JSR 303: Bean Validation

Bean Validation Expert Group

Specification lead
Emmanuel Bernard, Red Hat, Inc.

Version
1.0 final
2009-10-12

Table of Contents

1

Introduction

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

1.1. Expert group

This work is being conducted as part of JSR 303 under the Java Community Process Program. This specification is the result of the collaborative work of the members of the JSR 303 Expert Group. These include the following present and former expert group members in alphabetical order:

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

The validation API developed by this JSR 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 Java-Beans 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.

1.3. How this document is organized

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

Chapter 2 describes how constraints are defined.

Chapter 3 describes how a JavaBean class is decorated with annotations to describe constraints.

Chapter 4 describes how to programmatically validate a JavaBean.

Chapter 5 describes how the metadata query API works.

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

JPA, JSF and EE integration are present at Appendix D, Appendix E and Appendix F for archive purposes, one should refer to the respective specifications for the up-to-date integration rules.

The changelog can be found at Appendix G.

1.4. How to comment

The expert group is eager to receive feedback from readers. Feel free to contact us or use jsr-303-comments@jcp.org.

2

Constraint Definition

Constraints are defined by the combination of a constraint annotation and a list of constraint validation implement-ations. The constraint annotation is applied on types, methods, fields or other constraint annotations in case of com-position.

Unless stated otherwise the default package name for the Bean Validation APIs is `javax.validation`.

2.1. 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 `javax.validation.Constraint`.

```
/**
 * Link between a constraint annotation and its constraint validation implementations.
 * <p/>
 * A given constraint annotation should be annotated by a <code>@Constraint</code>
 * annotation which refers to its list of constraint validation implementations.
 *
 * @author Emmanuel Bernard
 * @author Gavin King
 * @author Hardy Ferentschik
 */
@Documented
@Target({ ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface Constraint {
    /**
     * <code>ConstraintValidator</code> classes must reference distinct target types.
     * If two <code>ConstraintValidator</code> refer to the same type,
     * an exception will occur.
     *
     * @return array of ConstraintValidator classes implementing the constraint
     */
    public Class<? extends ConstraintValidator<?, ?>>[] validatedBy();
}
```

Constraint annotations can target any of the following `ElementType`s:

- `FIELD` for constrained attributes

- `METHOD` for constrained getters

- `TYPE` for constrained beans

- `ANNOTATION_TYPE` for constraints composing other constraints

While other `ElementType`s are not forbidden, the provider does not have to recognize and process constraints placed on such types. Built-in types do support `PARAMETER` and `CONSTRUCTOR` to allow Bean Validation provider specific extensions. It is considered good practice to follow the same approach for custom annotations.

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 Section 3.5.3) could lead to exceptions if the `ConstraintValidator` list leads to ambiguities.

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 Section 2.1.1).

2.1.1. 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` 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.

2.1.1.1. 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 Section 4.3 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 Bean Validation constraints follow this convention.

2.1.1.2. groups

Every constraint annotation must define a `groups` element that specifies the processing groups with which the constraint declaration is associated.

```
    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 Section 4.1.2 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.

2.1.1.3. payload

Constraint annotations must define a `payload` element that specifies the payload with which the the constraint declaration is associated.

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

The default value must be an empty array.

Each attachable payload extends `Payload`.

```
/**
 * Payload type that can be attached to a given
 * constraint declaration.
 * Payloads are typically used to carry on metadata information
 * consumed by a validation client.
 *
 * 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. Payloads are typically non-portable. Describing payloads as interface extensions as opposed to a string-based approach allows an easier and more type-safe approach.

One use case for payload shown in Example 2.1 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 2.1. 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 Section 4.2) or through the metadata API (see Section 5.5).

2.1.1.4. 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.

2.1.2. Examples

Example 2.2. Simple constraint definition

```
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 })
@Retention(RUNTIME)
public @interface OrderNumber {
    String message() default "{com.acme.constraint.OrderNumber.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Example 2.2 marks a String as a well-formed order number. The constraint Validator is implemented by `OrderNumberValidator`.

Example 2.3. 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.
 * Accept Numbers.
 */
@Documented
@Constraint(validatedBy = AudibleValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@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
    }
}
```

Example 2.3 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 2.4. 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 })
@Retention(RUNTIME)
public @interface Acceptable {
    int[] value();
    String message() default "{com.acme.constraint.Acceptable.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Example 2.4 defines a list of acceptable values expressed as an array: the `value` property must be specified when the constraint is applied.

2.2. 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 bean 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 Bean 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 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`.

Example 2.5. Multi-valued constraint definition

```
/**
 * Validate a zipcode for a given country
 * The only supported type is String
 */
@Documented
@Constraint(validatedBy = ZipCodeValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
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 })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        ZipCode[] value();
    }
}
```

Example 2.6. Multi-valued constraint declaration

```
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;
}
```

In this example, both constraints apply to the `zipcode` field but with different groups and with different error messages.

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.

2.3. 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 2.7. Composition is done by annotating the composed constraint

```
@Pattern(regexp="[0-9]*")
@Size(min=5, max=5)
@Constraint(validatedBy = FrenchZipcodeValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD, CONSTRUCTOR, PARAMETER})
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @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. Likewise, payload from the main constraint annotation is inherited by the composing annotations. Any `payload` definition on a composing annotation is ignored.

The property 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 list in the JavaDoc all the compatible types.

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 2.8. 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 })
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

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

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

The definition of `@ReportAsSingleViolation` is as follows.

```
/**
 * 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 is ignored.
 *
 * @author Emmanuel Bernard
 */
@Target({ ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface ReportAsSingleViolation {
}
```

More specifically, if a composed constraint is marked as `@ReportAsSingleViolation`, and if any of its composing constraint reports one or more violations, all reports from composing constraints are ignored and the error report corresponding to the composed constraint is generated.

Note

If a composing constraint fails and if the composed constraint is marked as `@ReportAsSingleViolation`, the Bean Validation provider is free to not process the other composing constraints for this composed constraint.

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

Example 2.9. 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 })
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @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 the `@OverridesAttribute` annotation or its multivalued equivalent `@OverridesAttribute.List`.

Example 2.10. 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 })
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @OverridesAttribute.List( {
        @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 })
    @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 Example 2.10,

```
@FrenchZipcode(size=9, sizeMessage="Zipcode should be of size {max}")
```

is equivalent to

```
@FrenchZipcode
```

if `@FrenchZipcode` is defined as

```
@Pattern(regexp="[0-9]*")
@Size(min=9, max=9, message="Zipcode should be of size {max}")
@Constraint(validatedBy = FrenchZipcodeValidator.class)
@Documented
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface FrenchZipcode {
    String message() default "Wrong zipcode";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @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 Section 2.2 is used. To select a specific composing constraint, `OverridesAttribute.constraintIndex` is used. It represents the constraint index in the `value` array. If `index` is undefined, the single constraint declaration is targeted.

Example 2.11. Use of constraintIndex in @OverridesAttribute

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

`@OverridesAttribute` definition is as follows:

```java
/**
 * Mark an attribute as overriding the attribute of a composing constraint.
 * Both attributes must share the same type.
 *
 * @author Emmanuel Bernard
 */
@Retention(RUNTIME)
@Target({ METHOD })
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</code>.
     *
     * @return name of constraint attribute overridden.
     */
    String name();

    /**
     * The index of the targeted constraint declaration when using
     * multiple constraints of the same type.
     * The index represents the index of the constraint in the value() array.
     *
     * 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 @OverridesAttribute annotations on the same element
     * @see javax.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 (infinitely recursive composition, wrong attribute overriding, a single attribute mapped to more than one source attribute 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 con-

straints defined by the specification. The composing constraints are exposed through the Bean Validation metadata API (Section 5.5). This metadata is particularly useful for third-party metadata consumers like persistence frameworks generating database schemas (such as Java Persistence) or presentation frameworks.

2.4. 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.

```
/**
 * Defines the logic to validate a given constraint A
 * for a given object type T.
 * Implementations must comply to the following restriction:
 * <ul>
 * <li>T must resolve to a non parameterized type</li>
 * <li>or generic parameters of T must be unbounded
 * wildcard types</li>
 * </ul>
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConstraintValidator<A extends Annotation, T> {
    /**
     * Initialize the validator in preparation for isValid 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.
     *
     * @param constraintAnnotation annotation instance for a given constraint declaration
     */
    void initialize(A constraintAnnotation);

    /**
     * Implement the validation logic.
     * The state of <code>value</code> must not be altered.
     *
     * 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 false if <code>value</code> 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. `<?>`)

Example 2.12 shows some examples of valid definitions.

Example 2.12. Valid ConstraintValidator definitions

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

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

//Collection uses generics and unbounded windcard type
public class SizeValidatorForCollection implements<Size, Collection<?>> {...}
```

And some invalid definitions in Example 2.13.

Example 2.13. Invalid ConstraintValidator definitions

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

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

Note

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

The life cycle of a constraint validation implementation instance is undefined. Compliant implementations are allowed to cache `ConstraintValidator` instances retrieved from the `ConstraintValidatorFactory`.

The `initialize` method is called by the Bean validation provider prior to any use of the constraint implementation.

The `isValid` method is evaluated by the Bean 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 Bean Validation provider (see the type matching algorithm described in Section 3.5.3).

If an exception occurs either in the `initialize` or `isValid` method, the runtime exception is wrapped into a `ValidationException` by the Bean 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.

```
/**
 * Provide contextual data and operation when applying a given constraint validator.
 *
 * At least one <code>ConstraintViolation</code> must be defined (either the default one,
 * of if the default <code>ConstraintViolation</code> is disabled, a custom one).
 *
 * @author Emmanuel Bernard
 */
public interface ConstraintValidatorContext {
    /**
     * Disable the default <code>ConstraintViolation</code> object generation (which
     * is using the message template declared on the constraint).
     * Useful to set a different violation message or generate a <code>ConstraintViolation</code>
     * based on a different property.
     */
    void disableDefaultConstraintViolation();

    /**
     * @return the current uninterpolated default message.
     */
    String getDefaultConstraintMessageTemplate();

    /**
     * Return an constraint violation builder building an violation report
     * allowing to optionally associate it to a sub path.
     * The violation message will be interpolated.
     * <p/>
     * To create the <code>ConstraintViolation</code>, one must call either one of
     * the #addConstraintViolation() methods available in one of the
     * interfaces of the fluent API.
     * If another method is called after #addConstraintViolation() on
     * <code>ConstraintViolationBuilder</code> or any of its associated nested interfaces
     * an <code>IllegalStateException</code> is raised.
     * <p/>
     * If <code>isValid<code> returns <code>false</code>, a <code>ConstraintViolation</code>
     * object will be built per ConstraintViolation report including the default one (unless
     * {@link #disableDefaultConstraintViolation()} has been called).
     * <p/>
     * <code>ConstraintViolation</code> objects generated from such a call
     * contain the same contextual information (root bean, path and so on) unless
     * the path has been overriden.
     * <p/>
     * To create a different <code>ConstraintViolation</code>, a new constraint violation builder
     * has to be retrieved from <code>ConstraintValidatorContext</code>
     *
     * Here are a few usage examples:
     * <pre>
     * {@code
     * // create new violation report with the default path the constraint is located on
     * context.buildConstraintViolationWithTemplate( "way too long" )
     *             .addConstraintViolation();
     *
     * // create new violation report in the "street" subnode of the default path
     * //the constraint is located on
     * context.buildConstraintViolationWithTemplate( "way too long" )
     *             .addNode( "street" )
```

```
      *                .addConstraintViolation();
      *
      * //create new violation report in the "addresses["home"].city.name" subnode
      * //of the default path the constraint is located on
      * context.buildConstraintViolationWithTemplate( "this detail is wrong" )
      *                .addNode( "addresses" )
      *                .addNode( "country" )
      *                    .inIterable().atKey( "home" )
      *                .addNode( "name" )
      *                .addConstraintViolation();
      * }
      * </pre>
      *
      * @param messageTemplate new uninterpolated constraint message.
      * @return Returns an constraint violation builder
      */
    ConstraintViolationBuilder buildConstraintViolationWithTemplate(String messageTemplate);

    /**
      * <code>ConstraintViolation</code> builder allowing to optionally associate
      * the violation report to a sub path.
      *
      * To create the <code>ConstraintViolation</code>, one must call either one of
      * the #addConstraintViolation() methods available in one of the
      * interfaces of the fluent API.
      * If another method is called after #addConstraintViolation() on
      * <code>ConstraintViolationBuilder</code> or any of its associated objects
      * an <code>IllegalStateException</code> is raised.
      *
      */
    interface ConstraintViolationBuilder {
        /**
          * Add a node to the path the <code>ConstraintViolation</code> will be associated to.
          *
          * <code>name</code> describes a single property. In particular,
          * dot (.) is not allowed.
          *
          * @param name property name
          * @return a builder representing node <code>name</code>
          */
        NodeBuilderDefinedContext addNode(String name);

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

        /**
          * Represent a node whose context is known
          * (ie index, key and isInIterable)
          */
        interface NodeBuilderDefinedContext {

            /**
              * Add a node to the path the <code>ConstraintViolation</code> will be associated to.
              *
              * <code>name</code> describes a single property. In particular,
              * dot (.) are not allowed.
              *
              * @param name property <code>name</code>
```

```
             * @return a builder representing this node
             */
            NodeBuilderCustomizableContext addNode(String name);

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

        /**
         * Represent a node whose context is
         * configurable (ie index, key and isInIterable)
         */
        interface NodeBuilderCustomizableContext {

            /**
             * Mark the node as being in an <code>Iterable</code> or a <code>Map</code>
             *
             * @return a builder representing iterable details
             */
            NodeContextBuilder inIterable();

            /**
             * Add a node to the path the <code>ConstraintViolation</code> will be associated to.
             *
             * <code>name</code> describes a single property. In particular,
             * dot (.) are not allowed.
             *
             * @param name property <code>name</code>
             * @return a builder representing this node
             */
            NodeBuilderCustomizableContext addNode(String name);

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

        /**
         * Represent refinement choices for a node which is
         * in an <code>Iterator<code> or <code>Map</code>.
         * If the iterator is an indexed collection or a map,
         * the index or the key should be set.
         */
        interface NodeContextBuilder {

            /**
             * Define the key the object is into the <code>Map</code>
             *
             * @param key map key
```

```
             * @return a builder representing the current node
             */
            NodeBuilderDefinedContext atKey(Object key);

            /**
             * Define the index the object is into the <code>List</code> or array
             *
             * @param index index
             * @return a builder representing the current node
             */
            NodeBuilderDefinedContext atIndex(Integer index);

            /**
             * Add a node to the path the <code>ConstraintViolation</code> will be associated to.
             *
             * <code>name</code> describes a single property. In particular,
             * dot (.) are not allowed.
             *
             * @param name property <code>name</code>
             * @return a builder representing this node
             */
            NodeBuilderCustomizableContext addNode(String name);

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

The `ConstraintValidatorContext` interface 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, attribute).

