

The constraints applying for a method's parameters and its return value are specified by adding `constraint` elements to the `parameter` and `return-value` elements respectively. Whether or not to perform cascaded validation is controlled using the `valid` element. Group conversion rules for cascaded validation are specified using the `convert-group` element.

The cross-parameter constraints applied on a method parameter list are specified by adding `constraint` elements to the `cross-parameter` element.

If `ignore-annotations` is declared on the `parameter`, `cross-parameter` or `return-value` element, Jakarta Validation must honor the explicit value for this element. Otherwise, if `ignore-annotations` is declared for the `method` element, Jakarta Validation must honor this value. Otherwise, the default value declared in the encapsulating `bean` element is considered.

When `ignore-annotations` is true, Jakarta Validation annotations on the targeted method or parameter are ignored (including `@Valid` and `@ConvertGroup`). When `ignore-annotations` is false:

- Constraints declared in XML and constraints declared in annotations are added and form the list of declared parameter, cross-parameter or return value constraints respectively.
- `@Valid` is considered unless the `valid` element is explicitly used. Note that the only way to disable cascading on a method parameter or return value marked as `@Valid` is to use `ignore-annotations=true`. This does not apply to cross-parameter elements as cascading does not make sense in this situation.
- Group conversions declared in XML and via the `@ConvertGroup` annotation are added and form the list of applied conversions. Note that the rules for the declaration of group conversions as outlined in Group conversion apply, in particular it is not legal to declare several conversions for the same source group. This does not apply to cross-parameter elements as cascading does not make sense in this situation.

Example 9.6: Method-level declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/mapping/validation-mapping-4.0.xsd"
        version="4.0">
    <default-package>com.acme.app.domain</default-package>
    <bean class="Customer" ignore-annotations="false">
        <method name="update" ignore-annotations="true">
```

```
            <parameter type="java.lang.String">
                [...]
            </parameter>
            <parameter type="int">
                <valid/>
                [...]
            </parameter>
            <parameter type="long" ignore-annotations="false"/>
            <cross-parameter ignore-annotations="false">
                [...]
            </cross-parameter>
            <return-value>
                <valid/>
                [...]
            </return-value>
            [...]
        </method>

    </bean>
</constraint-mappings>
```

9.1.1.6. Container-element overriding

To apply constraints to the elements of generic container types or to mark them for cascaded validation, the `container-element-type` element is used.

`container-element-type` can be used within the `field`, `getter`, `parameter` and `return-value` elements.

The `type-argument-index` is used to specify the index of the configured type argument. The `type-argument-index` can be omitted, if the container type has exactly one type argument. The `ignore-annotations` settings effectively applying to the encapsulating element (`field`, `getter` etc.) are applied to `container-element-type`, too. The `container-element-type` element can be nested for configuring nested generic containers such as `List<List<String>>`.

Constraints are applied by adding `constraint` elements to `container-element-type`. Whether or not to perform cascaded validation is controlled using the `valid` element. Group conversion rules for cascaded validation are specified using the `convert-group` element.

If an invalid container element type configuration is detected, a `ValidationException` is raised. This includes the following configuration errors:

- The type of the surrounding element (field, getter etc.) has no type arguments.
- The type of the surrounding element has no type argument with the index given via `type-argument-index`.
- The type of the surrounding element has multiple type arguments and no index is given via `type-argument-index`.
- The same type argument of the surrounding element is configured multiple times.

Example 9.7: Container-element declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/mapping/validation-mapping-4.0.xsd"
        version="4.0">
    <default-package>com.acme.app.domain</default-package>
    <bean class="Customer" ignore-annotations="false">
        <!-- Map<String, Address> -->
        <field name="addressesByType" ignore-annotations="true">
            <container-element-type type-argument-index="0">
                [...]
            </container-element-type>
            <container-element-type type-argument-index="1">
                <valid/>
                [...]
            </container-element-type>
            [...]
        </field>
```

```
        <!-- setContactsByType(Map<String, List<String>>) -->
        <method name="setContactsByType" ignore-annotations="true">
            <parameter type="java.util.Map">
                <container-element-type type-argument-index="1">
                    <valid/>
                    <container-element-type type-argument-index="0">
                        <valid/>
                        [...]
                    </container-element-type>
                    [...]
                </container-element-type>
                [...]
            </parameter>
            [...]
        </method>

    </bean>
</constraint-mappings>
```

9.1.1.7. Constraint declaration

New constraint declarations are represented by the `constraint` element. The `annotation` attribute is the class name of the annotation representing the constraint. Message, groups and payload are defined respectively by the `message`, `groups` and `payload` elements.

Other custom elements of an annotation are represented by `element`. The `name` attribute is mandatory and represents the name of the element in the constraint declaration. `message`, `groups` and `payload` are not permitted names, use the `message`, `groups` or `payload` elements instead. Otherwise a `ValidationException` is raised.

NOTE `validationAppliesTo` (see validationAppliesTo) is not necessary as cross-parameter constraints and return value constraints are declared in different XML elements, respectively `cross-parameter` and `return-value`.

If the element represents a primitive type, a class or an enum, the string representation of its value is placed in the element itself. See Converting the string representation of a value for a detailed explanation of the conversion rules from string to the type.

If the element represents a primitive type array, a class array or an enum array, the string representation of each value is placed in a `value` element placed under the element itself.

If the element represents an annotation, the `annotation` element is used to represent the annotation and placed under `element`. An `annotation` element contains `element` elements.

If the element represents an array of annotations, one or more `annotation` elements are placed under `element`.

Elements with default values in the annotation definition do not have to be represented in XML: the default value will be used in this case. If an XML constraint declaration is missing mandatory elements, or if it contains elements not part of the constraint definition, a `ValidationException` is raised.

Example 9.8: Constraint declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/mapping/validation-mapping-4.0.xsd"
        version="4.0">
    <default-package>com.acme.app.domain</default-package>
    <bean class="Customer" ignore-annotations="false">

        <field name="firstName">

            <!-- @LooksLike(patterns={
                      @Pattern(value="myRegExp", flag=PatternFlag.INSENSITIVE),
                      @Pattern(value="my2ndRegExp")}
                 )
             -->
            <constraint annotation="com.acme.app.constraint.LooksLike">
                <element name="patterns">
```

```
            <annotation>
                <element name="value">myRegExp</element>
                <element name="flag">
                    <value>INSENSITIVE</value>
                </element>
            </annotation>
            <annotation>
                <element name="value">my2ndRegExp</element>
            </annotation>
        </element>
    </constraint>


</field>
<field name="orders">
    <valid/>

    <!-- @DiscreteSize(value={ 0, 20 } )
     -->
    <constraint annotation="com.acme.app.constraint.DiscreteSize">
        <element name="value">
            <value>0</value>
            <value>20</value>
        </element>
    </constraint>


</field>

<!-- Map<@NotBlank String, @Valid PhoneNumber>
 -->
<field name="phoneNumbersByType">
    <container-element-type type-argument-index="0">
        <constraint annotation="jakarta.validation.constraints.NotBlank"/>
    </container-element-type>
    <container-element-type type-argument-index="1">
        <valid/>
    </container-element-type>
</field>

<getter name="orders">
    <valid/>

    <!-- @Size(message="Size is limited",
               groups={Default.class, LightValidation.class},
               max=30
         )
    -->
    <constraint annotation="jakarta.validation.constraints.Size">
        <message>Size is limited</message>
        <groups>
            <value>com.acme.app.model.LightValidation</value>
            <value>jakarta.persistence.Default</value>
        </groups>
        <payload>
            <value>com.acme.app.model.WARN</value>
        </payload>
        <element name="max">30</element>
    </constraint>


</getter>

<constructor ignore-annotations="true">
    <parameter type="java.lang.String">

        <!-- @DiscreteSize(value={ 0, 20 } ) -->
        <constraint annotation="com.acme.app.constraint.DiscreteSize">
            <element name="value">
                <value>0</value>
```

```
                    <value>20</value>
                </element>
            </constraint>
        </parameter>
    </constructor>

    <method name="update" ignore-annotations="true">
        <parameter type="java.lang.String">

            <!-- @DiscreteSize(value={ 0, 20 } ) -->
            <constraint annotation="com.acme.app.constraint.DiscreteSize">
                <element name="value">
                    <value>0</value>
                    <value>20</value>
                </element>
            </constraint>
        </parameter>

        <return-value>

            <!-- @ValidCustomer -->
            <constraint annotation="com.acme.app.constraint.ValidCustomer"/>
        </return-value>
    </method>

    <method name="resetPassword" ignore-annotations="false">
        <parameter type="java.lang.String"/>
        <parameter type="java.lang.String"/>

        <cross-parameter>
            <!-- @ValidResetPasswordParameters -->
            <constraint
                    annotation="com.acme.app.constraint.ValidResetPasswordParameters"/>
        </cross-parameter>
    </method>
</bean>
</constraint-mappings>
```

9.1.1.8. Declaration of group conversions

Group conversion rules are declared by specifying one or more `convert-group` elements within the `field`, `getter`, `parameter`, `return-value` and `container-element-type` elements.

Source and target group of a conversion rule are given by specifying their fully-qualified names within the `from` and `to` attribute respectively. If the `default-package` element is configured for the mapping file, any unqualified class names will be resolved using the given default package.

Example 9.9: Declaration of group conversions

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/mapping/validation-mapping-4.0.xsd"
        version="4.0">
    <default-package>com.acme.app.domain</default-package>
    <bean class="Customer" ignore-annotations="false">

        <field name="firstName">
            <valid/>
            <convert-group from="jakarta.validation.groups.Default"
                    to="com.acme.CustomerBasic"/>
            <convert-group from="com.acmenote.Advanced" to="com.acme.CustomerComplex"/>
        </field>

        <getter name="orders">
            <valid/>
```

```
            <convert-group from="jakarta.validation.groups.Default"
                    to="com.acme.CustomerBasic"/>
        </getter>

        <constructor>
            <parameter type="java.lang.String">
                <valid/>
                <convert-group from="jakarta.validation.groups.Default"
                        to="com.acme.CustomerBasic"/>
            </parameter>
            <return-value>
                <valid/>
                <convert-group from="jakarta.validation.groups.Default"
                        to="com.acme.CustomerBasic"/>
            </return-value>
        </constructor>

        <method name="update">
            <parameter type="java.lang.String">
                <valid/>
                <convert-group from="jakarta.validation.groups.Default"
                        to="com.acme.CustomerBasic"/>
            </parameter>
            <return-value>
                <valid/>
                <convert-group from="jakarta.validation.groups.Default"
                        to="com.acme.CustomerBasic"/>
            </return-value>
        </constructor>
    </bean>
</constraint-mappings>
```

9.1.2. Overriding constraint definitions in XML

A constraint definition (i.e. the annotation representing a constraint), cannot be fully expressed in XML but the list of `ConstraintValidator`s associated to a given constraint can be altered.

A constraint definition is represented by a `constraint-definition` element. The `annotation` attribute represents the constraint annotation being altered. The `validated-by` elements represent the (ordered) list of `ConstraintValidator` implementations associated to the constraint.

If `include-existing-validator` is set to false, `ConstraintValidator` defined on the constraint annotation are ignored. If set to true, the list of `ConstraintValidator`s described in XML are concatenated to the list of `ConstraintValidator` described on the annotation to form a new array of `ConstraintValidator` evaluated. Annotation based `ConstraintValidator`s come before XML based `ConstraintValidator`s in the array. The new list is returned by `ConstraintDescriptor.getConstraintValidatorClasses()`.