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 or property hosting the constraint (see Section 4.2 for more information). You can point it to a subpath of this default path by using the constraint violation builder fluent API.

Example 2.14 shows a few examples.

Example 2.14. Using the fluent API to build custom constraint violations

```
//default path
context.buildConstraintViolationWithTemplate( "this detail is wrong" )
            .addConstraintViolation();
```

```
//default path + "street"
context.buildConstraintViolationWithTemplate( "this detail is wrong" )
           .addNode( "street" )
           .addConstraintViolation();

//default path + "addresses["home"].country.name"
context.buildConstraintViolationWithTemplate( "this detail is wrong" )
           .addNode( "addresses" )
           .addNode( "country" )
               .inIterable().atKey( "home" )
           .addNode( "name" )
           .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.

2.4.1. Example

Example 2.15. ConstraintValidator implementation

```
/**
 * Check that a text is within the authorized syntax
 */
public class SyntaxValidator implements ConstraintValidator<Syntax, String> {
    private Set<Format> allowedFormats;

    /**
     * Configure the constraint validator based on the elements
     * specified at the time it was defined.
     * @param constraint the constraint definition
     */
    public void initialize(Syntax constraint) {
        allowedFormats = new HashSet( Arrays.asList( constraint.value() ) );
    }

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

        return allowedFormats.size() == 0
            || (! Collections.disjoint( guessFormat(value), allowedFormats ) );
    }

    Set<Format> guessFormats(String text) { ... }
}
```

This `ConstraintValidator` checks that a text is within the accepted syntax. It also demonstrates an interesting best practice: return `true` on a null parameter

The next example shows how to use `ConstraintValidatorContext`.

Example 2.16. Use of ConstraintValidatorContext

```
/**
 * Check that a text is within the authorized syntax
 * Error messages are using either key:
 *  - com.acme.constraint.Syntax.unknown if no particular syntax is detected
 *  - com.acme.constraint.Syntax.unauthorized if the syntax is not allowed
 */
public class FineGrainedSyntaxValidator implements ConstraintValidator<Syntax, String> {
    private Set<Format> allowedFormats;

    /**
     * Configure the constraint validator based on the elements
     * specified at the time it was defined.
     * @param constraint the constraint definition
     */
    public void initialize(Syntax constraint) {
        allowedFormats = new HashSet( Arrays.asList( constraint.value() ) );
    }

    /**
     * Validate a specified value.
     * returns false if the specified value does not conform to the definition
     */
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if ( value == null ) return true;
        Set<Format> guessedFormats = guessFormat(value);

        context.disableDefaultConstraintViolation();
        if ( guessedFormats.size() == 0 ) {
            String unknown = "{com.acme.constraint.Syntax.unknown}";
            context.buildConstraintViolationWithTemplate(unknown)
                        .addConstraintViolation();
        }
        else {
            String unauthorized = "{com.acme.constraint.Syntax.unauthorized}";
            context.buildConstraintViolationWithTemplate(unauthorized)
                        .addConstraintViolation();
        }

        return allowedFormats.size() == 0
            || (! Collections.disjoint( guessFormat(value), allowedFormats ) ));
    }

    Set<Format> guessFormats(String text) { ... }
}
```

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.

2.5. The ConstraintValidatorFactory

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

```
/**
 * Instantiate a <code>ConstraintValidator</code> instance based off its class.
 * The <code>ConstraintValidatorFactory</code> 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.
     *
     * @return A constraint validator instance of the specified class.
     */
    <T extends ConstraintValidator<?,?>> T getInstance(Class<T> key);
}
```

The default `ConstraintValidatorFactory` provided by the Bean 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. Any constraint implementation relying on `ConstraintValidatorFactory` behaviors specific to an implementation (dependency injection, no no-arg constructor and so on) are not considered portable, hence great care should be given before walking that path.

`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.

Constraint declaration and validation process

The Bean 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.

3.1. Requirements on classes to be validated

Objects that are to be validated 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.

- Static fields and static methods are excluded from validation.

- Constraints can be applied to interfaces and superclasses.

The target of an annotation definition can be a field, property, or type, provided that:

- the constraint definition supports the specified target (`java.lang.annotation.Target`)

- the one of the `ConstraintValidator`s declared on the constraint supports the declared type of the target (see Section 3.5.3).

3.1.1. 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.

3.1.2. 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).

Java Persistence and Bean Validation

For maximum portability, persistent properties hosting Bean Validation constraints should use the same ac-

cess strategy used in Java Persistence. In other words, place your Bean Validation constraint annotations on the same element (field or getter) as your Java 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 bean validation provider accesses the instance variable directly. When using the property access strategy, the bean 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, `get<Property-name>`. 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. Constraints on non getter methods are not supported.

3.1.3. Graph validation

In addition to supporting instance validation, validation of graphs of object is also supported. The result of a graph validation is returned as a unified set of constraint violations.

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 `@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 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.

The `@Valid` annotation is applied recursively. A conforming implementation avoids infinite loops according to the rules described in Section 3.5.1.

3.2. Constraint declaration

Constraint declarations are placed on classes or interfaces primarily through annotations. A constraint annotation (see Section 2.1), 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`.

3.3. 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 Bean Validation provider. Rules are formally described in Section 3.4.5.

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.

3.4. 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 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 3.1. 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 3.2. 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 group is evaluated on the object graph. In Example 3.2, `@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

```
package javax.validation.groups;

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

3.4.1. Group inheritance

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

Example 3.3. 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 3.4. 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`.

3.4.2. 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 Section 3.5 ; 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 constraint is evaluated only if another set of constraint 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.

```
/**
 * Define a group sequence
 * The interface hosting <code>@GroupSequence</code> is representing
 * the group sequence.
 * When hosted on a class, represents the <code>Default</code> group
 * for that class.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
@Target({ TYPE })
@Retention(RUNTIME)
public @interface GroupSequence {
    Class<?>[] value();
}
```

Here is a usage example

Example 3.5. 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 Example 3.5, 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 Section 3.5 for more information.

3.4.3. Redefining the Default group for a class

In Example 3.5, 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 3.6. 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 Example 3.6, 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.

3.4.4. 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 3.7. Example of interface / group hosting constraints

```java
/**
 * 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.

3.4.5. Formal group definitions

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

For every class `X`:

A. 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

B. The group `X` contains the following constraints:

   group `X` is a group used on sequences redefining the default group on a class (see Section 3.4.3)

   I. 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`

   II. 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.

   III. 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

C. 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`.

   I. every constraint in the group `X`

   II. 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`

D. 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 Section 3.4.3)

- the `@GroupSequence` annotation must declare the group x

E. For every interface z, the group z contains the following constraints:

this rule defines how non `Default` groups are defined

I. 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 Section 3.4.4)

II. 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 Section 3.4.1)

III. 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

IV. 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

V. 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

F. 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 Section 3.4.2)

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.

3.5. Validation routine

For a given group to validate, 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 Section 3.5.1) 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 Section 3.5.1) 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 Section 3.5.1) as part of a previous group match.

- for all *reachable* and *cascadable* associations, execute all cascading validations (see Section 3.5.1) including the ones expressed on interfaces and superclasses (see Section 3.4.5)

Reachable fields, getters and associations as well as cascadable associations are defined in Section 3.5.2.

Note that this implies that a given validation constraint will not be processed more than once per validation.

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 Section 3.5.3).

- execute the `isValid` operation (from the constraint validation implementation) on the appropriate data (see Section 2.4)

- if `isValid` returns true, continue to the next constraint,

- if `isValid` returns false, the Bean Validation provider populates `ConstraintViolation` object(s) according to the rules defined in Section 2.4 and appends these objects to the list of constraint violations.

3.5.1. Object graph validation

The `@Valid` annotation on a given association (i.e. object reference or collection, array, `Iterable` of objects), dictates the Bean Validator implementation to apply recursively the Bean 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 Bean 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 Example 3.8 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 Example 3.8 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 3.8. 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 Section 4.2).

`@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 3.9. 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 3.10. Class Car with redefined default group

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

```
  @AssertTruegroups = Later.class)
  Boolean roadWorthy;

  // setter/getters
}
```

Example 3.11. Defining a group sequence

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

Example 3.12. 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.

3.5.2. Traversable property

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

Bean Validation offers a way to control which property can and cannot be accessed via the `TraversableResolv-`

er.isReachable() contract.

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

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

    /**
     * Determine if the Bean Validation provider is allowed to cascade validation on
     * the bean instance returned by the property value
     * marked as <code>@Valid</code>.
     * Note that this method is called only if <code>isReachable</code> returns true
     * for the same set of arguments and if the property is marked as <code>@Valid</code>
     *
     * @param traversableObject object hosting <code>traversableProperty</code> or null
     *                          if <code>validateValue</code> is called
     * @param traversableProperty the traversable property.
     * @param rootBeanType type of the root object passed to the Validator.
     * @param pathToTraversableObject path from the root object to
     *        <code>traversableObject</code>
     *        (using the path specification defined by Bean Validator).
     * @param elementType either <code>FIELD</code> or <code>METHOD</code>.
     *
     * @return <code>true</code> if the Bean Validation provider is allowed to
     *          cascade validation, <code>false</code> otherwise.
     */
    boolean isCascadable(Object traversableObject,
                         Path.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 `validate-Value` call.

`traversableProperty` is the `Node` representing the property hosted by the `traversableObject` being considered for traversal. The name of a property is defined in Section 3.1.2.

`rootBeanType` is the class of the root being validated (and passed to the `validate` method).

`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 Section 4.2 (`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 Bean 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 example assumes the object graph defined in Example 3.13 and assumes the validation operation is applied on an address object.

Example 3.13. Definitions used in the example

```java
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 Bean Validation provider is about to check constraints of `ISO3Code`, it calls the `TraversableResolv-`

er.isReachable() instance to ensure that the ISO3Code property is reachable with the following parameter values:

- traversableObject: country. The instance returned by address.getCountry().

- traversableProperty: a Node whose name is "ISO3Code". The name of the property of traversableObject being verified.

- rootBeanType: Address.class. The type of the root object being validated.

- pathtoTraversableObject: a Path containing a single Node 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 Bean Validation provider is about to cascade validation on country (Address object), it calls the TraversableResolver.isReachable() instance 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 Node whose name is "country". The name of the property of traversableObject being verified.

- rootBeanType: Address.class. The type of the root object being validated.

- pathtoTraversableObject: a Path containing a single Node whose name is null.

- elementType: ElementType.FIELD. The country property is annotated on its field.

Example 3.14. Java 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;
    }
}
```

The traversable resolver used by default in a Bean Validation behaves as followed:

- if Java Persistence is available in the runtime environment, a property is considered reachable if Java Persistence considers the property as loaded. A typical implementation will use `Persistence.getPersistenceUtil().isLoaded(Object, String)` to implement such contract.

- if Java Persistence is not available in the runtime environment, all properties are considered reachable.

- all properties are considered cascadable.

See Section 4.4 to know how to pass a custom `TraversableResolver`.

3.5.3. 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.

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 getter, the return type of the getter is the targeted type. In other words, the resolution algorithm considers the type as defined in the method signature and not the runtime type of the value.

The rules written below describe formally the following statement: the `ConstraintValidator` chosen to validate a declared type `T` is the one 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 constraint A placed on a target declaring the type `T`, the following resolution rules apply:

- Primitive types are considered equivalent to their respective primitive wrapper class. Likewise, arrays of primitive types are considered equivalent to arrays of their wrapper classes.

- A `ConstraintValidator<A, U>` is said to be *compliant* with `T` if `T` is a subtype of `U` (according to the Java Language Specification 3rd edition chapter 4.10 Subtyping [1]). Note that `T` is a subtype of `U` if `T` = `U`.

- If no `ConstraintValidator` compliant with `T` is found amongst the `ConstraintValidator`s listed by the constraint A, a `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 3rd edition chapter 4.10 Subtyping [2]).

- 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, a `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 a Java 6 annotation processor can

[1] http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.10
[2] http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.10

apply these rules and prevent compilation in case of ambiguity. The specification encourages Bean 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 Example 3.15:

Example 3.15. 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 Table 3.1 occur.

Table 3.1. 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() { ... }`	`UnexpectedTypeException`: none of the `ConstraintValidator` types are supertypes of `String`.

3.6. Examples

The first example demonstrates how beans, fields and getters are annotated to express some constraints.

Example 3.16. 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 implementation.

- `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 3.17. 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`, `zip-Code`, `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 `@NotEmpty` is defined as such

```
package com.acme.constraint;

@Documented
@NotNull
@Size(min=1)
@ReportAsSingleViolation
@Constraint(validatedBy = NotEmpty.NotEmptyValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface NotEmpty {
    String message() default "{com.acme.constraint.NotEmpty.message}"
    Class<?> groups() default {};
    Class<? extends Payload>[] payload() default {};

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

    class NotEmptyValidator implements ConstraintValidator<NotEmpty, String> {
        public void initialize(NotEmpty constraintAnnotation) {}

        public boolean isValid(String value, ConstraintValidatorContext context) {
            return true;
        }
    }
}
```

The third example demonstrates superclass, inheritance and composite constraints.

Example 3.18. Use inheritance, constraints on superclasses and composite constraints

```java
public interface Person {
    @NotEmpty
    String getFirstName();

    String getMiddleName();

    @NotEmpty
    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:

- `@NotEmpty`, `@NotNull` and `@Size(min=1)` on `firstName`

- `@NotEmpty`, `@NotNull` and `@Size(min=1)` on `lastName`

- `@NotNull` on `customerId`, `@Password` on `password`

- `@CreditCard` on `guestCreditCardNumber`

When validating `CommonGuest`, the following constraints are processed:

- `@NotEmpty`, `@NotNull` and `@Size(min=1)` on `firstName`

- `@NotEmpty`, `@NotNull` and `@Size(min=1)` on `lastName`

- `@NotNull` on `customerId`, `@Password` on `password`

The fourth example demonstrates the influence of group sequence.

Example 3.19. Use groups and group sequence to define constraint ordering

```
@GroupSequence({First.class, Second.class, Last.class})
public interface Complete {}

public class Book {
    @NotEmpty(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 {
    @NotEmpty(groups=Last.class)
    private String firstName;

    @NotEmpty(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 `@NotEmpty`) 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 `@NotEmpty`) 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.

Validation APIs

The default package for the Bean Validation APIs is `javax.validation`

4.1. Validator API

The main Bean Validation API is the `javax.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 are thread-safe.

```
/**
 * Validate bean instances. Implementations of this interface must be thread-safe.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface Validator {
    /**
     * Validates all constraints on <code>object</code>.
     *
     * @param object object to validate
     * @param groups group or list of groups targeted for validation
     *               (default to {@link javax.validation.groups.Default})
     *
     * @return constraint violations or an empty Set if none
     *
     * @throws IllegalArgumentException if object is null
     *                                  or if 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</code>
     * named <code>propertyName</code>.
     *
     * @param object object to validate
     * @param propertyName property to validate (ie field and getter constraints)
     * @param groups group or list of groups targeted for validation
     *               (default to {@link javax.validation.groups.Default})
     *
     * @return constraint violations or an empty Set if none
     *
     * @throws IllegalArgumentException if <code>object</code> is null,
     *             if <code>propertyName</code> null, empty or not a valid object property
     *             or if 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</code>
     * of the class <code>beanType</code> would the property value be <code>value</code>
     * <p/>
     * <code>ConstraintViolation</code> objects return 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 group or list of groups targeted for validation
     *               (default to {@link javax.validation.groups.Default})
     *
     * @return constraint violations or an empty Set if none
     *
     * @throws IllegalArgumentException if <code>beanType</code> is null,
     *             if <code>propertyName</code> null, empty or not a valid object property
     *             or if 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);

    /**
     * Return the descriptor object describing bean constraints.
     * The returned object (and associated objects including
     * <code>ConstraintDescriptor<code>s) are immutable.
     *
     * @param clazz class or interface type evaluated
     *
     * @return the bean descriptor for the specified class.
     *
     * @throws IllegalArgumentException if clazz is null
     * @throws ValidationException if a non recoverable error happens
     *                             during the metadata discovery or if some
     *                             constraints are invalid.
     */
    BeanDescriptor getConstraintsForClass(Class<?> clazz);

    /**
     * Return an instance of the specified type allowing access to
     * provider-specific APIs.  If the Bean Validation provider
     * implementation does not support the specified class,
     * <code>ValidationException</code> is thrown.
     *
     * @param type the class 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);
}
```

getConstraintsForClass is described in Chapter 5.

unwrap is provided as a way to access objects of a given type specific to a Bean Validation provider typically as a complement to the Validator contract. Using this method makes your code non portable.

Example 4.1. Using unwrap to access a provider specific contract

```
//if using the ACME provider
ACMEValidator acmeValidator = factory.unwrap(ACMEValidator.class);
acmeValidator.setSpecificConfiguration(...);
```

4.1.1. 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 Section 3.5. 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. The property name is the JavaBeans property name (as defined by the JavaBeans `Introspector` class). This method implements the logic described in Section 3.5 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*. This method implements the logic described in Section 3.5 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.

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 Chapter 8 or the relative sections for more information).

4.1.1.1. 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` object because the value "Paris" for `city` would not raise any constraint failure.

```
validator.validateValue("city", "Paris").size() == 0
```

4.1.2. groups

Groups allow you to restrict the set of constraints applied during validation. Groups targeted are passed as paramet-

ers to the `validate`, `validateProperty` and `validateValue` methods. All constraints belonging to the targeted group are applied during the Section 3.5. If no group is passed, the `Default` group is assumed. Section 2.1.1.2 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.

4.1.2.1. Examples

```
/** Validates a minimal set of constraints */
public interface Minimal {}

public class Address {

    @NotEmpty(groups = Minimal.class)
    @Size(max=50)
    private String street1;

    @NotEmpty
    private String city;

    @NotEmpty(groups = {Minimal.class, Default.class})
    private String zipCode;
    ...
}
```

In the previous example, `@NotEmpty` (and it's composing constraints) on `street1` applies to the group `Minimal`, `@Size` on `street1` applies to the group `Default` and `@NotEmpty` (and it's composing constraints) on `zipCode` applies to the groups `Default` and `Minimal`.

```
validator.validate(address);
```

validates the group `Default` (implicitly) and applies `@Size` on `street1`, `@NotEmpty` (and its composing constraints) on `city`, `@NotEmpty` (and its composing constraints) on `zipCode`. Particularly, `@NotEmpty` (and its composing constraints) on `street1` are not applied.

```
validator.validate(address, Minimal.class);
```

applies `@NotEmpty` (and its composing constraints) on `street1` and `@NotEmpty` (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 `@NotEmpty` (and its composing constraints) and `@Size` on `street1`, `@NotEmpty` (and its composing constraints) on `city`, `@NotEmpty` (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 {
    @NotEmpty(groups = Minimal.class)
    @Size(max=50, groups=FirstStep.class)
    private String street1;
```

```
    @NotEmpty(groups=SecondStep.class)
    private String city;

    @NotEmpty(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 `@NotEmpty` (and it's composing constraints) and `@Size` from `street1` and `@NotEmpty` (and it's composing constraints) from `zipCode`. If `@Size` from `street1` does not generate a failure, then `@NotEmpty` (and it's composing constraints) from `city` will be processed as part of `SecondStep`. Note that `@NotEmpty` (and it's composing constraints) from `zipCode` are not reprocessed as they have already been processed before.

When running:

```
validator.validate(address, Total.class, SecondStep.class);
```

`@NotEmpty` (and it's composing constraints) from `city` and `@NotEmpty` (and it's composing constraints) from `zip-Code` 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.

4.2. ConstraintViolation

`ConstraintViolation` is the class describing a single constraint failure. A set of `ConstraintViolation` is returned for an object validation.

```
/**
 * Describe a constraint violation. This object exposes the constraint
 * violation context as well as the message describing the violation.
 *
 * @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();

    /**
     * @return The root bean being validated. Null when returned by
     *         {@link javax.validation.Validator#validateValue(Class, String, Object, Class[])}
     */
    T getRootBean();

    /**
     * @return The class of the root bean being validated
     */
    Class<T> getRootBeanClass();

    /**
     * If a bean constraint, the bean instance the constraint is applied on
     * If a property constraint, the bean instance hosting the property the
     * constraint is applied on
     *
     * @return the leaf bean the constraint is applied on. Null when returned by
     *         {@link javax.validation.Validator#validateValue(Class, String, Object, Class[])}
     */
    Object getLeafBean();

    /**
     * @return the property path to the value from {@code rootBean}.
     */
    Path getPropertyPath();

    /**
     * @return the value failing to pass the constraint.
     */
    Object getInvalidValue();

    /**
     * Constraint metadata reported to fail.
     * The returned instance is immutable.
     *
     * @return constraint metadata
     */
    ConstraintDescriptor<?> getConstraintDescriptor();
}
```

The `getMessage` method returns the interpolated (localized) message for the failing constraint (see Section 4.3 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.

The `getInvalidValue` method returns the value (field, property or validated object) being passed to `isValid`.

`getConstraintDescriptor` provides access to the failing constraint metadata (see Section 5.5).

The `getPropertyPath` returns the Path object representing the navigation path from the root object to the failing

object.

```
/**
 * Represent the navigation path from an object to another
 * in an object graph.
 * Each path element is represented by a <code>Node</code>.
 *
 * The path corresponds to the succession of nodes
 * in the order they are returned by the <code>Iterator</code>
 *
 * @author Emmanuel Bernard
 */
public interface Path extends Iterable<Path.Node> {

    /**
     * Represents an element of a navigation path
     */
    interface Node {
        /**
         * Property name the node represents
         * or null if representing an entity on the leaf node
         * (in particular the node in a <code>Path</code> representing
         * the root object has its name null).
         *
         * @return property name the node represents
         */
        String getName();

        /**
         * @return true if the node represents an object contained in an Iterable
         * or in a Map.
         */
        boolean isInIterable();

        /**
         * @return The index the node is placed in if contained
         * in an array or List. Null otherwise.
         */
        Integer getIndex();

        /**
         * @return The key the node is placed in if contained
         * in a Map. Null otherwise.
         */
        Object getKey();
    }
}
```

`Path` is made of `Node`s and is built according to the following rules:

* if the failing object is the root object, a `Node` with name set to null is added to the `Path`.

* When an association is traversed:

  * a `Node` object whose `name` equals the name of the association property (field name or Java Bean property name) is added to `Path`

  * if the association is a `List` or an array, the following `Node` object added contains the index value in `getIndex`.

  * if the association 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 property level constraint (field and getter)

  - a `Node` object is added to `Path` whose `name` equals the name of the property (field name or Java Bean property name)

  - the property path is considered complete

- For a class level constraint:

  - a `Node` object is added to `Path` whose `name` is null

  - the property path is considered complete

Note

A given `Node` object derives its `inIterable`, `key` and `index` properties from the previous association traversed.

Note

From `rootBean` and `propertyPath`, it is possible to rebuild the context of the failure

Assuming the following object definitions and when book is validated:

```
@SecurityChecking
public class Author {
    private String firstName;

    @NotEmpty(message="lastname must not be null")
    private String lastName;

    @Size(max=30)
    private String company;
    ...
}

@AvailableInStore(groups={Availability.class})
public class Book {
    @NotEmpty(groups={FirstLevelCheck.class, Default.class})
    private String title;

    @Valid
    @NotNull
    private List<Author> authors;

    @Valid
    private Map<String, Review> reviewsPerSource;

    @Valid
    private Review pickedReview;
    ...
}
```

```
public class Review {
    @Min(0) private int rating;
    ...
}
```

propertyPath evaluations are described in Table 4.1:

Table 4.1. propertyPath examples

Constraint	propertyPath
@AvailableInStore on Book	Node(name=null,inIterable=false, index=null, key=null)
@NotEmpty on Book.title	Node(name=title,inIterable=false, index=null, key=null)
@NotNull on Book.authors	Node(name=authors,inIterable=false, index=null, key=null)
@SecurityChecking on the fourth author, Author	Node(name=authors,inIterable=false, index=null, key=null) Node(name=null,inIterable=true, index=3, key=null)
@Size on the fourth author, Author.lastname	Node(name=authors,inIterable=false, index=null, key=null) Node(name=lastname,inIterable=true, index=4, key=null)
@NotEmpty on the first author, Author.company	Node(name=authors,inIterable=false, index=null, key=null) Node(name=company,inIterable=true, index=0, key=null)
@Min on the review associated to Consumer Report, Review.rating	Node(name=reviewsPerSource,inIterable=false, index=null, key=null) Node(name=rating,inIterable=true, index=null, key="Consumer Report")
@Min on the picked review, Review.rating	Node(name=pickedReview,inIterable=false, index=null, key=null) Node(name=rating,inIterable=false, index=null,

Constraint	propertyPath
	key=null)

Note

Bean 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`.

4.2.1. Examples

These examples assume the following definition of `@NotEmpty`.

```
package com.acme.constraint;

@Documented
@NotNull
@Size(min=1)
@ReportAsSingleViolation
@Constraint(validatedBy = NotEmpty.NotEmptyValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface NotEmpty {
    String message() default "{com.acme.constraint.NotEmpty.message}"
    Class<?> groups() default {};
    Class<? extends Payload>[] payload() default {};

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

    class NotEmptyValidator implements ConstraintValidator<NotEmpty, String> {
        public void initialize(NotEmpty constraintAnnotation) {}

        public boolean isValid(String value, ConstraintValidatorContext context) {
            return true;
        }
    }
}
```

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 void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCompany() {
        return company;
    }

    public void setCompany(String company) {
        this.company = company;
    }
}

public class Book {
    @NotEmpty(groups={FirstLevelCheck.class, Default.class})
    private String title;

    @Valid
    @NotNull
    private Author author;

    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;
    }
}

Author author = new Author();
author.setCompany("ACME");
Book book = new Book();
book.setTitle("");
book.setAuthor(author);

Set<ConstraintViolation> constraintViolations = validator.validate(book);
```

ConstraintViolations is a set of size 2. One of the entries represents the failure of @NotEmpty (or more precisely @Size(min=1) a composing constraint of @NotEmpty) on the title property.

The ConstraintViolation object for this failure passes the following assertions:

```
//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();
assert "title".equals( nodeIter.next().getName() );
assert false == nodeIter.hasNext();
```

The second failure, `@NotEmpty` (or more precisely `@NotNull` a composing constraint of `@NotEmpty`) on the author's `lastname`, will produce the `ConstraintViolation` object satisfying the following assertions:

```
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();
assert "author".equals( nodeIter.next().getName() );
assert "lastName".equals( nodeIter.next().getName() );
assert false == nodeIter.hasNext();
```

4.3. Message interpolation

A message interpolator is responsible for transforming the message string from the constraint into a human readable error message.

4.3.1. Default message interpolation

A conforming implementation includes a default message interpolator. This message interpolator shall use the algorithm defined here to interpolate message descriptors into human-readable messages.

Each constraint defines a message descriptor via its `message` property. Every constraint definition shall define a default message descriptor for that constraint. Messages can be overridden at declaration time in constraints by setting the `message` property on the constraint.

The message descriptor is a string literal and may contain one or more message parameters. Message parameters are string literals enclosed in braces. The following character escaping 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

Example 4.2. Message using parameters

```
Value must be between {min} and {max}
```

4.3.1.1. 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 below). 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 Bean Validation provider's built-in `ResourceBundle` using the defined locale (see below). 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.

The defined locale is as followed:

- if the locale is passed to the interpolator method `interpolate(String, Context, Locale)`, this `Locale` instance is used.

- otherwise, the default `Locale` as provided by `Locale.getDefault()` is used.

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.