Example 9.10: Overriding constraint definitions

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings
    xmlns="https://jakarta.ee/xml/ns/validation/mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        https://jakarta.ee/xml/ns/validation/mapping
        https://jakarta.ee/xml/ns/validation/validation-mapping-4.0.xsd"
    version="4.0">
   <default-package>com.acme.app.domain</default-package>
   <bean class="com.acme.common.model.Address">
      [...]
    </bean>

    <constraint-definition annotation="jakarta.validation.constraints.Size">
        <validated-by include-existing-validators="true">
            <value>com.acme.app.constraint.SizeValidatorForDictionary</value>
        </validated-by>
    </constraint-definition>
    <constraint-definition annotation="AcmeOrderNumber">
```

```
        [...]
    </constraint-definition>
</constraint-mappings>
```

9.1.3. Converting the string representation of a value

Primitive types, `Class` and `Enum` are represented as strings in the XML descriptor. Elements of an array are represented by the `value` element.

A `byte` is represented according to the rules defined in `Byte.parseByte(String)`.

A `short` is represented according to the rules defined in `Short.parseShort(String)`.

An `int` is represented according to the rules defined in `Integer.parseInt(String)`.

A `long` is represented according to the rules defined in `Long.parseLong(String)`.

A `float` is represented according to the rules defined in `Float.parseFloat(String)`.

A `double` is represented according to the rules defined in `Double.parseDouble(String)`.

A `boolean` is represented according to the rules defined in `Boolean.parseBoolean(String)`.

A `char` is represented according to the following rules:

- the string must be of one character long
- the character extracted from the string is the returned `char`

A `Class` is represented by the fully qualified class name of the class or more precisely according to the syntax described in the documentation of `java.lang.Class.getName()`. Note that if the raw string is unqualified, default package is taken into account.

An enum is represented by its `enum.name()` value.

If any of the string representation does not match its type counterpart, a `ValidationException` is raised.

9.1.4. XML Schema

This section contains the XML schema used for constraint mapping descriptors.

From Jakarta Validation revision 1.1 onwards, mapping authors must specify the used version of the schema within the `version` attribute of the `constraint-mappings` element. Implementations supporting Jakarta Validation 2.0 must properly parse mapping descriptors of Jakarta Validation 1.0, 1.1 and 2.0. If the `version` attribute attribute is not given, schema version 1.0 is to be assumed by the Jakarta Validation provider.

In case an unknown version is given (e.g. if a mapping descriptor adhering to a future schema version is parsed by a Jakarta Validation 2.0 provider) a `ValidationException` is raised.

Listing 9.1: XML schema for constraint mapping descriptors

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           targetNamespace="https://jakarta.ee/xml/ns/validation/mapping"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:map="https://jakarta.ee/xml/ns/validation/mapping"
           version="4.0">

    <xs:annotation>
        <xs:documentation><![CDATA[
            This is the XML Schema for Jakarta Validation constraint mapping files.

            Jakarta Validation constraint mapping files must indicate the Jakarta Validation
            XML schema by using the constraint mapping namespace:

            https://jakarta.ee/xml/ns/validation/mapping

            and indicate the version of the schema by using the version attribute
            as shown below:

            <constraint-mappings
```

```
            xmlns="https://jakarta.ee/xml/ns/validation/mapping"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="
                https://jakarta.ee/xml/ns/validation/mapping
                https://jakarta.ee/xml/ns/validation/validation-mapping-4.0.xsd"
            version="4.0">
            ...
        </constraint-mappings>
    ]]>
    </xs:documentation>
</xs:annotation>

<xs:element name="constraint-mappings" type="map:constraint-mappingsType"/>

<xs:complexType name="payloadType">
    <xs:sequence>
        <xs:element type="xs:string" name="value" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="groupsType">
    <xs:sequence>
        <xs:element type="xs:string" name="value" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="groupSequenceType">
    <xs:sequence>
        <xs:element type="xs:string" name="value" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="groupConversionType">
    <xs:attribute type="xs:string" name="from" use="optional"/>
    <xs:attribute type="xs:string" name="to" use="required"/>
</xs:complexType>
<xs:complexType name="constraint-mappingsType">
    <xs:sequence>
        <xs:element type="xs:string" name="default-package" minOccurs="0"/>
        <xs:element type="map:beanType"
                    name="bean"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
        <xs:element type="map:constraint-definitionType"
                    name="constraint-definition"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="version" type="map:versionType" fixed="4.0" use="required"/>
</xs:complexType>
<xs:simpleType name="versionType">
    <xs:restriction base="xs:token">
        <xs:pattern value="[0-9]+(\.[0-9]+)*"/>
    </xs:restriction>
</xs:simpleType>
<xs:complexType name="validated-byType">
    <xs:sequence>
        <xs:element type="xs:string" name="value" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:boolean" name="include-existing-validators" use="optional"/>
</xs:complexType>
<xs:complexType name="constraintType">
    <xs:sequence>
        <xs:element type="xs:string" name="message" minOccurs="0"/>
        <xs:element type="map:groupsType"
                    name="groups"
                    minOccurs="0"/>
        <xs:element type="map:payloadType"
                    name="payload"
                    minOccurs="0"/>
        <xs:element type="map:elementType"
                    name="element"
                    maxOccurs="unbounded"
```

```
                    minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:string" name="annotation" use="required"/>
</xs:complexType>
<xs:complexType name="elementType" mixed="true">
    <xs:sequence>
        <xs:element type="xs:string" name="value" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element type="map:annotationType"
                    name="annotation"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:string" name="name" use="required"/>
</xs:complexType>
<xs:complexType name="containerElementTypeType">
    <xs:sequence>
        <xs:element type="xs:string" name="valid" minOccurs="0" fixed=""/>
        <xs:element type="map:groupConversionType"
                    name="convert-group"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:containerElementTypeType"
                    name="container-element-type"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
        <xs:element type="map:constraintType"
                    name="constraint"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="type-argument-index" use="optional">
        <xs:simpleType>
            <xs:restriction base="xs:int">
                <xs:minInclusive value="0" />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="classType">
    <xs:sequence>
        <xs:element type="map:groupSequenceType"
                    name="group-sequence"
                    minOccurs="0"/>
        <xs:element type="map:constraintType"
                    name="constraint"
                    maxOccurs="unbounded"
                    minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
</xs:complexType>
<xs:complexType name="beanType">
    <xs:sequence>
        <xs:element type="map:classType"
                    name="class"
                    minOccurs="0">
        </xs:element>
        <xs:element type="map:fieldType"
                    name="field"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:getterType"
                    name="getter"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:constructorType"
                    name="constructor"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:methodType"
                    name="method"
```

```
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:string" name="class" use="required"/>
        <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"
                default="true"/>
    </xs:complexType>
    <xs:complexType name="annotationType">
        <xs:sequence>
            <xs:element type="map:elementType"
                        name="element"
                        maxOccurs="unbounded"
                        minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="getterType">
        <xs:sequence>
            <xs:element type="xs:string" name="valid" minOccurs="0" fixed=""/>
            <xs:element type="map:groupConversionType"
                        name="convert-group"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
            <xs:element type="map:containerElementTypeType"
                        name="container-element-type"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
            <xs:element type="map:constraintType"
                        name="constraint"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute type="xs:string" name="name" use="required"/>
        <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
    </xs:complexType>
    <xs:complexType name="methodType">
        <xs:sequence>
            <xs:element type="map:parameterType"
                        name="parameter"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
            <xs:element type="map:crossParameterType"
                        name="cross-parameter"
                        minOccurs="0"
                        maxOccurs="1"/>
            <xs:element type="map:returnValueType"
                        name="return-value"
                        minOccurs="0"
                        maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute type="xs:string" name="name" use="required"/>
        <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
    </xs:complexType>
    <xs:complexType name="constructorType">
        <xs:sequence>
            <xs:element type="map:parameterType"
                        name="parameter"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
            <xs:element type="map:crossParameterType"
                        name="cross-parameter"
                        minOccurs="0"
                        maxOccurs="1"/>
            <xs:element type="map:returnValueType"
                        name="return-value"
                        minOccurs="0"
                        maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
    </xs:complexType>
    <xs:complexType name="parameterType">
```

```
    <xs:sequence>
        <xs:element type="xs:string" name="valid" minOccurs="0" fixed=""/>
        <xs:element type="map:groupConversionType"
                    name="convert-group"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:containerElementTypeType"
                    name="container-element-type"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:constraintType"
                    name="constraint"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute type="xs:string" name="type" use="required"/>
    <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
</xs:complexType>
<xs:complexType name="returnValueType">
    <xs:sequence>
        <xs:element type="xs:string" name="valid" minOccurs="0" fixed=""/>
        <xs:element type="map:groupConversionType"
                    name="convert-group"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:containerElementTypeType"
                    name="container-element-type"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:constraintType"
                    name="constraint"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
</xs:complexType>
<xs:complexType name="crossParameterType">
    <xs:sequence>
        <xs:element type="map:constraintType"
                    name="constraint"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
</xs:complexType>
<xs:complexType name="constraint-definitionType">
    <xs:sequence>
        <xs:element type="map:validated-byType"
                    name="validated-by"/>
    </xs:sequence>
    <xs:attribute type="xs:string" name="annotation" use="required"/>
</xs:complexType>
<xs:complexType name="fieldType">
    <xs:sequence>
        <xs:element type="xs:string" name="valid" minOccurs="0" fixed=""/>
        <xs:element type="map:groupConversionType"
                    name="convert-group"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:containerElementTypeType"
                    name="container-element-type"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element type="map:constraintType"
                    name="constraint"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute type="xs:string" name="name" use="required"/>
    <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
```

```
    </xs:complexType>
</xs:schema>
```

9.2. Configuration schema

XML Configuration is set in `META-INF/validation.xml`. The file is optional. The XML schema followed by the configuration file is as followed.

Listing 9.2: XML schema for XML configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           targetNamespace="https://jakarta.ee/xml/ns/validation/configuration"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:config="https://jakarta.ee/xml/ns/validation/configuration"
           version="4.0">

    <xs:annotation>
        <xs:documentation><![CDATA[
            This is the XML Schema for the Jakarta Validation configuration file.
            The configuration file must be named "META-INF/validation.xml".

            Jakarta Validation configuration files must indicate the Jakarta Validation
            XML schema by using the validation namespace:

            https://jakarta.ee/xml/ns/validation/configuration

            and indicate the version of the schema by using the version attribute
            as shown below:

            <validation-config
                xmlns="https://jakarta.ee/xml/ns/validation/configuration"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="
                    https://jakarta.ee/xml/ns/validation/configuration
                    https://jakarta.ee/xml/ns/validation/validation-configuration-4.0.xsd"
                version="4.0">
                [...]
            </validation-config>
        ]]>
        </xs:documentation>
    </xs:annotation>

    <xs:element name="validation-config" type="config:validation-configType"/>
    <xs:complexType name="validation-configType">
        <xs:sequence>
            <xs:element type="xs:string" name="default-provider" minOccurs="0"/>
            <xs:element type="xs:string" name="message-interpolator" minOccurs="0"/>
            <xs:element type="xs:string" name="traversable-resolver" minOccurs="0"/>
            <xs:element type="xs:string" name="constraint-validator-factory" minOccurs="0"/>
            <xs:element type="xs:string" name="parameter-name-provider" minOccurs="0"/>
            <xs:element type="xs:string" name="clock-provider" minOccurs="0"/>
            <xs:element type="xs:string" name="value-extractor" maxOccurs="unbounded"
                    minOccurs="0"/>
            <xs:element type="config:executable-validationType" name="executable-validation"
                    minOccurs="0"/>
            <xs:element type="xs:string" name="constraint-mapping" maxOccurs="unbounded"
                    minOccurs="0"/>
            <xs:element type="config:propertyType" name="property" maxOccurs="unbounded"
                    minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="version" type="config:versionType" fixed="4.0" use="required"/>
    </xs:complexType>

    <xs:complexType name="executable-validationType">
        <xs:sequence>
            <xs:element type="config:default-validated-executable-typesType"
```

```
                    name="default-validated-executable-types" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="enabled" use="optional" type="xs:boolean" default="true"/>
    </xs:complexType>
    <xs:complexType name="default-validated-executable-typesType">
        <xs:sequence>
            <xs:element name="executable-type" maxOccurs="unbounded" minOccurs="1">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="NONE"/>
                        <xs:enumeration value="CONSTRUCTORS"/>
                        <xs:enumeration value="NON_GETTER_METHODS"/>
                        <xs:enumeration value="GETTER_METHODS"/>
                        <xs:enumeration value="ALL"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="propertyType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="name" use="required" type="xs:string"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:simpleType name="versionType">
        <xs:restriction base="xs:token">
            <xs:pattern value="[0-9]+(\.[0-9]+)*" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

From Jakarta Validation revision 1.1 onwards, the used version of the schema must be specified within the `version` attribute of the `validation-config` element. Implementations supporting Jakarta Validation 2.0 must properly parse configuration descriptors of Jakarta Validation 1.0, 1.1 and 2.0. If the `version` attribute attribute is not given, schema version 1.0 is to be assumed by the Jakarta Validation Provider.

In case an unknown version is given a `ValidationException` is raised.

See XML configuration: META-INF/validation.xml for more information on XML based configuration.

10. Exception model

Illegal arguments passed to the Jakarta Validation APIs generally lead to an `IllegalArgumentException` (see the JavaDoc for specific details). Other exceptions raised by Jakarta Validation are or inherit from the runtime exception `jakarta.validation.ValidationException`. Exception cases are described in their respective sections but include (non exhaustive list):

- invalid constraint definitions (missing mandatory elements, illegal composition cycle, illegal attribute overriding, etc.)
- invalid constraint declarations (`ConstraintValidator` implementation matching failure, etc.)
- invalid group definition (circularity)
- invalid `Default` group redefinition for classes (missing class group etc)
- invalid group conversion definitions
- error when retrieving, initializing, executing `ConstraintValidator`s
- error when parsing the XML configuration or mappings
- multiple XML configuration files found
- missing expected provider or no default provider found
- missing no-arg constructor on extension implementations described in XML configuration files
- same entity described more than once across the XML mapping files
- same property or field described more than once for a given entity in the XML mapping files
- class, field or getter declared in XML mapping files but not found
- illegal XML constraint definition
- illegal XML constraint declaration
- exception raised either at initialization time or execution time by any of the extension interfaces

Each of these error cases lead to a `ValidationException` or a subclass of `ValidationException` (see following subsections).

Every (runtime) exception raised either at initialization time or execution time by any of the extension interfaces (`ConstraintValidator`, `ConstraintValidatorFactory`, `MessageInterpolator`, `TraversableResolver`, `ValidationProviderResolver`, `ParameterNameProvider`, `ClockProvider`, `ValueExtractor`) is wrapped in a `ValidationException`.

If a constraint definition or constraint declaration is invalid for a given class, the metadata API should raise the according exception.

10.1. Error report: `ConstraintViolationException`

Some frameworks or applications need to convey the result of a validation by raising an exception if the validation returns constraint violations.

Jakarta Validation provides a reference exception for such cases. Frameworks and applications are encouraged to use `ConstraintViolationException` as opposed to a custom exception to increase consistency of the Java platform. The exception can be raised directly or wrapped into the framework or application specific parent exception.

Example 10.1: ConstraintViolationException

```
/**
 * Reports the result of constraint violations.
 *
 * @author Emmanuel Bernard
 * @author Gunnar Morling
 */
public class ConstraintViolationException extends ValidationException {
    [...]

    /**
     * Creates a constraint violation report.
     *
     * @param message error message
     * @param constraintViolations {@code Set} of {@link ConstraintViolation}
     */
    public ConstraintViolationException(String message,
                                        Set<? extends ConstraintViolation<?>> constraintViolations) {
        [...]
    }

    /**
     * Creates a constraint violation report.
     *
```

```
         * @param constraintViolations {@code Set} of {@link ConstraintViolation}
         */
        public ConstraintViolationException(Set<? extends ConstraintViolation<?>> constraintViolations) {
            [...]
        }

        /**
         * Set of constraint violations reported during a validation.
         *
         * @return {@code Set} of {@link ConstraintViolation}
         */
        public Set<ConstraintViolation<?>> getConstraintViolations() {
            [...]
        }
    }
```

The `ConstraintViolationException` carries a `Set` of `ConstraintViolation`.

NOTE Jakarta Validation never raises this exception itself. Other frameworks like Jakarta Persistence 2 or interception framework wiring method validation do.

NOTE If this exception is meant to be sent remotely, `ConstraintViolation` objects should be `Serializable` as defined and explained in ConstraintViolation.

10.2. Constraint definition: `ConstraintDefinitionException`

If a constraint definition does not respect the Jakarta Validation rules or is inconsistent, a `ConstraintDefinitionException` is raised. `ConstraintDefinitionException` is a subclass of `ValidationException`.

This exception can be raised during validation or when the metadata model for the class hosting this constraint is requested.

NOTE These exception cases can be determined at compile time by a tool such as an annotation processor.

10.3. Constraint declaration: `ConstraintDeclarationException` and `UnexpectedTypeException`

When a constraint declaration is illegal, `ConstraintDeclarationException` is raised. Reasons include:

- incorrect group conversion rules (definition or positioning)
- no suitable value extractor could be unambiguously identified for container element validation or cascaded validation of a container
- illegal method constraint declarations (e.g. inheritance rules, cross-parameter constraint used in an illegal situation, improper use of `validationAppliesTo`).

`ConstraintDeclarationException` is a subclass of `ValidationException`.

When the return type of a property cannot be processed for a given constraint, an `UnexpectedTypeException` is raised. This problem typically arises when either no `ConstraintValidator`s or too many `ConstraintValidator`s match the return type (see ConstraintValidator resolution algorithm).

`UnexpectedTypeException` is a subclass of `ConstraintDeclarationException`.

This exception can be raised during validation or when the metadata model for the class hosting this constraint is requested.

NOTE These exception cases can be determined at compile time by a tool such as an annotation processor.

10.4. Group definition: `GroupDefinitionException`

When a group definition is illegal, a `GroupDefinitionException` is raised. This typically arises when a cyclic group dependency is discovered, an illegal attribute overriding is defined etc.

`GroupDefinitionException` is a subclass of `ValidationException`.

NOTE These exception cases can be determined at compile time by a tool such as an annotation processor.

10.5. Value extractor definition: `ValueExtractorDefinitionException`

When detecting an illegal value extractor definition, a `ValueExtractorDefinitionException` will be raised.

Reasons for raising this exception include:

- The extracted type is not marked with `@ExtractedValue`
- The `@ExtractedValue` annotation is given more than once for one value extractor type

`ValueExtractorDefinitionException` is a subclass of `ValidationException`.

10.6. Value extractor declaration: `ValueExtractorDeclarationException`

When detecting an illegal configuration of value extractors, a `ValueExtractorDeclarationException` will be raised. One example is the configuration of multiple extractors for the same container element type in `META-INF/validation.xml`.

`ValueExtractorDeclarationException` is a subclass of `ValidationException`.

10.7. No Jakarta Validation Provider detected: `NoProviderFoundException`

When trying to bootstrap Jakarta Validation via `Validation.buildDefaultValidatorFactory()` or `Validation.byDefaultProvider().configure()` and no Jakarta Validation provider could be found, a `NoProviderFoundException` is raised.

`NoProviderFoundException` is a subclass of `ValidationException`.

11. Integration

In this chapter, integration points between Jakarta Validation and other technologies are discussed. We first address the integration in generic terms applying to all integrations and we then detail how integration with various Jakarta EE specifications is handled more specifically.

11.1. General requirements

This section covers general requirements that should be followed by any container and interception technology integrating Jakarta Validation.

11.1.1. Objects lifecycle

Generally speaking, containers and frameworks controlling the lifecycle of objects (such as Jakarta EE, dependency injection frameworks or component frameworks) should:

- build and bootstrap the `ValidatorFactory` instance for an application.
- provide access to the `ValidatorFactory` instance as well as `Validator` instances in their default configuration using the paradigm of the container: for example, such instances would be injectable in other objects via a dependency injection framework.
- configure `ValidatorFactory` with a custom `ConstraintValidatorFactory` instance that returns managed `ConstraintValidator` instances, unless a custom `ConstraintValidatorFactory` is requested by the user. The scope of `ConstraintValidator` instances is still fully controlled by the Jakarta Validation provider as described in The ConstraintValidatorFactory, but as managed beans they can receive expected services like injection of other objects.
- configure `ValidatorFactory` with managed instances of `ConstraintValidatorFactory`, `MessageInterpolator`, `ParameterNameProvider`, `ClockProvider` and `TraversableResolver`, if such instances are defined in the XML deployment descriptor. Services provided by the container (like dependency injection) should thus be available to these instances.
- invoke `ValidatorFactory.close()` when the `ValidatorFactory` instance is no longer needed.
- dispose of managed instances provided to the Jakarta Validation bootstrap process after `ValidatorFactory.close()` has been invoked.

IMPORTANT

In this context, a default `ValidatorFactory` is a factory configured like the factory returned by `Validation.buildDefaultValidatorFactory` (see also Validation) except for the enhancements described above. A default `Validator` instance is a `Validator` instance retrieved via `getValidator()` from the default `ValidatorFactory`.

11.1.2. Method and constructor validation

This section expresses the behavior that integration with interception frameworks should follow. Any deviation should be considered with care as it will surprise Jakarta Validation users.

Method interception frameworks (such as AOP or interceptor frameworks) enable interception of constrained methods following the steps defined in Triggering method validation. Method validation execution is implicit for any method or constructor annotated with constraints.

By default, method validation is applied to all constrained methods or constructors provided the integration technology can intercept the call. By default, getters (as defined in Requirements on classes to be validated) are not considered constrained methods. Static methods are ignored by validation. Putting constraints on a static method is not portable.

Jakarta Validation - via the interception technology - offers a way to customize whether or not a constructor, method or getter is validated when executed. This is achieved:

- via the `@ValidateOnExecution` annotation on the executable (see `@ValidateOnExecution` annotation)
- via the `@ValidateOnExecution` annotation on the type declaring the executable
- via a global configuration defined in `validation.xml`: `executable-validation` and `default-validated-executable-types`. See XML configuration: META-INF/validation.xml for more details.

NOTE

Integration layers can read the list of validated executable types defined in the global configuration as well as read whether or not executable validation is disabled via the `Configuration` object: `configuration.getBootstrapConfiguration().getDefaultValidatedExecutableTypes()` and `configuration.getBootstrapConfiguration().isExecutableValidationEnabled()` respectively. This list is extracted from `validation.xml`.

More formally, a given executable (constructor or method) is validated upon execution according to the following rules in decreasing order:

- the executable is validated if it is annotated with `@ValidateOnExecution` and the `type` attribute contains the executable type or `IMPLICIT`. If the `type` attribute does neither contain the executable type nor `IMPLICIT`, the executable is not validated.
- otherwise the executable is validated if the type (class, interface) on which it is declared is annotated with `@ValidateOnExecution` and

the `type` attribute contains the executable type. If the `type` attribute contains `IMPLICIT`, then this rule is ignored and the behavior is equivalent to `@ValidateOnExecution` not being present. If the `type` attribute does not contain the executable type, the executable is not validated.

- otherwise the executable is validated if the global executable validation setting contains the executable type. If the global setting does not contain the executable type, the executable is not validated.
- The rules above do not apply to methods overriding a superclass method or implementing an interface method. In this case, the method inherits the behavior of the method it overrides / implements. Out of the box, a conforming integration implementation raises a `ValidationException` if the overriding / implementing method hosts the `@ValidateOnExecution` annotation.

The last point is present to enforce the Liskov substitution principle (more info at Method constraints in inheritance hierarchies). In addition, providers may implement alternative, potentially more liberal, approaches for handling validated methods in inheritance hierarchies. Possible means for activating such alternative behavior include provider-specific configuration properties or annotations. Note that client code relying on such alternative behavior is not portable.

The following executable types are available:

- `NONE`: parameters and return values are not validated upon execution. This option is equivalent to an empty list of executable types and is present to improve readability. A list containing `NONE` and other types of executables is equivalent to a list containing the types of executables without `NONE`.
- `CONSTRUCTORS`: parameters and return values are validated provided the executable is a constructor.
- `NON_GETTER_METHODS`: parameters and return values are validated provided the executable is a method but not a getter.
- `GETTER_METHODS`: return values are validated provided the executable is a getter method.
- `ALL`: parameters and return values are validated for all executables (getters, non getters and constructors). This option is equivalent to a list of all executable types and is present to improve readability. A list containing `ALL` and other types of executables is equivalent to a list containing only `ALL`.
- `IMPLICIT`: if `@ValidateOnExecution` is on a type (class or interface), then it is equivalent to `@ValidateOnExecution` not being present; if `@ValidateOnExecution` is on a an executable, the following applies:
  - if on a constructor, it is equivalent to `CONSTRUCTORS`.
  - if on a non-getter method, it is equivalent to `NON_GETTER_METHODS`.
  - if on a getter, it is equivalent to `GETTER_METHODS`.

Mixing `IMPLICIT` and other executable types is illegal.