4.3.2. 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.

```
/**
 * Interpolate a given constraint violation message.
 * Implementations should be as tolerant as possible on syntax errors.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface MessageInterpolator {
    /**
     * Interpolate the message template based on the contraint validation context.
     * The locale is defaulted according to the <code>MessageInterpolator</code>
     * 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);

    /**
     * Interpolate the message template based on the contraint validation context.
     * The <code>Locale</code> 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 ConstraintDescriptor corresponding to the constraint being validated
         */
        ConstraintDescriptor<?> getConstraintDescriptor();

        /**
         * @return value being validated
         */
        Object getValidatedValue();
    }
}
```

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 is the `ConstraintDescriptor` object representing the metadata of the failing constraint (see Chapter 5).

getValidatedValue is the value being validated.

`MessageInterpolator.interpolate(String, Context)` is invoked for each constraint violation report generated. The default `Locale` 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 Example 4.3).

A message interpolator implementation shall 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 Bean Validation

default `MessageInterpolator` to ensure standard Bean 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`.

4.3.3. Examples

These examples describe message interpolation based on the default message interpolator's built-in messages (see Appendix B), and the `ValidationMessages.properties` file shown in table Table 4.2. The current locale is assumed English.

```
//ValidationMessages.properties
myapp.creditcard.error=credit card number not valid
```

Table 4.2. 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`. `Locale` aware `MessageInterpolator`. See Section 4.4 for more details on the APIs.

Example 4.3. Use MessageInterpolator to use a specific Locale value

```
/**
 * delegates to a MessageInterpolator implementation but enforce 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;
    }

    /**
     * enforece the locale passed to the interpolator
     */
    public String interpolate(String message,
                              Context context) {
        return defaultInterpolator.interpolate(message,
                                               context,
```

```
                                              this.defaultLocale);
    }

    // no real use, implemented for completeness
    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 4.4. Contextual container possible `MessageInterpolator` implementation

```
public class ContextualMessageInterpolator implements MessageInterpolator {
    private final MessageInterpolator delegate;

    public ContextualMessageInterpolator(MessageInterpolator delegate) {
        this.delegate = delegate;
    }

    public String interpolate(String message, Context context) {
        Locale locale = Container.getManager().getUserLocale();
        return this.delegate.interpolate(
                        message, context, locale );
    }

    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();
```

4.4. 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 Bean 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 Java EE 6 and other containers

- type safety

The main artifacts involved in the bootstrap process are:

- `Validation`: API entry point. Lets you optionally define the Bean 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 Bean Validation provider implementation.

- `ValidationProviderResolver`: returns a list of all Bean 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 Bean Validation providers. This sub interface typically hosts provider specific configurations.

- `ValidatorFactory`: result of the bootstrap process. Build `Validator` instances from a given Bean Validation provider.

- `META-INF/validation.xml`: a configuration file, Bean 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.

4.4.1. Examples

The most simple approach is to initialize the default Bean Validation provider or the one defined in the XML configuration file. The `ValidatorFactory` is then ready to provide `Validator` instances.

Example 4.5. Simple Bean Validation bootstrap sequence

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

//cache the factory somewhere
Validator validator = factory.getValidator();
```

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 Bean Validation implementation documentation for more accurate details.

The second example shows how a container can customize some Bean Validator resource handling to match its own behavior.

Example 4.6. Customize message resolution, traversable resolver and constraint Validator factory implementation

```
//some customization from a container
ValidatorFactory factory = Validation
        .byDefaultProvider().configure()
            .messageInterpolator( new ContainerMessageInterpolator() )
            .constraintValidatorFactory( new ContainerComponentConstraintValidatorFactory() )
            .traversableResolver( new JPAAwareTraversableResolver() )
            .buildValidatorFactory();

//cache the factory somewhere
Validator validator = factory.getValidator();
```

The third example shows how to bootstrap Bean Validation in an environment not following the traditional Java classloader strategies (such as tools or alternative service containers like OSGi). They can provider some alternative provider resolution strategy to discover Bean Validation providers.

Example 4.7. Customize the Bean Validation provider resolution mechanism

```
//osgi environment
ValidatorFactory factory = Validation
        .byDefaultProvider()
            .providerResolver( new OSGiServiceDiscoverer() )
            .configure()
                .buildValidatorFactory();

//cache the factory somewhere
Validator validator = factory.getValidator();
```

The next example shows how a client can choose a specific Bean Validation provider and configure provider specific properties programmatically in a type-safe way.

Example 4.8. 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 4.9. 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.

4.4.2. ValidatorFactory

`ValidatorFactory` objects build and provide initialized instances of `Validator` to Bean Validation clients. Each `Validator` instance is configured for a given context (message interpolator, traversable resolver). 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` implementations must be thread-safe. `ValidatorFactory` implementations can cache `Validator` instances if needed.

Example 4.10. ValidatorFactory interface

```
/**
 * Factory returning initialized <code>Validator</code> instances.
 * Implementations are thread-safe
 * This object is typically cached and reused.
 *
 * @author Emmanuel Bernard
 */
public interface ValidatorFactory {
    /**
     * Returns an initialized <code>Validator</code> 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.
     * </p>
     * @return an initialized <code>Validator</code> instance
     */
    Validator getValidator();

    /**
     * Defines a new validator context and return a <code>Validator</code>
     * compliant this new context.
     *
     * @return a <code>ValidatorContext</code>.
     */
    ValidatorContext usingContext();

    /**
     * Returns the <code>MessageInterpolator</code> instance configured at
     * initialization time for the <code>ValidatorFactory<code>.
     * This is the instance used by #getValidator().
     *
     * @return MessageInterpolator instance.
     */
    MessageInterpolator getMessageInterpolator();

    /**
     * Returns the <code>TraversableResolver</code> instance configured
     * at initialization time for the <code>ValidatorFactory<code>.
     * This is the instance used by #getValidator().
     *
     * @return TraversableResolver instance.
     */
    TraversableResolver getTraversableResolver();

    /**
     * Returns the <code>ConstraintValidatorFactory</code> instance
     * configured at initialization time for the
     * <code>ValidatorFactory<code>.
```

```
     * This is the instance used by #getValidator().
     *
     * @return ConstraintValidatorFactory instance.
     */
    ConstraintValidatorFactory getConstraintValidatorFactory();

    /**
     * Return an instance of the specified type allowing access to
     * provider-specific APIs. If the Bean Validation provider
     * implementation does not support the specified class,
     * <code>ValidationException,</code> is thrown.
     *
     * @param type  the class 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);
}
```

A `ValidatorFactory` is provided by a `Configuration` object.

`unwrap` is provided as a way to access objects of a given type specific to a Bean Validation provider typically as a complement to the `ValidatorFactory` contract. Using this method makes your code non portable.

Example 4.11. Using unwrap to access a provider specific contract

```
//if using the ACME provider
ACMEValidatorFactory acmeFactory = factory.unwrap(ACMEValidatorFactory.class);
acmeFactory.setSpecificConfiguration(...);
```

`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`.

`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` or the `ConstraintValidatorFactory`.

Example 4.12. ValidatorContext interface

```
/**
```

```
 * Represents the context that is used to create <code>Validator</code>
 * instances.
 *
 * A client may use methods of the <code>ValidatorContext</code> returned by
 * <code>ValidatorFactory#usingContext</code> to customize
 * the context used to create <code>Validator</code> instances
 * (for instance establish different message interpolators or
 * traversable resolvers).
 *
 * @author Emmanuel Bernard
 */
public interface ValidatorContext {
    /**
     * Defines the message interpolator implementation used by the
     * <code>Validator</code>.
     * If not set or if null is passed as a parameter,
     * the message interpolator of the <code>ValidatorFactory</code>
     * is used.
     *
     * @return self following the chaining method pattern
     */
    ValidatorContext messageInterpolator(MessageInterpolator messageInterpolator);

    /**
     * Defines the traversable resolver implementation used by the
     * <code>Validator</code>.
     * If not set or if null is passed as a parameter,
     * the traversable resolver of the <code>ValidatorFactory</code> is used.
     *
     * @return self following the chaining method pattern
     */
    ValidatorContext traversableResolver(TraversableResolver traversableResolver);

    /**
     * Defines the constraint validator factory implementation used by the
     * <code>Validator</code>.
     * If not set or if null is passed as a parameter,
     * the constraint validator factory of the <code>ValidatorFactory</code> is used.
     *
     * @return self following the chaining method pattern
     */
    ValidatorContext constraintValidatorFactory(ConstraintValidatorFactory factory);

    /**
     * @return an initialized <code>Validator</code> instance respecting the defined state.
     * Validator instances can be pooled and shared by the implementation.
     */
    Validator getValidator();
}
```

The `MessageInterpolator`, the `TraversableResolver` or the `ConstraintValidatorFactory` passed to the `Valid-atorContext` are used instead of the `ValidatorFactory`'s `MessageInterpolator` or `TraversableResolver` instances.

Example 4.13. Use of ValidatorFactory

```
ValidatorFactory factory = ...
Validator validatorUsingDefaults = factory.getValidator();
Validator validatorUsingCustomTraversable = factory
                    .usingContext()
                    .traversableResolver( new JPATraversableResolver() )
```

```
                         .getValidator();
```

See Example 4.3 for an example using `ValidatorFactory.getMessageInterpolator()`.

4.4.3. Configuration

`Configuration` collects configuration information, determines the correct provider implementation and delegates it the `ValidatorFactory` creation. This class lets you define:

- the message interpolator strategy instance

- the traversable resolver strategy instance

- the constraint validator factory instance

- XML constraint mappings

- provider specific properties

- whether or not `META-INF/validation.xml` is considered.

A `Configuration` does provide a `MessageInterpolator` implementation following the default Bean Validation `MessageInterpolator` rules as defined in Section 4.3.1. You can access it by calling `getDefaultMessageInterpolator()`. Such an implementation is useful to let a custom `MessageInterpolator` delegate to the standard `MessageInterpolator` (see Section 4.3.2 and an example making use of `getDefaultMessageInterpolator()` in Example 4.4).

A `Configuration` does provide a `TraversableResolver` implementation following the default Bean Validation `TraversableResolver` rules as defined in Section 3.5.2. You can access it by calling `getDefaultTraversableResolver()`. Such an implementation is useful to let a custom `TraversableResolver` delegate to the standard `TraversableResolver`.

A `Configuration` does provide a `ConstraintValidatorFactory` implementation following the default Bean Validation `ConstraintValidatorFactory` rules as defined in Section 2.5. You can access it by calling `getDefaultConstraintValidatorFactory()`. Such an implementation is useful to let a custom `ConstraintValidatorFactory` delegate to the standard `TraversableResolver`.

Clients call `Configuration.buildValidatorFactory()` to retrieve the initialized `ValidatorFactory` instance.

Example 4.14. Configuration interface

```
/**
 * Receives configuration information, selects the appropriate
 * Bean Validation provider and builds the appropriate <code>ValidatorFactory</code>.
 * <p/>
 * Usage:
 * <pre>
 * {@code
 * Configuration<?> configuration = //provided by one of the Validation bootstrap methods
 *     ValidatorFactory = configuration
```

```
 *          .messageInterpolator( new CustomMessageInterpolator() )
 *          .buildValidatorFactory();}
 * </pre>
 * <p/>
 * By default, the configuration information is retrieved from
 * <i>META-INF/validation.xml</i>.
 * It is possible to override the configuration retrieved from the XML file
 * by using one or more of the <code>Configuration</code> methods.
 * <p/>
 * The {@link ValidationProviderResolver} is specified at configuration time
 * (see {@link javax.validation.spi.ValidationProvider}).
 * If none is explicitly requested, the default <code>ValidationProviderResolver</code> is used.
 * <p/>
 * The provider is selected in the following way:
 * <ul>
 * <li>if a specific provider is requested programmatically using
 * <code>Validation.byProvider(Class)</code>, find the first provider implementing
 * the provider class requested and use it</li>
 * <li>if a specific provider is requested in <i>META-INF/validation.xml</i>,
 * find the first provider implementing the provider class requested and use it</li>
 * <li>otherwise, use the first provider returned by the <code>ValidationProviderResolver<code></li>
 * </ul>
 * <p/>
 * Implementations are not meant to be thread-safe.
 *
 * @author Emmanuel Bernard
 */
public interface Configuration<T extends Configuration<T>> {

    /**
     * Ignore data from the <i>META-INF/validation.xml</i> file if this
     * method is called.
     * This method is typically useful for containers that parse
     * <i>META-INF/validation.xml</i> themselves and pass the information
     * via the <code>Configuration</code> methods.
     *
     * @return <code>this</code> following the chaining method pattern.
     */
    T ignoreXmlConfiguration();

    /**
     * Defines the message interpolator used. Has priority over the configuration
     * based message interpolator.
     * If <code>null</code> is passed, the default message interpolator is used
     * (defined in XML or the specification default).
     *
     * @param interpolator message interpolator implementation.
     *
     * @return <code>this</code> following the chaining method pattern.
     */
    T messageInterpolator(MessageInterpolator interpolator);

    /**
     * Defines the traversable resolver used. Has priority over the configuration
     * based traversable resolver.
     * If <code>null</code> is passed, the default traversable resolver is used
     * (defined in XML or the specification default).
     *
     * @param resolver traversable resolver implementation.
     *
     * @return <code>this</code> following the chaining method pattern.
     */
    T traversableResolver(TraversableResolver resolver);

    /**
     * Defines the constraint validator factory. Has priority over the configuration
```

```
    * based constraint factory.
    * If null is passed, the default constraint validator factory is used
    * (defined in XML or the specification default).
    *
    * @param constraintValidatorFactory constraint factory inmplementation.
    *
    * @return <code>this</code> following the chaining method pattern.
    */
  T constraintValidatorFactory(ConstraintValidatorFactory constraintValidatorFactory);

  /**
    * Add a stream describing constraint mapping in the Bean Validation
    * XML format.
    * <p/>
    * The stream should be closed by the client API after the
    * <code>ValidatorFactory</code> has been built. The Bean Validation provider
    * must not close the stream.
    *
    * @param stream XML mapping stream.
    *
    * @return <code>this</code> following the chaining method pattern.
    * @throws IllegalArgumentException if <code>stream</code> is null
    */
  T addMapping(InputStream stream);

  /**
    * Add 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 <code>Configuration<code> subclass.
    * <code>ValidatorFactory factory = Validation.byProvider(ACMEPrivoder.class)
    * .configure()
    * .providerSpecificProperty(ACMEState.FAST)
    * .buildValidatorFactory();
    * </code>
    * This method is typically used by containers parsing <i>META-INF/validation.xml</i>
    * themselves and injecting the state to the 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.
    *
    * If 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</code> following the chaining method pattern.
    *
    * @throws IllegalArgumentException if <code>name</code> is null
    */
  T addProperty(String name, String value);

  /**
    * Return an implementation of the <code>MessageInterpolator</code> interface
    * following the default <code>MessageInterpolator</code> defined in the
    * specification:
    * <ul>
    * <li>use the ValidationMessages resource bundle to load keys</li>
    * <li>use Locale.getDefault()</li>
    * </ul>
    *
```

```
     * @return default MessageInterpolator implementation compliant with the specification
     */
    MessageInterpolator getDefaultMessageInterpolator();

    /**
     * Return an implementation of the <code>TraversableResolver</code> interface
     * following the default <code>TraversableResolver</code> 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 TraversableResolver implementation compliant with the specification
     */
    TraversableResolver getDefaultTraversableResolver();

    /**
     * Return an implementation of the <code>ConstraintValidatorFactory</code> interface
     * following the default <code>ConstraintValidatorFactory</code> defined in the
     * specification:
     * <ul>
     * <li>uses the public no-arg constructor of the <code>ConstraintValidator</code></li>
     * </ul>
     *
     * @return default ConstraintValidatorFactory implementation compliant with the specification
     */
    ConstraintValidatorFactory getDefaultConstraintValidatorFactory();

    /**
     * Build a <code>ValidatorFactory</code> implementation.
     *
     * @return ValidatorFactory
     * @throws ValidationException if the ValidatorFactory cannot be built
     */
    ValidatorFactory buildValidatorFactory();
}
```

A Bean 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 4.15.

Example 4.15. 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 Bean 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 Section 4.4.6 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.

Example 4.16. ConfigurationState interface

```
package javax.validation.spi;

/**
 * Contract between a <code>Configuration</code> and a
 * </code>ValidatorProvider</code> to create a <code>ValidatorFactory</code>.
 * The configuration artifacts defined in the XML configuration and provided to the
 * <code>Configuration</code> are merged and passed along via
 * <code>ConfigurationState</code>.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ConfigurationState {

    /**
     * Returns true if Configuration.ignoreXMLConfiguration() has been called
     * In this case, the ValidatorFactory must ignore META-INF/validation.xml
     *
     * @return {@code true} if META-INF/validation.xml should be ignored
     */
    boolean isIgnoreXmlConfiguration();

    /**
     * Returns the message interpolator of this configuration.
     * Message interpolator is defined in the following decreasing priority:
     * <ul>
     * <li>set via the <code>Configuration</code> programmatic API</li>
     * <li>defined in META-INF/validation.xml provided that 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 provider instance or null if not defined
     */
    MessageInterpolator getMessageInterpolator();

    /**
     * Returns a set of configuration streams.
     * The streams are defined by:
     * <ul>
     * <li>mapping XML streams passed programmatically in <code>Configuration</code></li>
     * <li>mapping XML stream located in the resources defined in</li>
     * META-INF/validation.xml (constraint-mapping element)
     * </ul>
     * Streams represented in the XML configuration and opened by the
     * <code>Configuration</code> implementation must be closed by the
     * <code>Configuration</code> implementation after the <code>ValidatorFactory</code>
     * creation (or if an exception occurs).
     *
     * @return set of input stream
     */
    Set<InputStream> getMappingStreams();

    /**
     * Returns the constraint validator factory of this configuration.
     * The {@code ConstraintValidatorFactory} implementation is defined in the following
     * decreasing priority:
     * <ul>
     * <li>set via the <code>Configuration</code> programmatic API</li>
     * <li>defined in META-INF/validation.xml provided that ignoredXmlConfiguration
     * is 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.
     * <code>TraversableResolver</code> is defined in the following decreasing priority:
     * <ul>
     * <li>set via the <code>Configuration</code> programmatic API</li>
     * <li>defined in META-INF/validation.xml provided that ignoredXmlConfiguration
     * is false. In this case the instance is created via its no-arg constructor.</li>
     * <li>{@code null} if undefined.</li>
     * </ul>
     *
     * @return traversable provider instance or {@code null} if not defined
     */
    TraversableResolver getTraversableResolver();

    /**
     * Returns a map of non type-safe custom properties.
     * Properties defined via:
     * <ul>
     * <li>Configuration.addProperty(String, String)</li>
     * <li>META-INF/validation.xml provided that ignoredXmlConfiguration</li>
     * is false.
     * </ul>
     * If a property is defined both programmatically and in XML,
     * the value defined programmatically has priority
     *
     * @return 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 Section 4.4.6) 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 Bean Validation client through the `Validation` methods. `Configuration` instances are created by `ValidationProvider`.

If a problem occurs while building the `ValidationFactory`, 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 4.17. Use Configuration

```
Configuration<?> configuration = ...
ValidatorFactory factory = configuration
            .messageInterpolator( new WBMessageInterpolator() )
            .traversableResolver( new JPAAwareTraversableResolver() )
            .buildValidatorFactory();
```

4.4.4. ValidationProvider and ValidationProviderResolver

`ValidationProvider` is the contract between the bootstrap process and a specific Bean Validation provider. `Valid-`

ationProviderResolver implements the discovery mechanism for Bean Validation provider implementations. Any Bean Validation client can implement such a discovery mechanism but it is typically implemented by containers having specific classloader structures and restrictions.

4.4.4.1. ValidationProviderResolver

ValidationProviderResolver returns the list of Bean 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.

Example 4.18. ValidationProviderResolver

```
/**
 * Determines the list of Bean Validation providers available in the runtime environment
 * <p/>
 * Bean Validation providers are identified by the presence of
 * META-INF/services/javax.validation.spi.ValidationProvider
 * files following the Service Provider pattern described
 * <a href="http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Service%20Provider">here</a>
 * <p/>
 * Each META-INF/services/javax.validation.spi.ValidationProvider file contains the list of
 * <code>ValidationProvider</code> implementations each of them representing a provider.
 * <p/>
 * Implementations must be thread-safe.
 *
 * @author Emmanuel Bernard
 */
public interface ValidationProviderResolver {
    /**
     * Returns a list of ValidationProviders 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://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Service%20Provider. Bean Validation providers must supply a service provider configuration file by creating a text file javax.validation.spi.ValidationProvider and placing it in the META-INF/services directory of one of its jar files. The content of the file should contain the name of the provider implementation class of the javax.validation.spi.ValidationProvider interface.

Bean 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 Bean 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 classloader constrained container like OSGi or in a tool environment (IDE).

The default ValidationProviderResolver can be accessed via BootStrap-State.getDefaultValidationProviderResolver(). This method is typically used by the Bean Validation provider Configuration implementation.

4.4.4.2. 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`.

Example 4.19. ValidationProvider

```
package javax.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.
 *
 * <code>T</code> represents 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 <code>Configuration</code> instance implementing <code>T</code>,
     * the <code>Configuration</code> subinterface.
     * The returned <code>Configuration</code> instance must use the current provider
     * (<code>this</code>) to build the <code>ValidatorFactory</code> instance.
     * <p/>
     *
     * @param state bootstrap state
     *
     * @return specific Configuration implementation
     */
    T createSpecializedConfiguration(BootstrapState state);

    /**
     * Returns a <code>Configuration</code> instance. This instance is not bound to
     * use the current provider. The choice of provider follows the algorithm described
     * in {@link javax.validation.Configuration}
     * <p/>
     * The <code>ValidationProviderResolver</code> used by <code>Configuration</code>
     * is provided by <code>state</code>.
     * If null, the default <code>ValidationProviderResolver</code> is used.
     *
     * @param state bootstrap state
     *
     * @return Non specialized Configuration implementation
     */
    Configuration<?> createGenericConfiguration(BootstrapState state);

    /**
     * Build a <code>ValidatorFactory</code> using the current provider implementation.
```

```
     * The <code>ValidatorFactory</code> is assembled and follows the configuration passed
     * via <code>ConfigurationState</code>.
     * <p>
     * The returned <code>ValidatorFactory</code> is properly initialized and ready for use.
     * </p>
     *
     * @param configurationState the configuration descriptor
     *
     * @return the instanciated ValidatorFactory
     * @throws javax.validation.ValidationException if the ValidatorFactory cannot be built
     */
    ValidatorFactory buildValidatorFactory(ConfigurationState configurationState);
}
```

Example 4.20. BootstrapState interface

```
package javax.validation.spi;

/**
 * Defines the state used to bootstrap the <code>Configuration</code>
 *
 * @author Emmanuel Bernard
 * @author Sebastian Thomschke
 */
public interface BootstrapState {
    /**
     * User defined <code>ValidationProviderResolver</code> strategy
     * instance or <code>null</code> if undefined.
     *
     * @return ValidationProviderResolver instance or null
     */
    ValidationProviderResolver getValidationProviderResolver();

    /**
     * Specification default <code>ValidationProviderResolver</code>
     * strategy instance.
     *
     * @return default implementation of ValidationProviderResolver
     */
    ValidationProviderResolver getDefaultValidationProviderResolver();
}
```

A client can request a specific Bean 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 Bean Validation provider is the `ValidationProvider` implementation specific to this provider.

A `ValidationProvider` implementation is linked (via it's generic parameter) to a specific sub interface of `Configuration`. The Bean 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 method but is the natural holder for provider specific configuration methods.

Example 4.21. 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 `ACME-Configuration` 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 Bean Validation provider is requested, the first `ValidationProvider` returned by the `ValidationProviderResolver` strategy is returned.

Every Bean Validation provider must provide a `ValidationProvider` implementation containing a public no-arg constructor and add the corresponding `META-INF/services/javax.validation.spi.ValidationProvider` file descriptor in one of its jars.

If a problem occurs while building the `ValidationFactory`, 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

4.4.5. Validation

The `Validation` class is the entry point used to bootstrap Bean Validation providers. The first entry point, `build-DefaultValidatorFactory()`, is considered to be the default `ValidatorFactory` and is equivalent to the `Validat-orFactory` returned by `Validation.byDefaultProvider().configure().buildValidatorFactory()`.

Warning

Should the resolver strategy be configurable by XML

Example 4.22. Validation methods available

```
/**
 * This class is the entry point for Bean Validation. There are three ways
 * to bootstrap it:
 * <ul>
 * <li>
 * The easiest approach is to build the default <code>ValidatorFactory</code>.
 * <pre>{@code ValidatorFactory factory = Validation.buildDefaultValidatorFactory();}</pre>
 * In this case, the default validation provider resolver
 * will be used to locate available providers.
 * 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
 * <code>ValidationProviderResolver</code> instance is used.</li>
 * </ul>
 * </li>
 * <li>
 * The second bootstrap approach allows to choose a custom
 * <code>ValidationProviderResolver</code>. The chosen
 * <code>ValidationProvider</code> is then determined in the same way
 * as in the default bootstrapping case (see above).
 * <pre>{@code
 * Configuration<?> 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</code>.
 * <pre>{@code
 * ACMEConfiguration configuration = Validation
 *    .byProvider(ACMEProvider.class)
 *    .providerResolver( new MyResolverStrategy() )  // optionally set the provider resolver
 *    .configure();
 * ValidatorFactory factory = configuration.buildValidatorFactory();}
 * </pre>
 * </li>
 * </ul>
 * Note:<br/>
 * <ul>
```

```
 * <li>
 * The <code>ValidatorFactory</code> object built by the bootstrap process should be cached
 * and shared amongst <code>Validator</code> consumers.
 * </li>
 * <li>
 * This class is thread-safe.
 * </li>
 * </ul>
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public class Validation {

    /**
     * Build and return a <code>ValidatorFactory</code> instance based on the
     * default Bean 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()</code>
     *
     * @return <code>ValidatorFactory</code> instance.
     *
     * @throws ValidationException if the ValidatorFactory cannot be built
     */
    public static ValidatorFactory buildDefaultValidatorFactory() {
        [...]
    }

    /**
     * Build a <code>Configuration</code>. 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 exsist or if no provider is specified,
     * the first available provider will be returned.
     *
     * @return instance building a generic <code>Configuration</code>
     *         compliant with the bootstrap state provided.
     */
    public static GenericBootstrap byDefaultProvider() {
        [...]
    }

    /**
     * Build a <code>Configuration</code> for a particular provider implementation.
     * Optionally overrides the provider resolution strategy used to determine the provider.
     * <p/>
     * Used by applications targeting a specific provider programmatically.
     * <p/>
     * <pre>
     * ACMEConfiguration configuration =
     *     Validation.byProvider(ACMEProvider.class)
     *               .providerResolver( new MyResolverStrategy() )
     *               .configure();
     * </pre>,
     * where <code>ACMEConfiguration</code> is the
     * <code>Configuration</code> sub interface uniquely identifying the
```

```
     * ACME Bean Validation provider. and <code>ACMEProvider</code> is the
     * <code>ValidationProvider</code> implementation of the ACME provider.
     *
     * @param providerType the <code>ValidationProvider</code> implementation type
     *
     * @return instance building a provider specific <code>Configuration</code>
     *         sub interface implementation.
     */
    public static <T extends Configuration<T>, U extends ValidationProvider<T>>
            ProviderSpecificBootstrap<T> byProvider(Class<U> providerType) {
        [...]
    }

    [...]
}
```

The second entry point lets the client provide a custom `ValidationProviderResolution` 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.