Listing 11.1: `@ValidateOnExecution` *annotation*

```
package jakarta.validation.executable;


/**
 * Expresses which executables (methods or constructors) should have their parameters
 * and return value validated upon execution. Can be on executable (method, constructor)
 * or type level (with the former taking precedence).
 * <p>
 * If not present for a given executable, the default configuration from
 * {@code META-INF/validation.xml} and finally the implicit default
 * validated executable types (constructors and non-getters) are taken into account to
 * determine whether a given executable is validated upon execution or not.
 * <p>
 * The following describes the formal rules for deciding whether an executable is validated.
 * They are applied in decreasing order:
 * <ul>
 *     <li>the executable is validated if it is annotated with {@code @ValidateOnExecution}
 *     and the {@code type} attribute contains the executable type or
 *     {@link ExecutableType#IMPLICIT}. If the {@code type} attribute does neither contain
 *     the executable type nor {@code IMPLICIT}, the executable is not validated.</li>
 *     <li>otherwise the executable is validated if the type (class, interface) on which it
 *     is declared is annotated with {@code @ValidateOnExecution} and the {@code type}
 *     attribute contains the executable type. If the {@code type} attribute contains
 *     {@code IMPLICIT}, then this rule is ignored and the behavior is equivalent to
 *     {@code ValidateOnExecution} not being present. If the {@code type} attribute does not
 *     contain the executable type, the executable is not validated.</li>
 *     <li>otherwise the executable is validated if the global executable validation setting
 *     contains the executable type. If the global setting does not contain the executable
 *     type, the executable is not validated.</li>
 *     <li>The rules above do not apply to methods overriding a superclass method or
 *     implementing an interface method. In this case, the method inherits the behavior
 *     of the method it overrides or implements. Out of the box, a conforming implementation
```

```
 *     raises a {@link ValidationException} if the overriding / implementing method hosts
 *     the {@code ValidateOnExecution} annotation.</li>
 * </ul>
 * <p>
 * Note that you can exclude an executable from validation by making sure the rules above do
 * not match or by annotating the executable with {@code @ValidateOnExecution(NONE)}.
 *
 * @author Emmanuel Bernard
 * @since 1.1
 */
@Target({ CONSTRUCTOR, METHOD, TYPE, PACKAGE })
@Retention(RUNTIME)
@Documented
public @interface ValidateOnExecution {

    /**
     * List of executable types to be validated when called.
     * Defaults to the types discovered implicitly (see {@link ExecutableType#IMPLICIT}).
     *
     * @return array of {@code ExecutableType}s to be validated
     */
    ExecutableType[] type() default {ExecutableType.IMPLICIT};
}


package jakarta.validation.executable;

/**
 * Defines the types of executables targeted by an operation.
 *
 * @author Emmanuel Bernard
 * @since 1.1
 */
public enum ExecutableType {

    /**
     * If the annotation using {@code ExecutableType} is on a type (class or interface),
     * the behavior is equivalent to the annotation not being present.
     * <p>
     * If on a constructor, it is equivalent to {@link #CONSTRUCTORS}.
     * <p>
     * If on a non-getter method, it is equivalent to {@link #NON_GETTER_METHODS}.
     * <p>
     * If on a getter method, it is equivalent to {@link #GETTER_METHODS}.
     */
    IMPLICIT,

    /**
     * None of the executables.
     * <p>
     * Note that this option is equivalent to an empty list of executable types
     * and is present to improve readability. If {@code NONE} and other types of executables
     * are present in a list, {@code NONE} is ignored.
     */
    NONE,

    /**
     * All constructors.
     */
    CONSTRUCTORS,

    /**
     * All methods except the ones following the getter pattern. A getter according to the
     * JavaBeans specification is a method whose:
     * <ul>
     *     <li>name starts with get, has a return type but no parameter</li>
     *     <li>name starts with is, has a return type and is returning {@code boolean}.</li>
     * </ul>
     */
    NON_GETTER_METHODS,
```

```
        /**
         * All methods following the getter pattern. A getter according to the
         * JavaBeans specification is a method whose:
         * <ul>
         *     <li>name starts with get, has a return type but no parameter</li>
         *     <li>name starts with is, has a return type and is returning {@code boolean}.</li>
         * </ul>
         */
        GETTER_METHODS,

        /**
         * All executables (constructors and methods).
         */
        ALL
    }
```

If a sub type overrides/implements a method originally defined in several parallel types of the hierarchy (e.g. two interfaces not extending each other, or a class and an interface not implemented by said class), `@ValidateOnExecution` cannot be placed in the parallel types of the hierarchy. This is to avoid an unexpected altering of the post conditions to be guaranteed to the caller.

You can globally disable executable validation by using `<executable-validation enabled="false"/>`, in this case, `<default-validated-executable-types/>` and `@ValidateOnExecution` are ignored.

Example 11.1: validation.xml disabling executable validation

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
    xmlns="https://jakarta.ee/xml/ns/validation/configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        https://jakarta.ee/xml/ns/validation/configuration
        https://jakarta.ee/xml/ns/validation/validation-configuration-3.0.xsd"
    version="3.0">
    <default-provider>com.acme.ACMEProvider</default-provider>
    <message-interpolator>com.acme.ACMEAwareMessageInterpolator</message-interpolator>

    <executable-validation enabled="false"/>

    <constraint-mapping>META-INF/validation/order-constraints.xml</constraint-mapping>
    <constraint-mapping>META-INF/validation/catalog-constraints.xml</constraint-mapping>
    <constraint-mapping>META-INF/validation/customer-constraints.xml</constraint-mapping>

    <property name="com.acme.validation.logging">WARN</property>
    <property name="com.acme.validation.safetyChecking">failOnError</property>

</validation-config>
```

`@ValidateOnExecution(type=IMPLICIT)` on a type (class or interface) is useful to mark a class as being involved in executable validation without affecting the behavior. This is used when the integration technology needs a little help to find the classes and interfaces involved.

> **NOTE** The proper selection of the validated executables is the responsibility of the integration between the interception technology and Jakarta Validation. Jakarta Validation engines ignore the XML configuration around executable validation and `@ValidateOnExecution` when validating executables and when providing metadata.

11.1.2.1. Examples

The following example shows some of the way you can refine executable validation with `@ValidateOnExecution`.

Example 11.2: Method validation configurations

```
//optional: @ValidateOnExecution
public class OrderService {
```

```
    boolean isValidCustomer(@NotNull String customerCode) { [...] }

    @ValidateOnExecution
    @Min(0)
    Integer getBacklog() { [...] }

    @ValidateOnExecution(type=NONE)
    Order placeOrder(@NotNull String customerCode, @Valid Item item, int quantity) { [...] }

}

@ValidateOnExecution(type={GETTER_METHODS, NON_GETTER_METHODS})
public class SimpleOrderService extends OrderService {

    public SimpleOrderService(@NotNull ServiceProvider provider) { [...] }

    @Overrides
    Order placeOrder(String customerCode, Item item, int quantity) { [...] }

}

//optional: @ValidateOnExecution
public class ComplexOrderService extends SimpleOrderService {
    public ComplexOrderService(@NotNull ServiceProvider provider) { [...] }
}
```

All constructors and non-getter methods of `OrderService` are validated upon execution as this is the default setting. `isValidCustomer()` is validated as this method is not a getter (it has a parameter). `getBacklog()` is a getter but is validated thanks to `@ValidateOnExecution` defaulting to `GETTER_METHODS`. `placeOrder()` is not validated as `@ValidateOnExecution` is set to `NONE`.

All getter and non-getter methods of `SimpleOrderService` are validated upon execution by default due to the presence of `@ValidateOnExecution` on the class. The `SimpleOrderService` constructor is thus not validated. `SimpleOrderService.placeOrder()` is not validated either because it overrides `OrderService.placeOrder()` and thus inherits its settings.

All constructors and non-getter methods of `ComplexOrderService` are validated upon execution as this is the default setting - the type level settings of `SimpleOrderService` are not inherited. This means that the `ComplexOrderService` constructor is validated.

`@ValidateOnExecution` can be optionally set on `OrderService` and `ComplexOrderService` without altering the semantic. This marker is necessary for some integration technology in some situations.

11.2. Jakarta EE

Jakarta EE must obey the rules defined above and make the following instances available under JNDI:

- `ValidatorFactory` under `java:comp/ValidatorFactory`
- `Validator` under `java:comp/Validator`

Instead of looking the instances up via JNDI, the user can request them to be injected via the `Resource` annotation:

```
@Resource ValidatorFactory validatorFactory;
@Resource Validator validator;
```

When the application is Jakarta Context and Dependency Injection enabled, the `ValidatorFactory` and `Validator` instances returned by JNDI or `@Resource` injection are Jakarta Context and Dependency Injection enhanced as defined in Jakarta Context and Dependency Injection integration. In particular, dependency injection is available to Jakarta Validation components.

11.3. Jakarta Context and Dependency Injection integration

There are several integrations points between Jakarta Validation and Jakarta Context and Dependency Injection. If a Jakarta Validation provider integrates with Jakarta Context and Dependency Injection, it must follow the rules laid out in this section. In a Jakarta EE container, a Jakarta Validation provider must integrate with Jakarta Context and Dependency Injection.

11.3.1. `ValidatorFactory` and `Validator`

Similar to the Jakarta EE integration via `@Resource` (see Jakarta EE), a Jakarta Context and Dependency Injection container must support injection of built-in default `ValidatorFactory` and `Validator` beans via `@Inject`. These default beans are injectable via the `@Default`

qualifier.

```
@Inject ValidatorFactory;
@Inject Validator;
```

Optionally, the Jakarta Context and Dependency Injection container can support injection of provider specific - as defined by `Validation.byProvider()` - `ValidatorFactory` and `Validator` beans via `@Inject`. These beans must be registered with a custom qualifier, for example `@ACME`. Using the product name or brand for the qualifier is considered good practice.