Example 4.23. GenericBootstrap interface

```
package javax.validation.bootstrap;

/**
 * Defines the state used to bootstrap Bean Validation and
 * creates a provider agnostic <code>Configuration</code>.
 *
 * @author Emmanuel Bernard
 */
public interface GenericBootstrap {
    /**
     * Defines the provider resolution strategy.
     * This resolver returns the list of providers evaluated
     * to build the <code>Configuration</code>
     * <p/>
     * If no resolver is defined, the default <code>ValidationProviderResolver</code>
     * implementation is used.
     *
     * @return <code>this</code> following the chaining method pattern
     */
    GenericBootstrap providerResolver(ValidationProviderResolver resolver);

    /**
     * Returns a generic <code>Configuration</code> implementation.
     * At this stage the provider used to build the <code>ValidatorFactory</code>
     * is not defined.
     * <p/>
     * The <code>Configuration</code> implementation is provided by the first provider
     * returned by the <code>ValidationProviderResolver</code> strategy.
     *
     * @return a Configuration implementation compliant with the bootstrap state
     * @throws javax.validation.ValidationException if the Configuration object cannot be built
     *                        this is generally due to an issue with the ValidationProviderResolver
     */
    Configuration<?> configure();
```

```
}
```

The last entry point lets the client define the specific Bean Validation provider requested as well as a custom `Val-idationProviderResolver` implementation if needed. The entry point method, `Validation.byProvider(Class<U> providerType)`, takes the provider specific `ValidationProvider` implementation type and returns a `ProviderSpe-cificBootstrap` 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.

Example 4.24. ProviderSpecificBootstrap interface

```
package javax.validation.bootstrap;

/**
 * Defines the state used to bootstrap Bean Validation and
 * creates a provider specific <code>Configuration</code>
 * of type<code>T</code>.
 * <p/>
 * The specific <code>Configuration</code> is linked to the provider via the generic
 * parameter of the <code>ValidationProvider</code> implementation.
 * <p/>
 * The requested provider is the first provider instance assignable to
 * the requested provider type (known when <code>ProviderSpecificBootstrap</code> is built).
 * The list of providers evaluated is returned by {@link ValidationProviderResolver}.
 * If no <code>ValidationProviderResolver</code> is defined, the
 * default <code>ValidationProviderResolver</code> strategy is used.
 *
 * @author Emmanuel Bernard
 */
public interface ProviderSpecificBootstrap<T extends Configuration<T>> {

    /**
     * Optionally defines the provider resolver implementation used.
     * If not defined, use the default <code>ValidationProviderResolver</code>
     *
     * @param resolver <code>ValidationProviderResolver</code> implementation used
     *
     * @return <code>this</code> following the chaining method pattern
     */
    public ProviderSpecificBootstrap<T> providerResolver(ValidationProviderResolver resolver);

    /**
     * Determines the provider implementation suitable for <code>T</code> and delegates
     * the creation of this specific <code>Configuration</code> subclass to the provider.
     *
     * @return <code>Configuration</code> sub interface implementation
     * @throws javax.validation.ValidationException if the Configuration object cannot be built
     *                      this is generally due to an issue with the ValidationProviderResolver
     */
    public T configure();
}
```

`ProviderSpecificBootstrap.configure()` must return the result of `ValidationPro-vider.createSpecializedConfiguration(BootstrapState state)`. The state parameter holds the `Validation-ProviderResolver` passed to `ProviderSpecificBootstrap`. The validation provider instance used is the one as-

signable to the type passed as a parameter in `Validation.byProvider(Class)`. The validation provider is selected according to the algorithm described in (Section 4.4.4.2).

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>> ProviderSpecific-Bootstrap<T> byProvider(Class<U> providerType)`

The bootstrap API is designed to allow complete portability amongst Bean Validation provider implementations. The bootstrap implementation must ensure it can bootstrap third party providers.

When building the `Configuration` object, if the `ValidationProviderResolver` either fail or if the expected provider is not found, a `ValidationException` is raised.

4.4.6. 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 Bean 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. This element is optional.

`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. This element is optional.

`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. This element is optional.

`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(InputString)` 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 `javax.validation` and its subnamespaces must not be used for vendor-specific information. The

namespace `javax.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.

If a public no-arg constructor is missing, a `ValidationException` is raised during the `Configuration.buildValidatorFactory()` call.

Example 4.25. Example of META-INF/validation.xml file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
        xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://jboss.org/xml/ns/javax/validation/configuration validation-configuration-1.0.xsd">
    <default-provider>com.acme.ACMEProvider</default-provider>
    <message-interpolator>com.acme.ACMEAwareMessageInterpolator</message-interpolator>

    <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 Section 7.2.

4.4.7. Usage

The Bean Validation bootstrap API can be used directly by the application, through the use of a container or by framework in need for validation. In all cases, the following rules apply:

- `ValidatorFactory` is a thread-safe object that should be built once per deployment unit

- `Validator` is thread-safe too and should be considered a lightweight object. `ValidatorFactory` would typically implement appropriate `Validator` instance caching strategies if needed.

Containers such as Java EE, dependency injection frameworks, component frameworks are encouraged to propose access to `ValidatorFactory` and `Validator` objects in a way that respects the previous rules. For example, injection of `Validator` should be possible.

5

Constraint metadata request APIs

The Bean 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 JSRs. The Bean 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 Bean Validation specification for these services avoiding any unnecessary duplication work from an application and infrastructure point of view.

5.1. Validator

The main API to access all metadata related to a given object is `Validator` (see Section 4.4 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.

```
/**
 * Validate bean instances. Implementations of this interface must be thread-safe.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface Validator {

    [...] //See 4.1

    /**
     * Return the descriptor object describing bean constraints.
     * The returned object (and associated objects including
     * <code>ConstraintDescriptor<code>s) are immutable.
     *
     * @param clazz class or interface type evaluated
     *
     * @return the bean descriptor for the specified class.
     *
     * @throws IllegalArgumentException if clazz is 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 Section 3.1.1) and providing access to the property level constraints metadata.

If a constraint definition or declaration hosted by the requested class (or any of it's superclasses and interfaces ac-

cording to the constraint propagation rules) is invalid, a `ValidationException` is raised. This can be a subclass of `ValidationException` like `ConstraintDefinitionException`, `ConstraintDeclarationException`, `Unexpected-TypeException`.

5.2. 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 field, a method or a class).

`ElementDescriptor` lives in the `javax.validation.metadata` package.

```
package javax.validation.metadata;

/**
 * Describes a validated element (class, field or property).
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ElementDescriptor {

    /**
     * Return <code>true</code> if at least one constraint declaration is present
     * for this element in the class hierarchy, <code>false</code> otherwise.
     */
    boolean hasConstraints();

    /**
     * @return Statically defined returned type.
     */
    Class<?> getElementClass();

    /**
     * Return all constraint descriptors for this element in the class hierarchy
     * or an empty <code>Set</code> if none are present.
     *
     * @return <code>Set</code> of constraint descriptors for this element
     */
    Set<ConstraintDescriptor<?>> getConstraintDescriptors();

    /**
     * Find constraints and potentially restricts them to certain criteria.
     *
     * @return ConstraintFinder object.
     */
    ConstraintFinder findConstraints();

    /**
     * Declare restrictions on retrieved constraints.
     * Restrictions are cumulative.
     *
     * A <code>ConstraintFinder</code> is not thread-safe. The set of matching
     * <code>ConstraintDescriptor</code> is.
     */
    interface ConstraintFinder {
        /**
         * Restrict to the constraints matching a given set of groups for this element
         *
         * This method respects group sequences and group inheritance (including
         * class-level <code>Default</code> group overriding) but does not return
         * <code>ConstraintDescriptor</code>s in any particular order.
```

```
        * Specifically, ordering of the group sequence is not respected.
        *
        * @param groups groups targeted
        *
        * @return <code>this</code> following the chaining method pattern
        */
       ConstraintFinder unorderedAndMatchingGroups(Class<?>... groups);

       /**
        * Restrict to the constraints matching the provided scope for this element.
        *
        * Defaults to <code>Scope.HIERARCHY</code>
        *
        * @param scope expected scope
        * @return <code>this</code> following the chaining method pattern
        */
       ConstraintFinder lookingAt(Scope scope);

       /**
        * Restrict to the constraints hosted on the listed <code>types</code>
        * for a given element.
        *
        * Default to all possible types of the element.
        *
        * Typically used to restrict to fields (<code>FIELD</code>)
        * or getters (<code>METHOD</code>)
        *
        * @param types targeted types
        * @return <code>this</code> following the chaining method pattern
        */
       ConstraintFinder declaredOn(ElementType... types);

       /**
        * Retrieve the constraint descriptors following the defined
        * restrictions and hosted on the element described by
        * <code>ElementDescriptor</code>
        *
        * @return matching constraint descriptors
        */
       Set<ConstraintDescriptor<?>> getConstraintDescriptors();

       /**
        * Returns <code>true</code> if at least one constraint declaration
        * matching the restrictions is present on the element,
        * <code>false</code> otherwise.
        *
        * @return is there any constraint
        */
       boolean hasConstraints();
    }
}
```

```
package javax.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 either the object type for a class, or the returned type for a property.

`getConstraintDescriptors` returns all the `ConstraintDescriptors` (see Section 5.5) 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 (class, field or property) 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 one 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`.

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 5.1. 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();
```

`unorderedAndMatchingGroups` restricts to the `ConstraintDescriptors` (see Section 5.5) matching the set of groups passed as parameters and present on the element. 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 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 BeanDescritptor (Scope.LOCAL_ELEMENT), or constraints belonging to the element but hosted anywhere in the class hierarchy (Scope.HIERARCHY).

5.3. BeanDescriptor

The BeanDescriptor interface describes a constrained Java Bean. This interface is returned by Validator.getConstraintsForClass(Class<?>).

BeanDescriptor lives in the javax.validation.metadata package.

```
package javax.validation.metadata;

/**
 * Describes a constrained Java Bean and the constraints associated to it.
 *
 * @author Emmanuel Bernard
 */
public interface BeanDescriptor extends ElementDescriptor {
    /**
     * Returns <code>true</code> 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 cascade (<code>@Valid</code>)</li>
     * </ul>
     *
     * @return <code>true</code> if the bean involves validation, <code>false</code> otherwise.
     */
    boolean isBeanConstrained();

    /**
     * Return the property descriptor for a given property.
     * Return <code>null</code> if the property does not exist or has no
     * constraint nor is marked as cascaded (see {@link #getConstrainedProperties()} )
     * <p/>
     * The returned object (and associated objects including <code>ConstraintDescriptor</code>s)
     * are immutable.
     *
     * @param propertyName property evaluated
     *
     * @return the property descriptor for a given property.
     *
     * @throws IllegalArgumentException if propertyName is null
     */
    PropertyDescriptor getConstraintsForProperty(String propertyName);

    /**
     * Returns a set of property descriptors having at least one constraint defined
     * or marked as cascaded (<code>@Valid<c/ode>). If not property matches,
     * an empty set is returned.
     */
    Set<PropertyDescriptor> getConstrainedProperties();
}
```

isBeanConstrained returns true if the given class (and superclasses and interfaces) hosts at least one validation de-

claration (either constraint or `@Valid` annotation). If the method returns false, the Bean 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 Section 3.1.2). 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.

`getConstrainedProperties` returns the `PropertyDescriptor`s of the bean properties having at least one constraint or being cascaded (`@Valid` annotation).

5.4. PropertyDescriptor

The `PropertyDescriptor` interface describes a constrained property of a Java Bean.

`PropertyDescriptor` lives in the `javax.validation.metadata` package.

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 Java Bean rules are returned by this descriptor.

```
package javax.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 {
    /**
     * Is the property marked by the <code>@Valid</code> annotation.
     * @return <code>true</code> if the annotation is present, <code>false</code> otherwise.
     */
    boolean isCascaded();

    /**
     * Name of the property acording to the Java Bean specification.
     * @return property name.
     */
    String getPropertyName();
}
```

The `isCascaded` method returns `true` if the property is marked with `@Valid`.

`getPropertyName` returns the property name as described in Section 4.2.

5.5. ConstraintDescriptor

A `ConstraintDescriptor` object describes a given constraint declaration (i.e. a constraint annotation).

`ConstraintDescriptor` lives in the `javax.validation.metadata` package.

```
package javax.validation.metadata;
```

```
/**
 * Describes a single constraint and its composing constraints.
 * <code>T</code> is 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 set of groups the constraint is applied on.
     * If the constraint declares no group, a set with only the <code>Default</code>
     * 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();

    /**
     * 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.
     * 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();

    /**
     * Return a set of composing <code>ConstraintDescriptor</code>s where each
     * descriptor describes a composing constraint. <code>ConstraintDescriptor</code>
     * instances of composing constraints reflect overridden attribute values in
     * {@link #getAttributes()}  and {@link #getAnnotation()}.
     *
     * @return a set of <code>ConstraintDescriptor<code> objects or an empty set
     *         in case there are no composing constraints.
     */
    Set<ConstraintDescriptor<?>> getComposingConstraints();

    /**
     * @return true if the constraint is annotated with <code>@ReportAsSingleViolation</code>
```

```
    */
    boolean isReportAsSingleViolation();
}
```

getAnnotation returns the annotation instance (or an annotation instance representing the given constraint declaration). If ConstraintDescriptor represents a composing annotation (see Section 2.3), 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 Section 2.3), the returned Map must reflects attribute overriding.

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.

getConstraintValidatorClasses returns the ConstraintValidator classes associated with the constraint.

5.6. Example

Assuming the following @NotEmpty definition

```
package com.acme.constraint;

@Documented
@NotNull
@Size(min=1)
@ReportAsSingleViolation
@Constraint(validatedBy = NotEmpty.NotEmptyValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface NotEmpty {
    String message() default "{com.acme.constraint.NotEmpty.message}"
    Class<?> groups() default {};

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

    class NotEmptyValidator implements ConstraintValidator<NotEmpty, String> {
        public void initialize(NotEmpty constraintAnnotation) {}

        public boolean isValid(String value, ConstraintValidatorContext context) {
            return true;
        }
    }
}
```

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 {
    private String title;
    private String description;

    @Valid
    @NotNull
    private Author author;

    @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;
    }
}
```

The following assertions are true.

```
BeanDescriptor bookDescriptor = validator.getConstraintsForClass(Book.class);

assert ! bookDescriptor.hasConstraints();

assert bookDescriptor.isBeanConstrained();

assert bookDescriptor.getConstraintDescriptors().size() == 0 //no bean-level constraint

//more specifically "author" and "title"
assert bookDescriptor.getConstrainedProperties().size() == 2;

//not a property
assert bookDescriptor.getConstraintsForProperty("doesNotExist") == null;

//property with no constraint
assert bookDescriptor.getConstraintsForProperty("description") == null;

ElementDescriptor 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().getAnnotationType().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().getAnnotationType().equals( NotNull.class ) ) {
        notNullPresence = true;
    }
}
assert notNullPresence;

//assuming the implementation returns the Size constraint second
constraintDescriptor = propertyDescriptor.getConstraintDescriptors().iterator().next().next();
assert constraintDescriptor.getAnnotation().getAnnotationType().equals( Size.class );
assert constraintDescriptor.getAttributes().get("max") == 30;
assert constraintDescriptor.getGroups().size() == 1;

propertyDescriptor = bookDescriptor.getConstraintsForProperty("author");
assert propertyDescriptor.getConstraintDescriptors().size() == 1
assert propertyDescriptor.isCascaded()
```

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 Bean 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 Section 7.1.2).