```
@Inject @ACME ValidatorFactory;
@Inject @ACME Validator;
```

NOTE

Discussion on possible implementations

Registration of the built-in default beans and the provider specific beans may be achieved using the Jakarta Context and Dependency Injection portable extension SPI or a vendor specific SPI.

11.3.2. `ConstraintValidatorFactory`, `MessageInterpolator`, `ParameterNameProvider`, `ClockProvider`, `TraversableResolver` and `ValueExtractor`

If custom `ConstraintValidatorFactory`, `MessageInterpolator`, `ParameterNameProvider`, `ClockProvider`, `TraversableResolver` or `ValueExtractor` classes are defined in the XML deployment descriptor (see XML configuration: META-INF/validation.xml), the `ValidatorFactory` must be configured with Jakarta Context and Dependency Injection managed beans representing the requested classes. Services like dependency injection, interception and decoration must thus be made available to these instances by the container. The same applies to value extractors discovered through the service loader mechanism (see Registering ValueExtractor implementations).

If no custom `ConstraintValidatorFactory` is requested by the user, the `ValidatorFactory` must be configured with a custom `ConstraintValidatorFactory` instance that returns Jakarta Context and Dependency Injection managed beans representing the requested `ConstraintValidator` types. The factory

- creates non-contextual `ConstraintValidator` instances for each `ConstraintValidatorFactory.getInstance()` call. To inject dependencies into the `ConstraintValidator` instance, the Jakarta Context and Dependency Injection `InjectionTarget` API should be used. Before returning the instance the following calls should be made: `InjectionTarget.produce()`, `InjectionTarget.inject()` and `InjectionTarget.postConstruct()`.
- calls `InjectionTarget.preDestroy()` and `InjectionTarget.dispose()` upon `ConstraintValidatorFactory.releaseInstance` (see also The ConstraintValidatorFactory for more information about the lifecycle of a `ConstraintValidator`).

Using directly or indirectly a Jakarta Persistence `EntityManager` that might call back Jakarta Validation for validation is not allowed in the Jakarta Validation extension points and in `ConstraintValidator` instances. This would lead to infinite flush or unexpected behavior.

11.3.3. Method and constructor validation

Jakarta Validation requires that Jakarta Context and Dependency Injection beans support constructor and method validation as defined in Method and constructor validation. Validation must happen at the equivalent time an interceptor occurs when having priority `Interceptor.Priority.PLATFORM_AFTER+800`, in other words priority of `4800`.

For maximum portability, it is recommended to mark Jakarta Context and Dependency Injection bean interfaces and classes involved in executable validation with `@ValidateOnExecution` (defaults to `IMPLICIT`). This helps some implementations to bind the method validation interceptor. Most Jakarta Context and Dependency Injection - Jakarta Validation integration implementations do not need such marker. In particular this marker should not be needed on validated beans annotated with constraint annotations, `@Valid` or `@ValidateOnExecution` anywhere in the class. Such limitation will be removed in a future version of this specification.

NOTE

Discussion on possible implementations

The Jakarta Context and Dependency Injection interceptor binding facility does not directly support this, but the effect may be achieved using the Jakarta Context and Dependency Injection portable extension SPI, or vendor specific SPIs. For example, an interceptor with the expected priority can be programmatically bound to the constructors and methods expected to be validated according to the rules at Method and constructor validation.

It is recommended to only intercept methods and constructors that are both constrained and validated according to the rules defined at Method and constructor validation. Triggering method validation gives examples how the metadata API can be used to determine whether or not a method is constrained (regardless of the filtering rules of `@ValidateOnExecution`).

11.3.4. Java records

Java records are a new language feature introduced in Java 14. They are classes that are designed to be immutable and have a concise syntax for declaring properties and methods. They are unproxyable bean types as per Jakarta Context and Dependency Injection since they are final and have no non-private constructor with no parameters. Because of this, Jakarta Context and Dependency Injection beans that are records are not validated by default. To enable validation of records, one would need to create a producer method that performs the validation, or utilize some bytecode enhancement to intercept the constructor call and perform the validation. There are no requirements for Jakarta Validation providers to support validation of records automatically in Jakarta Context and Dependency Injection environments.

11.4. Jakarta Persistence 2.0 integration

Integration with Jakarta Persistence is described in the Jakarta Persistence 2 specification. Persistence frameworks are encouraged to mimic the integration work done with Jakarta Persistence.

11.5. Jakarta Server Faces 2.0 integration

Integration with Jakarta Server Faces is described in the Jakarta Server Faces 2 specification. Presentation frameworks are encouraged to study the integration work done with Jakarta Server Faces 2.

11.6. Jakarta RESTful Web Services 2 integration

Integration with Jakarta RESTful Web Services is described in the Jakarta RESTful Web Services 2 specification.

Appendix A: Terminology

This appendix aims at giving an overview on the different key terms used through this specification. They are not to be considered formal definitions. Formal definitions are to be inferred from the core specification.

Constraint

A restriction on a bean instance, the value of a field or the value of a JavaBean property.

Constraint declaration

Assignment of a constraint to a target (bean, field, property) for a specific class. Typically by declaring an annotation on the target but can also be done through a XML deployment descriptor.

Validation routine

Sequence of operations executed by the Jakarta Validation provider to validate a given object graph.

Constraint definition

Defines a type of constraint, its attributes and the actual constraint validation implementations. Done through annotations. The list of constraint validation implementations can be provided via XML.

Group

Constraints can belong to one or more group or context. Useful to apply a subset of the constraints for a given use case. By default, the `Default` group is used.

Group Sequence

Defines a group ordering in the validation process. If a given group in the sequence contains one or more failure, the following groups in the sequence must be ignored.

Constraint validation

Constraint logic algorithm used to determine whether a given value passes a constraint or not.

Constraint validation implementation

Class implementing the constraint logic and used to determine whether a given value passes a constraint or not.

Jakarta Validation provider

Product implementing this specification.

Message interpolator

Algorithm used to build the end user message associated to a constraint failure. Typically useful for i18n.

Constraint metadata API

API exposing the constraints applied to a given bean type. Also considered one of the integration points with other specification or frameworks.

Bootstrap API

Bootstrapping part of the Jakarta Validation API producing a `ValidatorFactory`.

jakarta.validation.ConstraintValidator

Interface implemented by a constraint validation implementation.

Composing constraint

Constraint declared on another constraint definition. When the main constraint is validated, the composing constraints are validated too.

jakarta.validation.Validator

Main API. Holds contracts to validate object graphs.

jakarta.validation.ConstraintViolation

Interface describing a given constraint failure on a given bean.

Getter

Method whose:

- name starts with `get` and has a return type but no parameter
- name starts with `is`, has no parameter and is returning `boolean`

Generic type

A class or interface with one or more type parameters, e.g. `class List<E> { … }`

Parameterized type (instantiated type)

A type created from a generic type by providing an actual type argument per formal type parameter, e.g. `List<String>`

Type argument

A reference type or a wildcard that is used for instantiation / invocation of a generic type or a reference type used for instantiation / invocation of a generic method. In the following example, `?` and `String` are two type arguments:

```
List<?> list = new LinkedList<String>();
```

Type parameter

A place holder for a type argument. Each type parameter is replaced by a type argument when a generic type or generic method is instantiated / invoked. In the following example, `E` is a type parameter:

```
interface Comparable<E> {
}
```

Container type

A type of object containing one or more elements, e.g. `List`, `Map` or `Optional`

Container element type

A type of object contained in a container, e.g. the type of the elements of a `List`, the type of the keys and values of a `Map` or the type contained in an `Optional`

Appendix B: Standard ResourceBundle messages

The properties listed below are resolved by the default message interpolator.

```
jakarta.validation.constraints.AssertFalse.message=must be false
jakarta.validation.constraints.AssertTrue.message=must be true
jakarta.validation.constraints.DecimalMax.message=\
    must be less than ${inclusive == true ? 'or equal to ' : ''}{value}
jakarta.validation.constraints.DecimalMin.message=\
    must be greater than ${inclusive == true ? 'or equal to ' : ''}{value}
jakarta.validation.constraints.Digits.message=\
    numeric value out of bounds (<{integer} digits>.<{fraction} digits> expected)
jakarta.validation.constraints.Email.message=must be a well-formed email address
jakarta.validation.constraints.Future.message=must be a future date
jakarta.validation.constraints.FutureOrPresent.message=must be a date in the present or in the future
jakarta.validation.constraints.Max.message=must be less than or equal to {value}
jakarta.validation.constraints.Min.message=must be greater than or equal to {value}
jakarta.validation.constraints.Negative.message=\
    must be less than 0
jakarta.validation.constraints.NegativeOrZero.message=\
    must be less than or equal to 0
jakarta.validation.constraints.NotBlank.message=must not be blank
jakarta.validation.constraints.NotEmpty.message=must not be empty
jakarta.validation.constraints.NotNull.message=must not be null
jakarta.validation.constraints.Null.message=must be null
jakarta.validation.constraints.Past.message=must be a past date
jakarta.validation.constraints.PastOrPresent.message=must be a date in the past or in the present
jakarta.validation.constraints.Pattern.message=\
    must match the following regular expression: {regexp}
jakarta.validation.constraints.Positive.message=\
    must be greater than 0
jakarta.validation.constraints.PositiveOrZero.message=\
    must be greater than or equal to 0
jakarta.validation.constraints.Size.message=size must be between {min} and {max}
```

Appendix C: Jakarta Persistence 2.0 and schema generation

While not specified by this specification or the Jakarta Persistence 2.0 specification, Persistence Providers are encouraged to make use of Jakarta Validation constraint metadata when generating DDL schemas. The proposal is as followed.

```
Ideas explored and not standardized

Jakarta Persistence consumes Jakarta Validation metadata to enhance persistence property
metadata.

A Persistence provider must use the Jakarta Validation metadata of a given list of groups.
The default group evaluated is Default (default Jakarta Validation group). Groups evaluated
can be overridden by a property.
This property contains the comma separated groups (fully qualified class name).

For each entity, apply the following algorithm.
For each persistent property in a given entity:
 - extract the list of Jakarta Validation constraints (including the composing constraints)
 - determine the subset of applicable constraints
    (i.e. constraints understood by the persistence provider)
 - apply these constraints on the persistent property metadata
 - if the property type is an embeddable object or a collection
of embeddable objects, apply the algorithm on the embeddable object properties.

The list of constraints that must be understood by persistence providers are
as followed:
 - @NotNull should be considered equivalent to @Column(nullable=false) /
     @JoinColumn(nullable=false)
 - @Size.max should be considered equivalent to @Column.length
     for String properties
 - @Digits (which contains integer and fraction) should be considered
     equivalent to @Column.precision = integer+fraction,
     @Column.scale = fraction for decimal columns

The Jakarta Validation annotation metadata should have priority over Jakarta Persistence metadata
(Jakarta Persistence has no sensible "unset" values on their annotations).

Question: should we add @Unique that would map to @Column(unique=true)?
@Unique cannot be tested at the Java level reliably but could generate
a database unique constraint generation. @Unique is not part
of the Jakarta Validation spec today.

Persistence Provider should optionally recognize and try to apply the
following constraints as well:
 - @Min / @Max on numeric columns (TODO String too?)
 - @Future / @Past on temporal columns
 - @Size for collections and array (not sure it is feasible).

Persistence Providers can also apply non standard constraints to their metadata model.
For example, provider ACME might recognize and understand @com.acme.validation.Email
and apply it to the database model.

While most high level constraints will not be recognized, the Jakarta Validation built-in constraints
will be the common language spoken by Persistence Providers. Any high level constraint
can be composed of more modular constraints (constraint composition).

* additional proposal
In case of a constraint violation report detected and generated by the database
(not null, etc), the Java persistence provider catches this report and translates
it into a Jakarta Validation error report. From the perspective of the application, constraint
errors are viewed through a unified layer. Jakarta Validation must provide some API to create a
constraint violation error (constraintDescriptor.createConstraintViolation(…)).

While this proposal has a lot of value-add, I wonder how difficult it can be to
implement this in persistence providers.

Provide a way to disable Jakarta Validation metadata use by a persistence provider (property based).
```

This is not an endorsement of the Jakarta Persistence expert group or the Jakarta Validation expert group. Such approach may or may not

be standardized in the future. Such integration should not be considered portable.

Appendix D: Module name

This specification defines `jakarta.validation` as the module name for the Jakarta Validation API in terms of the Java Platform Module System (as defined by JSR 376).

Appendix E: Changelog

NOTE Names used under the JCP for specifications are preserved in the Changelog section for versions released prior to the move to Jakarta EE in order to preserve historical accuracy.

```
4.0.0-M1 (2025-10-22)
------------------
** New Feature
    * [jakartaee/validation#257] Support ConstraintValidator declaration via service loader

** Improvement
    * [jakartaee/validation#257] Set the default validatedBy on the Constraint annotation
    * [jakartaee/validation#270] Introduce ConstraintDescriptor#getAttribute(String attributeName, Class<V> type) to
simplify working with constraint attributes
    * [jakartaee/validation#301] Make Min/Max constraints to be applicable to the string representation of the number
    * [jakartaee/validation#268] Make jakarta.validation.Validation final and hide the constructor

** Task
    * [jakartaee/validation#288] Move XSDs in jar from the root to META-INF
    * [jakartaee/validation#260] Actually deprecate methods on ConstraintViolationBuilder (that previously only
mentioned that they are deprecated in the javadoc)
    * [jakartaee/validation#260] Clarify the expectations on applying the @Valid twice
    * [jakartaee/validation#259] Fix incorrect example in table 5.1 Resolution of ConstraintValidator for various
constraints declarations

3.1.0 (2024-01-26)
------------------
** New Feature
    * EE11, clarify the requirements around Java records and add basic TCK tests for them.

3.0.0 (2020-07-03)
------------------
** New Feature
    * EE9, all API package prefixes have changed from javax.validation to jakarta.validation.

2.0.2 (2019-08-01)
------------------

** Improvement
    * [BVAL-721] - Do not include the license check configuration files into the published
                   artifact

** Task
    * [BVAL-736] - Update TestNG to 6.11 for consistency with the TCK
    * [BVAL-730] - Move specification under Eclipse (EE4J umbrella)
    * [BVAL-723] - Upgrade Asciidoctor tooling
    * [BVAL-720] - Create a specific branch for the Jakarta artifact and adjust the GAV

2.0.1.Final
-----------

Skipped to sync API and specification releases for the first Jakarta EE release.

2.0.0.Final (2017-08-03)
------------------------

2.0.0.CR3 (Final Approval Ballot, 2017-07-11)
---------------------------------------------

** Improvement
    * [BVAL-693] - Make order of <constraint> and <container-element-type> consistent in XSD
    * [BVAL-692] - Misc. findings after review
    * [BVAL-690] - Clarify getLeafBean() for container element constraints
    * [BVAL-464] - Create CONTRIBUTING.md page

** Task
    * [BVAL-687] - Update license information for submission to Final Approval Ballot

2.0.0.CR2 (2017-07-05)
```

```
----------------------

** Bug
    * [BVAL-569] - Remove TYPE from targets of @ConvertGroup for consistency with @Valid

** Improvement
    * [BVAL-689] - Make @ConvertGroup#from() default to Default.class
    * [BVAL-685] - Use the full URLs in xsi:schemaLocation of our XML files
    * [BVAL-683] - Fix the release scripts: changelog injection needs to be updated
    * [BVAL-682] - Misc. clarifications around container element validation
    * [BVAL-675] - ValueExtractor exceptions should be wrapped in ValidationException
    * [BVAL-674] - Rename ConstraintDescriptor#validateUnwrappedValue() into
                   getValueUnwrapping()
    * [BVAL-673] - In @Future/@Past and allegates, use imports instead of FQNs (except for
                   the date types)
    * [BVAL-667] - Avoid usage of @NotEmpty example constraint in the spec
    * [BVAL-604] - Use less characters per line to avoid forced line breaks in the rendered
                   PDF

** New Feature
    * [BVAL-510] - Fix JavaDoc warnings, copy changed/new JavaDoc into spec

** Task
    * [BVAL-688] - Upgrade to hibernate-asciidoctor-theme 1.0.2.Final
    * [BVAL-686] - Add JavaDoc for class-level type parameters
    * [BVAL-681] - Clarify value returned by getInvalidValue() for implicitly unwrapped
                   container constraint
    * [BVAL-678] - Clarify "If there is exactly one maximally-specific type-compliant value
                   extractor marked with @UnwrapByDefault, this extractor is applied"
    * [BVAL-677] - Add a section for each built-in constraint so that they appear in the TOC
    * [BVAL-676] - Differentiate the API listings from the examples
    * [BVAL-613] - Add an Automatic-Module-Name entry to the JAR manifest
    * [BVAL-608] - Review the remaining examples in the API to see if we can get rid of them

2.0.0.CR1 (Proposed Final Draft 1, 2017-06-21)
----------------------------------------------

** Bug
    * [BVAL-657] - Fix wrong map key in example

** Improvement
    * [BVAL-671] - Address Yoann's review remarks
    * [BVAL-668] - Address Guillaume's review remarks
    * [BVAL-666] - Clarify that constraints on type parameters are unsupported
    * [BVAL-664] - ValueExtractor exceptions should be wrapped in ValidationException
    * [BVAL-661] - Proof-read on "type argument" vs. "type parameter" and "generic type" vs.
                   "parameterized type"
    * [BVAL-660] - Constructors from super-classes are not considered by
                   getConstraintsForConstructor()
    * [BVAL-659] - Fix some cut-off listings
    * [BVAL-658] - Some JavaDoc fixes
    * [BVAL-653] - In Examples for method and constructor constraint violations, we still use
                   the legacy @Valid location
    * [BVAL-649] - Use parameter names instead of arg<i> in Table 6.2: Property path examples
                   for constrained methods or constructors
    * [BVAL-647] - Relax a few tck-testable assertions
    * [BVAL-646] - Missing assertions in 9.1.1.6. Container-element overriding (XML definition
                   of container element constraints)
    * [BVAL-631] - Clarify which type should be used as the containerClass
    * [BVAL-630] - Add containerClass info in propertyPath examples and add examples for
                   container element
    * [BVAL-618] - Add missing tck-testable markers
    * [BVAL-616] - Provide @PositiveOrZero and @NegativeOrZero constraints

** New Feature
    * [BVAL-672] - Create new constraints @PastOrPresent and @FutureOrPresent

** Task
    * [BVAL-670] - Contradictory sentences about the default locale
    * [BVAL-663] - Raise an exception if both Unwrap and Skip are present in the payload
```

```
    * [BVAL-662] - Adjustment on the value extractor resolution algorithm for "non-generic
                   containers"
    * [BVAL-656] - Remove unused imports in Past and Future constraints
    * [BVAL-655] - Allow metadata API to expose container element constraints inherited from
                   the hierarchy
    * [BVAL-652] - State in the spec that a ValueExtractorDeclarationException will be raised
                   by addValueExtractor()
    * [BVAL-650] - Improve the value-extractor XML element description
    * [BVAL-648] - Add example of OptionalInt to value extractor resolution chapter
    * [BVAL-640] - Explicitly state that ValueExtractor#extractValues() must not be called if
                   the container is null
    * [BVAL-639] - Address Emmanuel's review remarks
    * [BVAL-637] - Indicate the extracted type for OptionalInt... value extractors
    * [BVAL-636] - Mandate JavaFX extractors only in environments with JavaFX present
    * [BVAL-635] - Add current git revision to generated tck-audit.xml
    * [BVAL-634] - @OverridesAttribute#name() should have a default value as stated in its
                   docs
    * [BVAL-628] - Fix node builder assertions
    * [BVAL-625] - Clarify structure of property paths for nested containers
    * [BVAL-624] - Clarify whether value extractors discovered through the service loader
                   benefit from CDI
    * [BVAL-601] - Clarify semantics around nested usage of @Valid
    * [BVAL-552] - Clarify the isInIterable() return value for arrays
    * [BVAL-551] - Consider removing the cache of validation providers

2.0.0.Beta2 (Public Review Draft 1, 2017-05-17)
-----------------------------------------------

** Bug
    * [BVAL-627] - Build - Always download new artifacts on CI

** Improvement
    * [BVAL-620] - Clarify semantics when @Valid is used for a collection in the old and the
                   new style at the same time
    * [BVAL-617] - Fix typo "in happening" in Group conversion paragraph
    * [BVAL-615] - Use the new asciidoctor-ant core artifact and upgrade to the latest
                   Asciidoctor
    * [BVAL-614] - Use new Ant 1.9 features to simplify the dependency management in our
                   build.xml
    * [BVAL-611] - Use the new hibernate-asciidoctor-extensions project
    * [BVAL-610] - Inject the section id constants in tck-audit.xml
    * [BVAL-609] - Consider JavaFX's set types in the list of built-in extractors
    * [BVAL-607] - Fix wrong message key in example constraint
    * [BVAL-605] - Avoid empty initialize() methods in spec examples
    * [BVAL-603] - Clarify that CDI is available to value extractors
    * [BVAL-600] - Only use the CustomRoleBlockProcessor for the DocBook output
    * [BVAL-526] - Restart example numbering for each chapter

** New Feature
    * [BVAL-592] - Consider container element constraints in node builder API
    * [BVAL-579] - Support OptionalInt, OptionalLong, OptionalDouble
    * [BVAL-517] - Define module name for BV API

2.0.0.Beta1 (Public Review Draft 1, 2017-04-24)
-----------------------------------------------

** Improvement
    * [BVAL-598] - Use the license-maven-plugin to check the presence of the license
    * [BVAL-596] - Move ValidateUnwrappedValue to javax.validation.metadata
    * [BVAL-595] - Adding missing @since tags

** New Feature
    * [BVAL-594] - Extend meta-data API to cover container element constraints
    * [BVAL-593] - Allow to configure group conversions for container elements in XML
    * [BVAL-591] - Dissolve "container element validation" appendix into spec sections
    * [BVAL-589] - Document release process of the spec
    * [BVAL-549] - Nested cascaded validation

** Task
    * [BVAL-602] - Check that every section has an id
```

    * [BVAL-599] - Remove metadata usage example from API
    * [BVAL-585] - Add descriptions of build goals to beanvalidation-spec/README.md
    * [BVAL-584] - Use short license headers
    * [BVAL-581] - Discuss with Java EE EG whether it's actually WEB-INF/validation.xml
    * [BVAL-577] - Proof-read "3.3. Constraint composition" on "composing" vs. "composed"
    * [BVAL-575] - Update documentation of Configuration#getDefaultParameterNameProvider()

2.0.0.Alpha2 (2017-03-28)

** Bug
    * [BVAL-566] - Configuration JavaDoc still references "TODO BVAL-496 paste definition of
                   the specification"

** New Feature
    * [BVAL-548] - New constraints: @NotEmpty, @NotBlank, @Email, @Positive, @Negative
    * [BVAL-559] - Add NoProviderFoundException
    * [BVAL-562] - Create new ElementKind for constraint violations on the type-use level
    * [BVAL-571] - Discover value extractor implementations using the service loader mechanism
    * [BVAL-572] - Allow to add value extractors per validator factory and validator and via
                   validation.xml
    * [BVAL-586] - Support defining constraint on container element type in XML mapping

** Task
    * [BVAL-560] - Mention Oracle instead of Sun in the evaluation license
    * [BVAL-565] - Configure japicmp plug-in for creating API change report
    * [BVAL-578] - Make the assertion related to NoProviderFoundException not testable
    * [BVAL-587] - Go back to coderay for code highlighting in the spec

** Improvement
    * [BVAL-264] - Improve toString() in ConstraintViolationException
    * [BVAL-528] - Replace all inline listings with includes of the actual API files / spec
                   examples
    * [BVAL-561] - Make @OverridesAttribute repeatable and documented
    * [BVAL-567] - Tone down @Past / @Future JavaDoc on clockprovider
    * [BVAL-568] - Make built-in annotations documented
    * [BVAL-576] - Revise wording in 10.3.3 "Method and constructor validation"
    * [BVAL-580] - Switch assertion section ids to a string instead of a numeric
                   representation
    * [BVAL-583] - Use the common theme for the asciidoc output

2.0.0.Alpha1 (Early Draft 1, 2017-02-02)

** Bug
    * [BVAL-558] - Ensure correct concurrent access to cacheValidationProviders in
                   Validation#GetValidationProviderListAction

** New Feature
    * [BVAL-467] - Support JDK8's Optional class
    * [BVAL-496] - @Future/@Past validators for JSR 310 data types
    * [BVAL-497] - Mark BV-defined constraint types with @Repeatable
    * [BVAL-498] - Retrieve method parameter names via new API in Java 8
    * [BVAL-508] - Offer validation of values contained in containers
    * [BVAL-544] - Promote TYPE_USE usage for constraint annotations
    * [BVAL-550] - Make @ConvertGroup repeatable and usable on type arguments

** Task
    * [BVAL-509] - Make BVAL compilable with Java 9
    * [BVAL-512] - Provide a way to reference validation-api source code in the spec
    * [BVAL-513] - Add Google Analytics to HTML version of the specification
    * [BVAL-538] - Rename createTckAuditFile ant task to create-tck-audit-file for consistency
    * [BVAL-540] - Use Maven to download the snapshot of the API sources
    * [BVAL-541] - Change CI links in README.md
    * [BVAL-547] - Use simplified license header for API files
    * [BVAL-554] - Add the ability to generate an asciidoc document containing all the spec in
                   a single file

** Improvement

```
    * [BVAL-455] - Move XML namespace to jcp.org from jboss.org
    * [BVAL-460] - Set ignore-annotation default value for beans to true in XSD to document
                   the spec behavior
    * [BVAL-486] - Do not use validation provider resolver when provider is explicitly given
    * [BVAL-527] - Raise Java baseline to version 8
    * [BVAL-529] - Mention JSR 380 and BV 2.0 in the spec
    * [BVAL-530] - Rename master.asciidoc to index.asciidoc
    * [BVAL-532] - Fix javadoc warnings
    * [BVAL-533] - Update Maven dependencies
    * [BVAL-534] - Update all outdated links to java.sun.com
    * [BVAL-535] - Reduce the number of compilation warnings in the API
    * [BVAL-536] - Update tck-audit.xsl to the current version of BV
    * [BVAL-537] - Make section numbers stable from 1.1 to 2.0
    * [BVAL-539] - Set the version of the maven-deploy-plugin
    * [BVAL-545] - Fix description of validateValue() method
    * [BVAL-546] - Make ValidatorFactory extend AutoClosable
    * [BVAL-555] - Provide default implementation for ConstraintValidator#initialize()


1.1.0.Final (2013-04-10)
------------------------

** Improvement
    * [BVAL-452] - Remove @Deprecate annotation from addNode() method


1.1.0.CR3 (2013-03-20)
----------------------

** Bug
    * [BVAL-444] - Remove revisionflags from specification
    * [BVAL-445] - Do not consider arrays of primitives equivalent to arrays of wrappers in
                   ConstraintValidation resolution

** Improvement
    * [BVAL-448] - Mention "boolean" instead of "Boolean" in getter definition
    * [BVAL-450] - Make @ValidateOnExecutable for @Override methods raise an exception