All built-in constraints are in the `javax.validation.constraints` package. Here is the list of constraints and their declaration.

Example 6.1. @Null constraint

```
package javax.validation.constraints;

/**
 * The annotated element must be <code>null</code>.
 * Accepts any type.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Null {
    String message() default "{javax.validation.constraints.Null.message}";

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

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

    /**
     * Defines several <code>@Null</code> annotations on the same element
     * @see javax.validation.constraints.Null
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Null[] value();
    }
```

```
}
```

Example 6.2. @NotNull constraint

```
package javax.validation.constraints;

/**
 * The annotated element must not be <code>null</code>.
 * Accepts any type.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface NotNull {
    String message() default "{javax.validation.constraints.NotNull.message}";

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

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

    /**
     * Defines several <code>@NotNull</code> annotations on the same element
     * @see javax.validation.constraints.NotNull
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        NotNull[] value();
    }
}
```

Example 6.3. @AssertTrue constraint

```
package javax.validation.constraints;

/**
 * The annotated element must be true.
 * Supported types are <code>boolean</code> and <code>Boolean</code>
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface AssertTrue {
    String message() default "{javax.validation.constraints.AssertTrue.message}";

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

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

    /**
     * Defines several <code>@AssertTrue</code> annotations on the same element
     * @see AssertTrue
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        AssertTrue[] value();
    }
}
```

Example 6.4. @AssertFalse constraint

```
package javax.validation.constraints;

/**
 * The annotated element must be false.
 * Supported types are <code>boolean</code> and <code>Boolean</code>
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface AssertFalse {
    String message() default "{javax.validation.constraints.AssertFalse.message}";

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

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

    /**
     * Defines several <code>@AssertFalse</code> annotations on the same element
     * @see javax.validation.constraints.AssertFalse
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        AssertFalse[] value();
    }
}
```

Example 6.5. @Min constraint

```
package javax.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</code></li>
 * <li><code>BigInteger</code></li>
 * <li><code>byte</code>, <code>short</code>, <code>int</code>, <code>long</code>,
 * and their respective wrappers</li>
 * </ul>
 * Note that <code>double</code> and <code>float</code> are not supported due to rounding errors
 * (some providers might provide some approximative support)
 * <p/>
 * <code>null</code> elements are considered valid
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Min {
    String message() default "{javax.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 <code>@Min</code> annotations on the same element
     * @see Min
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Min[] value();
    }
}
```

Example 6.6. @Max constraint

```
package javax.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</code></li>
 * <li><code>BigInteger</code></li>
 * <li><code>byte</code>, <code>short</code>, <code>int</code>, <code>long</code>,
 * and their respective wrappers</li>
 * </ul>
```

```
 * Note that <code>double</code> and <code>float</code> are not supported due to rounding errors
 * (some providers might provide some approximative support)
 * <p/>
 * <code>null</code> elements are considered valid
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Max {
    String message() default "{javax.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 <code>@Max</code> annotations on the same element
     * @see Max
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Max[] value();
    }
}
```

Example 6.7. @DecimalMin constraint

```
package javax.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</code></li>
 * <li><code>BigInteger</code></li>
 * <li><code>String</code></li>
 * <li><code>byte</code>, <code>short</code>, <code>int</code>, <code>long</code>,
 * and their respective wrappers</li>
 * </ul>
 * Note that <code>double</code> and <code>float</code> are not supported due to rounding errors
 * (some providers might provide some approximative support)
 * <p/>
 * <code>null</code> elements are considered valid
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
```

```
@Documented
@Constraint(validatedBy = {})
public @interface DecimalMin {
    String message() default "{javax.validation.constraints.DecimalMin.message}";

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

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

    /**
     * The <code>String</code> representation of the min value according to the
     * <code>BigDecimal</code> string representation
     * @return value the element must be higher or equal to
     */
    String value();

    /**
     * Defines several <code>@DecimalMin</code> annotations on the same element
     * @see DecimalMin
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        DecimalMin[] value();
    }
}
```

Example 6.8. @DecimalMax constraint

```
package javax.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</code></li>
 * <li><code>BigInteger</code></li>
 * <li><code>String</code></li>
 * <li><code>byte</code>, <code>short</code>, <code>int</code>, <code>long</code>,
 * and their respective wrappers</li>
 * </ul>
 * Note that <code>double</code> and <code>float</code> are not supported due to rounding errors
 * (some providers might provide some approximative support)
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { })
public @interface DecimalMax {
    String message() default "{javax.validation.constraints.DecimalMax.message}";

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

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

    /**
     * The <code>String</code> representation of the max value according to the
     * <code>BigDecimal</code> string representation
     *
     * @return value the element must be lower or equal to
     */
    String value();

    /**
     * Defines several <code>@DecimalMax</code> annotations on the same element
     *
     * @author Emmanuel Bernard
     * @see DecimalMax
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
            @interface List {
        DecimalMax[] value();
    }
}
```

Example 6.9. @Size constraint

```
package javax.validation.constraints;

/**
 * The annotated element size must be between the specified boundaries (included).
 *
 * Supported types are:
 * <ul>
 * <li><code>String</code> (string length is evaludated)</li>
 * <li><code>Collection</code> (collection size is evaluated)</li>
 * <li><code>Map</code> (map size is evaluated)</li>
 * <li>Array (array length is evaluated)</li>
 *
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Size {
    String message() default "{javax.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 <code>@Size</code> annotations on the same element
     * @see Size
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Size[] value();
    }
}
```

Example 6.10. @Digits constraint

```
package javax.validation.constraints;

/**
 * The annotated element must be a number within accepted range
 * Supported types are:
 * <ul>
 * <li><code>BigDecimal</code></li>
 * <li><code>BigInteger</code></li>
 * <li><code>String</code></li>
 * <li><code>byte</code>, <code>short</code>, <code>int</code>, <code>long</code>,
 * and their respective wrapper types</li>
 * </ul>
 * <p/>
 * <code>null</code> elements are considered valid
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Digits {
    String message() default "{javax.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 <code>@Digits</code> annotations on the same element
     * @see Digits
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
```

```
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Digits[] value();
    }
}
```

Example 6.11. @Past constraint

```
package javax.validation.constraints;

/**
 * The annotated element must be a date in the past.
 * Now is defined as the current time according to the virtual machine
 * The calendar used if the compared type is of type <code>Calendar</code>
 * is the calendar based on the current timezone and the current locale.
 * <p/>
 * Supported types are:
 * <ul>
 * <li><code>java.util.Date</code></li>
 * <li><code>java.util.Calendar</code></li>
 * </ul>
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Past {
    String message() default "{javax.validation.constraints.Past.message}";

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

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

    /**
     * Defines several <code>@Past</code> annotations on the same element
     * @see Past
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Past[] value();
    }
}
```

Example 6.12. @Future constraint

```
package javax.validation.constraints;

/**
 * The annotated element must be a date in the future.
```

```
 * Now is defined as the current time according to the virtual machine
 * The calendar used if the compared type is of type <code>Calendar</code>
 * is the calendar based on the current timezone and the current locale.
 * <p/>
 * Supported types are:
 * <ul>
 * <li><code>java.util.Date</code></li>
 * <li><code>java.util.Calendar</code></li>
 * </ul>
 * <p/>
 * <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Future {
    String message() default "{javax.validation.constraints.Future.message}";

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

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

    /**
     * Defines several <code>@Future</code> annotations on the same element
     * @see Future
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Future[] value();
    }
}
```

Example 6.13. @Pattern constraint

```
package javax.validation.constraints;

/**
 * The annotated String must match the following regular expression.
 * The regular expression follows the Java regular expression conventions
 * see {@link java.util.regex.Pattern}.
 *
 * Accepts String. <code>null</code> elements are considered valid.
 *
 * @author Emmanuel Bernard
 */
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
public @interface Pattern {
    /**
     * @return The regular expression to match.
     */
    String regexp();
```

```
    /**
     * @return Array of <code>Flag</code>s considered when resolving the regular expression.
     */
    Flag[] flags() default {};

    /**
     * @return The error message template.
     */
    String message() default "{javax.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 <code>@Pattern</code> annotations on the same element
     * @see Pattern
     *
     * @author Emmanuel Bernard
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Pattern[] value();
    }
}
```

XML deployment descriptor

Two kind of XML descriptors are used by Bean Validation. The first one describes the Bean Validation configuration provided as `META-INF/validation.xml`. The second one describes constraints declarations and closely matches the annotations declaration approach.

7.1. Constraint definition and declaration

Bean 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 Bean 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 Bean 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 Section 7.1.3). Likewise, `ConstraintDescriptor.getConstraintValidatorClasses()` must reflect XML based constraint definition overriding (see Section 7.1.2).

A given class must not be described more than once amongst 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 amongst all the XML mapping descriptors. If any of these rule is violated in a given validation deployment, a `ValidationException` is raised during the creation of the `ValidatorFactory`.

The schema is provided in Section 7.1.4.

7.1.1. 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 annotation are ignored for classes described in XML. You can force Bean Validation to consider both annotations and XML constraint declarations by using `ignore-annotation="false"` on `bean`.

Note

The `ignore-annotation` setting is not inherited from nor by the class hierarchy. In other words, it only applies to the current bean only.

If the name of the class does refer to a class not present in in the classpath, a `ValidationException` is raised.

Example 7.1. Example of bean XML declaration

```
<constraint-mappings
        xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd">

    <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>
```

7.1.1.1. Class-level overriding

Class level annotations are described via the `class` element. If `ignore-annotations` is declared, Bean 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 Bean 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 7.2. Example of class-level declaration

```
<constraint-mappings
        xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd">
    <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>
```

7.1.1.2. Field-level overriding

Field level annotations are described via the `field` element. The `name` attribute correspond to the name of the field considered. If `ignore-annotations` is declared, Bean 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 Bean Validation annotations on the targeted field are ignored (including the `@Valid`). 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`.

If the name of the field does not correspond to a field in the given bean a `ValidationException` is raised.

Example 7.3. Field-level declaration

```
<constraint-mappings
        xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd">
    <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>
```

7.1.1.3. Property-level overriding

Property-level annotations are described via the `getter` element. The `name` attribute correspond to the name of the property considered as defined in Section 3.1.2 (for example a getter `String getAge()` would have `<getter name="age"/>` as a corresponding descriptor). If `ignore-annotations` is declared, Bean 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 Bean Validation annotations on the targeted property are ignored (including the `@Valid`). 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`.

If the name of the property does not correspond to a property in the given bean a `ValidationException` is raised.

Example 7.4. Property-level declaration

```
<constraint-mappings
        xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd">
    <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>
```

7.1.1.4. Constraint declaration

A new constraint declaration is 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.

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 Section 7.1.3 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 7.5. Constraint declaration

```
<constraint-mappings
        xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd">
    <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>

        <getter name="orders">
            <valid/>


            <!-- @Size(message="Size is limited",
                    groups={Default.class, LightValidation.class},
                    max=30
                 )
            -->
            <constraint annotation="javax.validation.constraint.Size">
                <message>Size is limited</message>
                <groups>
                    <value>com.acme.app.model.LightValidation</value>
                    <value>javax.persistence.Default</value>
```

```
                </groups>
                <payload>
                    <value>com.acme.app.model.WARN</value>
                </payload>
                <element name="max">30</element>
            </constraint>


        </getter>
    </bean>
</constraint-mappings>
```

7.1.2. 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` 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` come before XML based `ConstraintValidatot` in the array. The new list is returned by `ConstraintDescriptor.getConstraintValidatorClasses()`.

Example 7.6. Overriding constraint definitions

```
<constraint-mappings
        xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd">
   <default-package>com.acme.app.domain</default-package>
   <bean class="com.acme.common.model.Address">
       [...]
    </bean>

   <constraint-definition annotation="javax.validation.constraint.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>
```

7.1.3. 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 repres-

ented by the `value` element.

`byte` are represented according to the rules defined in `Byte.parseByte(String)`.

`short` are represented according to the rules defined in `Short.parseShort(String)`.

`int` are represented according to the rules defined in `Integer.parseInt(String)`.

`long` are represented according to the rules defined in `Long.parseLong(String)`.

`float` are represented according to the rules defined in `Float.parseFloat(String)`.

`double` are represented according to the rules defined in `Double.parseDouble(String)`.

`boolean` are represented according to the rules defined in `Boolean.parseBoolean(String)`.

`char` are 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. 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.