** Task
    * [BVAL-449] - Remove tck-needs-update


1.1.0.CR2 (2013-03-14)
----------------------

** Bug
    * [BVAL-431] - Typo in EL expression exemple
    * [BVAL-435] - Rename element <validated-executables/> to
                   <default-validated-executable-types/>
    * [BVAL-436] - Offer global switch to disable executable validation altogether
    * [BVAL-437] - Redesign @ValidateExecutable into @ValidateOnExecution and as CDI marker
                   for portability

** Improvement
    * [BVAL-420] - Add missing @since in metadata package
    * [BVAL-421] - Clarify whether or not getters are provided by the metadata API
    * [BVAL-422] - Update TCK markers
    * [BVAL-423] - Clarify behavior of ConstraintViolation#getLeafBean() for validateValue()
    * [BVAL-424] - Refer to @SupportedValidationTarget in ConstraintValidation JavaDoc
    * [BVAL-429] - Throw a ConstraintDefinitionException if there a several cross-parameter
                   validators
    * [BVAL-432] - Rename areParametersConstrained to hasConstrainedParameters and
                   isReturnValueConstrained to hasConstrainedReturnValue on
                   ExecutableDescriptor
    * [BVAL-433] - Forbid @ValidatedExecutable on methods of parallel hierarchies
    * [BVAL-434] - Clarify exception type if cross-parameter validator support neither Object
                   nor Object[]
    * [BVAL-440] - Improve description of ExecutableType.GETTER_METHODS
    * [BVAL-442] - Make getConstrainedMethods(MethodType methodType,
                   MethodType... methodTypes) to be less error-prone
```

```
** New Feature
    * [BVAL-441] - Allow for identical configuration in subtypes

** Task
    * [BVAL-425] - Raise IllegalArgumentException when validateParameters and
                   validateReturnValue are passed parameters that do not match
    * [BVAL-428] - Map remaining assertions for BV 1.1
    * [BVAL-438] - Integration chapter should not mention that the boostrap API can be used to
                   create additional ValidationFactory
    * [BVAL-439] - Clarify that EE validator factory supports CDI services


1.1.0.CR1 (proposed final draft, 2013-02-20)
--------------------------------------------

** Bug
    * [BVAL-322] - Formatting and style improvements
    * [BVAL-369] - Specify copyright year correctly in license headers
    * [BVAL-391] - Use @SupportValidationTarget instead of @CrossParameterConstraint for
                   cross-parameter constraint validators
    * [BVAL-397] - Align the JavaDoc on temps (return vs returns, define vs defines)
    * [BVAL-401] - validateReturnValue should not throw an exception if the method has no
                   return value
    * [BVAL-402] - Remove notion of "reachable" parameters in method validation routine
    * [BVAL-403] - Add example on method validation to 4.6.3. ("Traversable property")
    * [BVAL-407] - ConstraintViolation.unwrap parameterized type hides ConstraintViolation
                   parameterized type

** Improvement
    * [BVAL-275] - Align on style for referencing methods in spec text
    * [BVAL-277] - Align on style for author names in JavaDoc
    * [BVAL-285] - ValidatorFactory#close should clearly state post conditions
    * [BVAL-350] - Add more examples on how to use methods for validating method and
                   constructor constraints
    * [BVAL-362] - Reference the various specs (JPA, JSF, CDI, JavaBeans)
    * [BVAL-400] - Add xml and exception chapters to the list in "How this document is
                   organized"
    * [BVAL-404] - Path examples in table 5.2 are missing node specific attributes like
                   parameterIndex
    * [BVAL-405] - Clarify what isBeanConstrained does and add hasExecutableConstrained
    * [BVAL-406] - Add ConstraintDescriptor.getValidationAppliesTo() and getMessageTemplate()
    * [BVAL-409] - Make ParameterNameProvider use List instead of arrays
    * [BVAL-410] - Make node creation suppress the cross-param and bean-level node in case of
                   subnode creation
    * [BVAL-412] - Make <convert-group/> follow <valid/> and precede <constraint/> in the
                   mapping XSD
    * [BVAL-413] - Fix method validation and ConstraintViolation example
    * [BVAL-414] - Add example for metadata API with executables
    * [BVAL-415] - Make sure maven plugins are set in beanvalidation-api
    * [BVAL-417] - Mention "validationAppliesTo" in docs of @SupportedValidationTarget
    * [BVAL-419] - Clarify that using a cross-parameter constraint on a method without
                   parameter is illegal


1.1.0.Beta4 (2013-02-15)
------------------------

** Sub-task
    * [BVAL-316] - Decide on whether to allow validation of static methods or not
    * [BVAL-330] - Refinements around metadata API

** Bug
    * [BVAL-221] - The constraint violation builder cannot put constraint on a top level map
                   key
    * [BVAL-283] - Clarify that ConstraintValidator instances must be destroyed after each
                   method validation call if the ConstraintValidatorFactory is provided to the
                  Validator
    * [BVAL-284] - Clarify that ConstraintValidator instances passed to CVF.releaseInstance
                   must be coming from the CVF creating them
```

```
    * [BVAL-326] - Fix metadata and error reports for cross-parameter validation
    * [BVAL-328] - Add recommendation that @Inherited shouldn't be added to constraint
                   annotation types
    * [BVAL-337] - Clarifications around ConstraintViolation for method validation
    * [BVAL-370] - Re-consider how cross-parameter constraints are represented in metadata API
                   and XML descriptors
    * [BVAL-375] - Add dedicated "validationAppliesTo" element to schema type representing
                   constraints
    * [BVAL-378] - Mismatch between enum ExecutableType and corresponding schema type
    * [BVAL-380] - Remove improper sentence around constraint being validated once globally in
                   validation routine
    * [BVAL-381] - Specify which path is pathed to traversable resolvers in case of cascaded
                   method validation
    * [BVAL-388] - Create sub-types of Node instead of Node#getElementDescriptor() and remove
                   ElementDescriptor.getKind()
    * [BVAL-389] - @ValidateExecutable.type should default to ALL and NONE should be renamed
                   OFF
    * [BVAL-390] - Clarify syntax for specifying parameter types in XML
    * [BVAL-393] - Revert "intersection type trick"

** Improvement
    * [BVAL-191] - Introduce a addBeanNode() method to the fluent node builder API
    * [BVAL-269] - Polish support for dependency injection after draft feedback
    * [BVAL-336] - Decide what to do about element descriptor when using constraint violation
                   builder API
    * [BVAL-344] - Improve wording around CDI integration
    * [BVAL-368] - Return constant value from Node#getName() for return value nodes
    * [BVAL-372] - Consider moving ExecutableValidator to the executable subpackage
    * [BVAL-379] - Clarify that modifications to BootstrapConfiguration have no effect
    * [BVAL-384] - Add example for ElementDescriptor#findConstraints() for methods
    * [BVAL-385] - Return void ReturnValueDescriptor from
                   ExecutableDescriptor#getReturnValueDescriptor() for void methods
    * [BVAL-386] - Clarify that CDI integration is mandatory under Java EE only
    * [BVAL-398] - Make validateReturnValue raise ValidationException if the method has no
                   return value

** New Feature
    * [BVAL-329] - Method validation support (III)
    * [BVAL-383] - Add a  unwrap method in ConstraintViolation
    * [BVAL-387] - Add ability to add a node corresponding to a parameter in
                   ConstraintViolationBuilder

** Task
    * [BVAL-394] - Verify that we don't need a spec defined API to expose classes hosting
                   constrained methods or constructor defined in XML


1.1.0.Beta3 (2013-02-01)
------------------------

** Sub-task
    * [BVAL-273] - Extend the XML descriptor schema to represent method-level constraints
    * [BVAL-314] - Provide ability to disable validation for method/constructor validation

** Bug
    * [BVAL-327] - Provide way to change the executable validation (ie accept getters)
    * [BVAL-342] - Clarify that validateProperty / validateValue does not support property
                   paths
    * [BVAL-343] - "Provider org.hibernate.validator.HibernateValidator not a subtype" error
                   during service discovery
    * [BVAL-345] - List of messages in the standard resource bundle is incomplete
    * [BVAL-346] - Clarify that getters must have no parameter
    * [BVAL-347] - Add implicit assumptions from TCK to spec text
    * [BVAL-351] - Clarify that EntityManager cannot be injected if validating from JPA
    * [BVAL-361] - Expose group conversions via meta-data API
    * [BVAL-363] - Clarify that super method constraints are considered in the validation
                   routine but not constructors
    * [BVAL-366] - Fix typo on ConfigurationState JavaDoc
    * [BVAL-371] - Add package level javadoc (package-info.java)
    * [BVAL-377] - Provide MessageInterpolator.Context#unwrap to allow for custom extensions
```

```
** Improvement
    * [BVAL-192] - Add 'exclusive' boolean attribute to @DecimalMin/@DecimalMax constraints
    * [BVAL-332] - Specify semantics of @ConvertGroup when given several times at overridden
                   property
    * [BVAL-340] - Denote method parameter constraints at declaration site (vs. at definition
                   site)
    * [BVAL-352] - Clarify what managed means in the integration chapter in particular for CDI
    * [BVAL-359] - Relax contract of ExecutableDescriptor#getParameterDescriptors()
    * [BVAL-360] - Describe IllegalArgumentException for ExecutableValidator methods
    * [BVAL-364] - Clarify whether or not the metadata API ignore the method enable/disable
                   settings
    * [BVAL-365] - Clarifications around group conversion in hierarchies
    * [BVAL-367] - Make clear whether methods/properties inherited from super types are
                   reflected by the meta-data API
    * [BVAL-373] - Move ConvertGroup to the groups subpackage

** New Feature
    * [BVAL-219] - Add support for interpolating the value in error messages
    * [BVAL-223] - Add formatter syntax for interpolated messages via EL expression support
    * [BVAL-249] - Add unwrap method to ConstraintValidatorContext for provider extension
    * [BVAL-333] - Enable configuration of group conversions via XML

** Task
    * [BVAL-338] - Clarify lifecycle of managed objects created by BV povider
    * [BVAL-348] - Add example for illegal group conversion on a return value in an
                   inheritance hierarchy
    * [BVAL-349] - Mark spec sentences as TCK-relevant (1.0 assertions)
    * [BVAL-353] - Mark spec sentences as TCK-relevant (1.1 assertions)
    * [BVAL-354] - Describe tagging of TCK-relevant sentences in README.md
    * [BVAL-355] - Rename Validator#forMethods() to forExecutables()
    * [BVAL-357] - Clarify that traversable resolver is not used on parameter and return
                   values during method validation
    * [BVAL-358] - Make ExecutableDescriptor#validateConstructorParameters() and
                   validateConstructorReturnValue() more usable
    * [BVAL-374] - Clarify exceptional case in section 5.5.5 bootstrapping
    * [BVAL-376] - Remove @MethodValidated as it is not adding value to the CDI integration


1.1.0.Beta2 (2012-11-27)
------------------------

** Sub-task
    * [BVAL-331] - Establish common super-interface for MethodDescriptor and
                   ConstructorDescriptor

** Bug
    * [BVAL-335] - @ConvertGroup.List is missing target types and retention policy

** Improvement
    * [BVAL-198] - Simplify creation of ConstraintViolationExceptions
    * [BVAL-334] - Refer to CDI provided beans as "built-in" beans


1.1.0.Beta1 (public review draft, 2012-10-19)
---------------------------------------------

** Sub-task
    * [BVAL-232] - Support cross-parameter constraints
    * [BVAL-274] - Extend the meta-data API with required convenience methods for method
                   validation
    * [BVAL-290] - Mark new method with @since annotation
    * [BVAL-300] - Clarify behavior of constructor validation in class hierachies
    * [BVAL-308] - Settle on approach for constraint refinement in sub-types
    * [BVAL-309] - Specify logic to be implemented by method validation interceptors
    * [BVAL-310] - Move methods related to method validation to delegate interface
    * [BVAL-317] - Rename 'method-level validation' with 'method validation'

** Bug
    * [BVAL-296] - Example using ConstraintValidatorContext is incorrect
```

```
    * [BVAL-298] - DefaultValidationProviderResolver should check context and current class
                   loader for service file
    * [BVAL-304] - Add OSGi headers in the reference implementation
    * [BVAL-306] - Clarify interceptor order in method validation triggering