7.1.4. XML Schema

This section contains the XML schema used for constraint mapping.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           targetNamespace="http://jboss.org/xml/ns/javax/validation/mapping"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0">
    <xs:element name="constraint-mappings"
                type="map:constraint-mappingsType"
                xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>

    <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="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"
                        xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            <xs:element type="map:constraint-definitionType"
                        name="constraint-definition"
                        maxOccurs="unbounded"
                        minOccurs="0"
                        xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
        </xs:sequence>
    </xs:complexType>
    <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"
                        xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            <xs:element type="map:payloadType"
                        name="payload"
                        minOccurs="0"
                        xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            <xs:element type="map:elementType"
                        name="element"
                        maxOccurs="unbounded"
                        minOccurs="0"
                        xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
        </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"
                        xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
        </xs:sequence>
        <xs:attribute type="xs:string" name="name" use="required"/>
    </xs:complexType>
    <xs:complexType name="classType">
        <xs:sequence>
            <xs:element type="map:groupSequenceType"
                        name="group-sequence"
                        minOccurs="0"
                        xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            <xs:element type="map:constraintType"
                        name="constraint"
                        maxOccurs="unbounded"
                        minOccurs="0"
                        xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
        </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"
                            xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping">
                </xs:element>
                <xs:element type="map:fieldType"
                            name="field"
                            minOccurs="0"
                            maxOccurs="unbounded"
                            xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
                <xs:element type="map:getterType"
                            name="getter"
                            minOccurs="0"
                            maxOccurs="unbounded"
                            xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            </xs:sequence>
            <xs:attribute type="xs:string" name="class" use="required"/>
            <xs:attribute type="xs:boolean" name="ignore-annotations" use="optional"/>
        </xs:complexType>
        <xs:complexType name="annotationType">
            <xs:sequence>
                <xs:element type="map:elementType"
                            name="element"
                            maxOccurs="unbounded"
                            minOccurs="0"
                            xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="getterType">
            <xs:sequence>
                <xs:element type="xs:string" name="valid" minOccurs="0" fixed=""/>
                <xs:element type="map:constraintType"
                            name="constraint"
                            minOccurs="0"
                            maxOccurs="unbounded"
                            xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            </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="constraint-definitionType">
            <xs:sequence>
                <xs:element type="map:validated-byType"
                            name="validated-by"
                            xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            </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:constraintType"
                            name="constraint"
                            minOccurs="0"
                            maxOccurs="unbounded"
                            xmlns:map="http://jboss.org/xml/ns/javax/validation/mapping"/>
            </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>
```

7.2. 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           targetNamespace="http://jboss.org/xml/ns/javax/validation/configuration"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0">
    <xs:element name="validation-config" type="config:validation-configType" xmlns:config="http://jboss.o
    <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="constraint-mapping" maxOccurs="unbounded" minOccurs="0"/>
            <xs:element type="config:propertyType" name="property" maxOccurs="unbounded" minOccurs="0" xm
        </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:schema>
```

See Section 4.4.6 for more information on XML based configuration.

8

Exception model

Illegal arguments passed to the Bean Validation APIs generally lead to a `IllegalArgumentException` (see the JavaDoc for specific details). Other exceptions raised by Bean Validation are or inherit from the runtime exception `javax.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)

- 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`) 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.

8.1. 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.

Bean 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.

```java
/**
 * Reports the result of constraint violations
 *                                                  `
 * @author Emmanuel Bernard
 */
public class ConstraintViolationException extends ValidationException {
    private final Set<ConstraintViolation<?>> constraintViolations;

    /**
     * Creates a constraint violation report
     *
     * @param message error message
     * @param constraintViolations <code>Set</code> of <code>ConstraintViolation</code>
     */
    public ConstraintViolationException(String message,
                                        Set<ConstraintViolation<?>> constraintViolations) {
        super( message );
        this.constraintViolations = constraintViolations;
    }

    /**
     * Creates a constraint violation report
     *
     * @param constraintViolations <code>Set</code> of <code>ConstraintViolation</code>
     */
    public ConstraintViolationException(Set<ConstraintViolation<?>> constraintViolations) {
        super();
        this.constraintViolations = constraintViolations;
    }

    /**
     * Set of constraint violations reported during a validation
     *
     * @return <code>Set</code> of <code>ConstraintViolation</code>
     */
    public Set<ConstraintViolation<?>> getConstraintViolations() {
        return constraintViolations;
    }
}
```

The `ConstraintViolationException` carries a `Set` of `ConstraintViolation`.

Note

Bean Validation never raises this exception itself. Other frameworks like Java Persistence 2 do.

Note

If this exception is meant to be send remotely, `ConstraintViolation` objects should be `Serializable` as defined an explained in Section 4.2.

8.2. Constraint definition: `ConstraintDefinitionException`

If a constraint definition does not respect the Bean 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.

8.3. Constraint declaration: `ConstraintDeclarationException` and `UnexpectedTypeException`

When a constraint declaration is illegal, `ConstraintDeclarationException` is raised.

`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 arise when either no `ConstraintValidator` or too many `ConstraintValidators` match the return type (see Section 3.5.3).

`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.

8.4. Group definition: `GroupDefinitionException`

When a group definition is illegal, `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.

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.

Table A.1. terminology

Term	Definition
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 Bean 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	Define 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 pass a constraint or not.
Bean 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

Term	Definition
Constraint metadata API	API exposing the constraints applied to a given bean type. Also considered one of the integration points with other JSR or frameworks.
Bootstrap API	Bootstrapping part of the Bean Validation API producing a `ValidatorFactory`.
javax.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.
javax.validation.Validator	Main API. Holds contracts to validate object graphs
javax.validation.ConstraintViolation	Interface describing a given constraint failure on a given bean

B

Standard ResourceBundle messages

The properties listed below are resolved by the default message interpolator.

```
javax.validation.constraints.Null.message=must be null
javax.validation.constraints.NotNull.message=must not be null
javax.validation.constraints.AssertTrue.message=must be true
javax.validation.constraints.AssertFalse.message=must be false
javax.validation.constraints.Min.message=must be greater than or equal to {value}
javax.validation.constraints.Max.message=must be less than or equal to {value}
javax.validation.constraints.Size.message=size must be between {min} and {max}
javax.validation.constraints.Digits.message= \
    numeric value out of bounds (<{integer} digits>.<{fraction} digits> expected)
javax.validation.constraints.Past.message=must be a past date
javax.validation.constraints.Future.message=must be a future date
javax.validation.constraints.Pattern.message=must match the following regular expression: {regexp}
```

Proposal for method-level validation

This proposition has not been integrated into the core specification and is not part of it. It remains here for archaeological purposes and will be seriously considered for a future revision of this specification. This proposal is likely to be a bit out of sync with the rest of the specification artifacts.

Note

Bean Validation providers are free to implement this proposal as a specific extension. Such specific extension could for example be accessed via the use of the `Validator.unwrap` method.

A popular demand was to provide a method and parameter level validation mechanism reusing the constraint descriptions of the specification. This set of APIs is meant to be used by interceptor frameworks such as:

- application frameworks like JSR-299

- component frameworks like Enterprise Java Beans

- aspect based frameworks

These frameworks can call the validation APIs to validate either the parameter list or the returned value of a method when such method is called. More precisely, validation occurs around a method invocation.

This extension of the Bean Validation API allows to reuse the core engine as well as the constraint definition and declaration for such method level validations.

The following APIs are added to `Validator`.

```java
public interface Validator {
    // [...]


    /**
     * Validate each parameter value based on the constraints described on
     * the parameters of <code>method</code>.
     *
     * @param clazz class hosting the method
     * @param method the method whose parameters are currently validated
     * @param parameterValues the parameter values passed to the method for invocation
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the method does not belong to <code>T</code>
     *         or if the Object[] does not match the method signature
     */
    <T> Set<ConstraintViolation<T>> validateParameters(Class<T> clazz, Method method,
                                                       Object[] parameterValues,
                                                       Class<?>... groups);
```

```
    /**
     * Validate the parameter value based on the constraints described on
     * the parameterIndex-th parameter of <code>method</code>.
     *
     * @param clazz class hosting the method
     * @param method the method whose parameters are currently validated
     * @param parameterValue the parameter value passed to the parameterIndex-t parameter of method
     * @param parameterIndex parameter index of the parameter validated in method
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the method does not belong to <code>T</code>
     *          or if prameterIndex is out of bound
     */
    <T> Set<ConstraintViolation> validateParameter(Class<T> clazz, Method method,
                                                   Object parameterValue,
                                                   int parameterIndex, Class<?>... groups);


    /**
     * Validate each parameter value based on the constraints described on
     * <code>method</code>.
     *
     * @param clazz class hosting the method
     * @param method the method whose result is validated
     * @param returnedValue the value returned by the method invocation
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the method does not belong to <code>T</code>
     */
    <T> Set<ConstraintViolation> validateReturnedValue(Class<T> clazz, Method method,
                                                       Object returnedValue, Class<?>... groups);


    /**
     * Validate each parameter value based on the constraints described on
     * the parameters of <code>constructor</code>.
     *
     * @param clazz class hosting the constructor
     * @param constructor the constructor whose parameters are currently validated
     * @param parameterValues the parameter values passed to the constructor for invocation
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
     * @throws IllegalArgumentException if the constructor does not belong to <code>T</code>
     *          or if the Object[] does not match the constructor signature
     */
    <T> Set<ConstraintViolation> validateParameters(Class<T> clazz, Constructor constructor,
                                                    Object[] parameterValues, Class<?>... groups);


    /**
     * Validate the parameter value based on the constraints described on
     * the parameterIndex-th parameter of <code>constructor</code>.
     *
     * @param clazz class hosting the constructor
     * @param constructor the method whose parameters are currectly validated
     * @param parameterValue the parameter value passed to the
     *                       parameterIndex-th parameter of constructor
     * @param parameterIndex parameter index of the parameter validated in constructor
     * @param groups groups targeted for validation
     *
     * @return set of constraint violations
     *
```

```
      * @throws IllegalArgumentException if the constructor does not belong to <code>T</code>
      *          or if prameterIndex is out of bound
      */
     <T> Set<ConstraintViolation> validateParameter(Class<T> clazz, Constructor constructor,
                                                  Object parameterValue, int parameterIndex,
                                                  Class<?>... groups);
```

The constraints declarations evaluated are the constraints hosted on the parameters of the method or constructor. If `@Valid` is placed on a parameter, constraints declared on the object itself are considered.

`validateReturnedValue` evaluates the constraints hosted on the method itself. If `@Valid` is placed on the method, the constraints declared on the object itself are considered.

```
public @NotNull String saveItem(@Valid @NotNull Item item, @Max(23) BigDecimal price)
```

In the previous example,

- `item` is validated against `@NotNull` and all the constraints it hosts

- `price` is validated against `@Max(23)`

- the result of `saveItem` is validated against `@NotNull`

Note that the Bean Validation specification does not trigger the validation call. An external framework is responsible for calling one of the `validateParameters`, `validateParameter` and `validateReturnedValue` methods at the appropriate time.

For completeness, Bean Validation exposes metadata for constraints hosted on parameters and methods.

```
public interface ParameterDescriptor extends ElementDescriptor {
    boolean isCascaded();
    int getIndex();
}

public interface MethodDescriptor extends ElementDescriptor {
    List<ParameterDescriptor> getParameterDescriptors(); //index aligned
    boolean isCascaded();
}

public interface ConstructorDescriptor extends ElementDescriptor {
    List<ParameterDescriptor> getParameterDescriptors(); //index aligned
}

public interface BeanDescriptor {
    MethodDescriptor getConstraintsForMethod(Method);
    MethodDescriptor getConstraintsForConstructor(Constructor);
    Set<String> getConstrainedProperties();
    Set<Method> getConstrainedMethods();
    Set<Constructor> getConstrainedConstructors();
}
```

D

Java Persistence 2.0 integration

Integration with Java Persistence is described in the Java Persistence 2 specification (JSR-317). Persistence frameworks are encouraged to mimic the integration work done with Java Persistence.

While not specified by this specification or the Java Persistence 2.0 specification, Persistence Providers are encouraged to make use of Bean Validation constraint metadata when generating DDL schemas. The proposal is as followed.

```
Ideas explored and not standardized

Java Persistence consumes Bean Validation (BV) metadata to enhance persistence property
metadata.

A Persistence provider must use the BV metadata of a given list of groups.
The default group evaluated is Default (default BV 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 BV 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 BV annotation metadata should have priority over JPA metadata
(JPA 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 BV 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 recognize, the BV 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 BV error report. From the perspective of the application, constraint
errors are viewed through a unified layer. BV 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 BV metadata use by a persistence provider (property based).
```

This is not an endorsement of the Java Persistence expert group or the Bean Validation expert group. Such approach may nor may not be standardized in the future. Such integration should not be considered portable.

E

Java Server Faces 2.0 integration

Integration with Java Server Faces is described in the Java Server Faces 2 specification (JSR-314). Presentation frameworks are encouraged to study the integration work done with JSF 2.

F

Proposal for Java EE integration

This proposal is being discussed with the EE expert group. It proposes a way to smoothly integrates Bean Validation to the platform. This proposition is the result of the work done on JSF and JPA integrations.

Java Persistence and Java Server Faces have deep integration proposals with Bean Validation. Java EE Connector Architecture uses Bean Validation to validate particular artifacts.

TODO: This section is outdated. See the latest version in the Java EE 6 specification.

`Validator` and `ValidatorFactory` are very good candidates for injectable Java EE resources (`ValidatorFactory` is responsible for providing `Validator` instances which executes the validation logic). To tie everything together in a unified way for the Java EE application developer, some integration at the Java EE level would be beneficial.

A `ValidatorFactory` is built by the EE container and exposed to applications as well as services like JPA, JSF and Java EE Connectors (this imply a `ValidatorFactory` must be built and ready before these services). While not required, it is recommended to cache and share the same `ValidatorFactory` for a given deployment unit. *Should we mandate it?*.

The container passes the `ValidatorFactory` instance to the JPA provider via the configuration Map of `Persistence-Provider.createContainerEntityManagerFactory(PersistenceUnitInfo, Map)`. The property name is `javax.persistence.validation.factory` (`Persistence.VALIDATOR_FACTORY`).

The container places the `ValidatorFactory` instance under the following JNDI name. JSF 2 implementations are required to look in this entry for any `ValidatorFactory` and use it.

In addition, `Validator` and `ValidatorFactory` should be considered as Java EE resources: injection aware services should allow injection of `Validator` instances as well as `ValidatorFactory` instances. `@Resource` is used for this. If `Validator` instances are injected, the default validation context is used. In other words, `Validator` are provided by `validatorFactory.getValidator()`. The factory is available for the lifetime of the deployment.

G

Changelog

```
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 Configur
    * [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 ConstraintValidatorCon
    * [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 => getUnorderedConstraintDescr
    * [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<


** 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 preven
    * [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 valudateValu
    * [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, Prop
    * [BVAL-164] - Validation.byProvider now accept the provider implementation class
    * [BVAL-166] - IllegalArgumentException raised on BeanDescriptor.getConstraintsForProperty and Valida
    * [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 constr
    * [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 arra
    * [BVAL-114] - Relax property names in ConstraintValidatorContext
    * [BVAL-120] - Rename ConstraintViolation getRawMessage=>getMessageTemplate, getInterpolatedMessage=>
    * [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
     * [BVAL-127] - Remove ConstraintViolation.getGroups()
     * [BVAL-128] - Clarify invalid cases for validateProperty / validateValue on proeprtyName being empty
     * [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
     * [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 .getConstraintImp
     * [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
```