** Improvement
    * [BVAL-208] - Support groups translation during cascaded validations
    * [BVAL-226] - Make clear whether the static or the runtime type should be considered when
                   creating property paths in case of cascaded validations
    * [BVAL-230] - Add support for validating CharSequence types instead of just Strings
    * [BVAL-259] - Evaluation of composed constraints should stops on first validation error
                   in case of @ReportAsSingleViolation
    * [BVAL-281] - Improve message when building a ValidatorFactory but no provider is
                   available in the classpath
    * [BVAL-292] - Clarify the behavior of ConfigurationSource methods when no configuration
                   file is present
    * [BVAL-299] - Add note on required Java version

** New Feature
    * [BVAL-272] - Method validation support (II)
    * [BVAL-295] - Should validation-configuration and validation-mapping xsds define a
                   version attribute

** Task
    * [BVAL-280] - Decide whether DefaultValidationProviderResolver should not throw an
                   exception when a specified provider cannot be loaded
    * [BVAL-307] - Decide how CDI and Bean Validation is integrated


1.1.0.Alpha1 (early draft 1, 2012-03-13)
----------------------------------------

** Sub-task
    * [BVAL-242] - Extend the meta-data API to represent method-level constraints
    * [BVAL-243] - Provide a means for specifying method parameter names
    * [BVAL-244] - Extend Validator API with methods for method validation
    * [BVAL-245] - Define how method constraints are declared at parameters and return values

** Bug
    * [BVAL-194] - Invalid license info
    * [BVAL-196] - Missing </code> element in Javadocs for
                   ConstraintValidatorContext.ConstraintViolationBuilder.NodeContextBuilder
    * [BVAL-212] - Wrong closing </code> element in javadocs of BeanDescriptor
    * [BVAL-236] - Fails to load META-INF/services provider configuration files on non-ASCII
                   platforms

** Improvement
    * [BVAL-201] - Fix typo in spec, chapter 4.4.3
    * [BVAL-270] - Specify that Bean Validation 1.1 providers must support deployment
                   descriptors version 1.0

** New Feature
    * [BVAL-238] - Support for container injection in ConstraintValidator
    * [BVAL-241] - Support for method validation
    * [BVAL-258] - Clean introduction section to reflect Bean Validation 1.1
    * [BVAL-263] - Add a close() method to ValidatorFactory
    * [BVAL-265] - Expose settings defined in XML in the Configuration API
                   (for ConstraintValidatorFactory, MessageInterpolator etc)

** Task
    * [BVAL-206] - Update pom to use the new distributationManagement information
    * [BVAL-228] - Prepare specification document and Git repository for public eyes
    * [BVAL-279] - Update POM file for Bean Validation API to use latest Git repo urls and
                   generally be ready for a release


1.0.0 final (2009-10-12)
------------------------

** Bug
```

```
    * [BVAL-181] - Fix some namespace issues in validation-configuration-1.0.xsd

** Improvement
    * [BVAL-182] - Add getDefaultTraversableResolver and getDefaultConstraintValidatorFactory
                   to Configuration
    * [BVAL-183] - Add getTraversableResolver and getConstraintValidatorFactory to
                   ValidatorFactory
    * [BVAL-184] - Replace Red Hat Middleware LLC to Red Hat, Inc. and/or its affiliates
    * [BVAL-186] - Clarify method names on the constraint violation builder DSL of
                   ConstraintValidatorContext
    * [BVAL-187] - Imply that ConstraintViolation is serializable if entities are serializable

** New Feature
    * [BVAL-185] - Allow overriding of ConstraintValidatorFactory when creating a Validator
    * [BVAL-190] - Add methods to filter ConstraintDescriptor per groups, target and scope

** Task
    * [BVAL-132] - Define behaviour for BeanDescriptor.getConstraintsForProperty(null)


1.0.CR5 (2009-08-27)
--------------------

** Bug
    * [BVAL-173] - Fix typo getUnorderdConstraintDescriptorsMatchingGroups =>
                   getUnorderedConstraintDescriptorsMatchingGroups
    * [BVAL-177] - Payload of composed constraints are ignored, the main constraint payload is
                   propagated
    * [BVAL-178] - Add payload to the XML schema
    * [BVAL-180] - ConstraintDescriptor.getPayload() should return
                   Set<Class<? extends Payload>> not Set<Class<Payload>>

** Improvement
    * [BVAL-174] - clearer default message for assertTrue and assertFalse
    * [BVAL-179] - Rename ConstraintPayload to Payload


1.0.CR4 Unpublished release
---------------------------


1.0.CR3 (Proposed Final Draft 2, 2009-07-08)
--------------------------------------------

** Bug
    * [BVAL-144] - validation-configuration.xsd property element does not extend basic string
                   type preventing Oxygen to be happy
    * [BVAL-159] - Fix example 3.8 on object graph validation

** Improvement
    * [BVAL-143] - Describe path with an object model
    * [BVAL-147] - Support for unbounded wildcards in ConstraintValidator
    * [BVAL-148] - Built-in constraints annotations now annotated with
                   @Constraint(validatedBy={})
    * [BVAL-151] - TraversableResolver#isTraversable can receive null traversableObject when
                   valudateValue is called
    * [BVAL-152] - TraversableResolver should differentiate reachability and cascadability
    * [BVAL-153] - Generify ConstraintValidatorException
    * [BVAL-154] - Iterable is a superclass of all collection, clarify it's interaction with
                   @Valid
    * [BVAL-155] - ignore-annotation is not inherited hierarchically: make that explicit
    * [BVAL-156] - Pattern.Flag takes the JDK flag int at construction time
    * [BVAL-157] - Add [] to non-indexed iterable path
    * [BVAL-158] - Clarify that @Valid is orthogonal to the idea of group
    * [BVAL-160] - rename message template key as [f.q.c.n of the constraint].message
    * [BVAL-162] - Move metadata classes to the metadata package
                   (BeanDescriptor, ElementDescriptor, PropertyDescriptor,
                   ConstraintDescriptor)
    * [BVAL-164] - Validation.byProvider now accept the provider implementation class
    * [BVAL-166] - IllegalArgumentException raised on BeanDescriptor.getConstraintsForProperty
```

```
                and Validator.getConstraintsForClass
    * [BVAL-167] - Recommend f.q.c.n.message for resource bundle keys and migrate examples
    * [BVAL-169] - Rename ElementDescriptor.getType to getElementClass
    * [BVAL-170] - Let built-in annotations to support ElementType.PARAMETER and
                   ElementType.CONSTRUCTOR

** New Feature
    * [BVAL-149] - Provide access to the ValidationProviderResolver via BootstrapState
    * [BVAL-150] - Add ConstraintViolation.getRootBeanClass
    * [BVAL-161] - Add unwrap methods to ValidatorFactory and Validator
    * [BVAL-163] - Add support for constraint payload
    * [BVAL-168] - Return the list of matching ConstraintDescriptor for a given set of groups
    * [BVAL-172] - Provide ConstraintDescriptor#getPayload


1.0.CR2 Unpublished release
---------------------------


1.0.CR1 (Proposed Final Draft, 2009-03-16)
------------------------------------------

** Bug
    * [BVAL-118] - ConstraintDescriptor.getGroups() returns Default if no group is declared on
                   the constraint
    * [BVAL-125] - @Size.min default value should be 0

** Improvement
    * [BVAL-32] - Describe what is happening when a composition is not consistent
    * [BVAL-50] - Be consistent in the spec, use @author or not
    * [BVAL-54] - Specify that constraints on non getter methods are ignored (if BVAL-36 is
                  not accepted)
    * [BVAL-72] - Validating an object multiple times if in a different branch of the graph
    * [BVAL-86] - Default TraversableResolver is JPA aware
    * [BVAL-88] - Improvement on MessageInterpolator
    * [BVAL-91] - Rename Constraint related classes to improve readability
    * [BVAL-95] - @Size should support Map
    * [BVAL-96] - Support byte in @Min/@Max
    * [BVAL-106] - Constraintdescriptor.getConstraintValidatorClasses() should return a List,
                   not an array
    * [BVAL-114] - Relax property names in ConstraintValidatorContext
    * [BVAL-120] - Rename ConstraintViolation getRawMessage=>getMessageTemplate,
                   getInterpolatedMessage=>getMessage
    * [BVAL-122] - Rename @GroupSequence.sequence to @GroupSequence.value
    * [BVAL-126] - Define group sequence logic more formally and eliminate corner cases
    * [BVAL-129] - Clarify ConstraintValidatorContext propertyPath generation
    * [BVAL-130] - Make ConstraintDescriptor generic:
                   ConstraintDescriptor<T extends Annotation>
    * [BVAL-131] - Provide object graph navigation determinism
    * [BVAL-134] - @Valid accepts objects implementing Iterable
    * [BVAL-135] - Remove DefaultValidationProviderResolver from the public API
    * [BVAL-136] - Add Context object for MessageInterpolator
    * [BVAL-137] - prefix for message template key is constraint. instead of validator.
    * [BVAL-138] - Rename OverridesParameter to OverridesAttribute
    * [BVAL-139] - Remove @OverridesParameters and use the inner class mode
                   (OverridesAttribute.LIst)
    * [BVAL-140] - BeanDescriptor.getConstrainedProperties() returns Set<PropertyDescriptor>
    * [BVAL-141] - Rename ConstraintDescriptor.getParameters() to getAttributes()

** New Feature
    * [BVAL-52] - Define the exception hierarchy and rules
    * [BVAL-55] - Exception policy
    * [BVAL-65] - Additional built-in constraints
    * [BVAL-98] - Type-safe ConstraintValidator
    * [BVAL-100] - Support XML mapping overriding
    * [BVAL-102] - Support META-INF/validation.xml
    * [BVAL-119] - Introduce @Pattern for regexp
    * [BVAL-121] - Define built-in constraints plural forms
    * [BVAL-123] - Add ConstraintViolationException
    * [BVAL-124] - Introduce backslash as escaping character
```

    * [BVAL-142] - @Min/@max no longer accept float/double and introduce
                   @DecimalMin/@DecimalMax

** Task
    * [BVAL-24] - What should be done when multiple META-INF/validation.xml are found?
    * [BVAL-117] - Specify behaviour of ConstraintValidator.initalize in the case of
                   inconsistent values in constraint parameters
    * [BVAL-127] - Remove ConstraintViolation.getGroups()
    * [BVAL-128] - Clarify invalid cases for validateProperty / validateValue on proeprtyName
                   being empty or null
    * [BVAL-133] - Remove JPA and JSF integration proposals

1.0.Beta2 (Public Draft, 2008-12-15)

** Bug
    * [BVAL-6] - Wrong example in validation methods section
    * [BVAL-17] - Validator<A>.validate(b) where b:B and B extends A should validate B.
                  Metadata APIs are specific to A
    * [BVAL-42] - Names of message keys in spec inconsistent
    * [BVAL-45] - Typo at ConstraintDescriptor.getContstraintClass()

** Improvement
    * [BVAL-29] - Types should be determined at runtime
    * [BVAL-33] - Should ConstraintDescriptor.getConstraintImplementation() replaced by
                  .getConstraintImplementationClass()?
    * [BVAL-40] - Rename InvalidConstraint to ConstraintViolation
    * [BVAL-48] - Add a way to access the default message resolver
    * [BVAL-49] - Mark metadata classes as immutable
    * [BVAL-59] - Rethink the group sequence inheritance rules
    * [BVAL-60] - ConstraintViolation points to the corresponding ConstraintDescriptor
    * [BVAL-68] - Specify that static methods and fields are not validated
    * [BVAL-73] - Rename ConstraintViolation.getBeanClass() to CV. getRootClass() or simply
                  remove it
    * [BVAL-78] - Forbid a Validation implementation to modify the state of the object being
                  validated

** New Feature
    * [BVAL-30] - Define validation Context to be passed to constraint implementation calls
    * [BVAL-36] - Validation of method parameters and returned values
    * [BVAL-67] - Allow MessageResolver to be Localizable
    * [BVAL-71] - Should we have group aggregation?
    * [BVAL-76] - Expose the raw message to ConstraintViolation
    * [BVAL-79] - Groups are now Type based rather than String based
    * [BVAL-81] - Provide a TraversableResolver contract

** Task
    * [BVAL-1] - Remove references to 'beancheck' in the spec
    * [BVAL-3] - Replace array return types with Sets
    * [BVAL-4] - Return value for @NotEmpty for null values
    * [BVAL-5] - Change order of exmaple classes in Book/Author example
    * [BVAL-7] - Use of example in ConstraintFactory section (2.4)
    * [BVAL-8] - StandardConstraint description (2.5)
    * [BVAL-23] - Make Validator<T> thread-safe