Hibernate Validator

JSR 349 Reference Implementation

Reference Guide

5.1.3.Final

by Hardy Ferentschik and Gunnar Morling

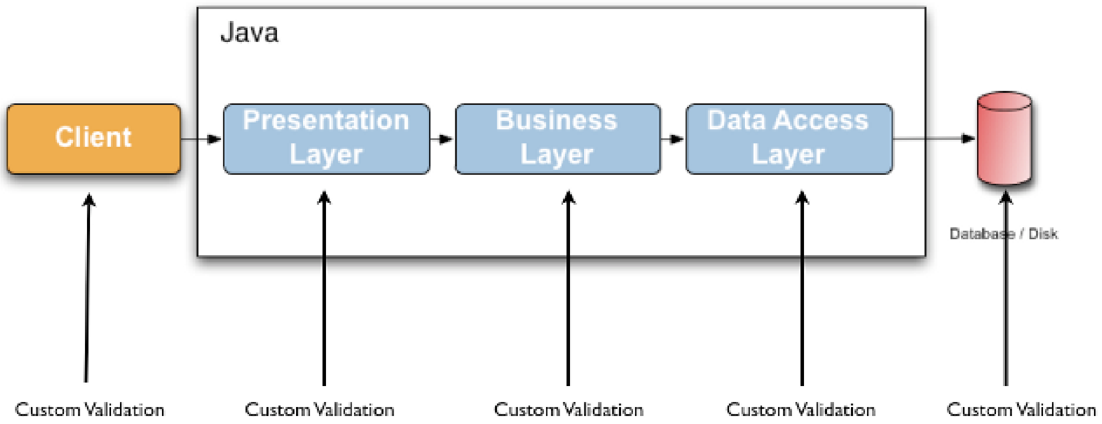
Preface vii
1. Getting started 1
1.1. Project set up 1
1.1.1. Unified EL 1
1.1.2. CDI 2
1.2. Applying constraints 2
1.3. Validating constraints 3
1.4. Where to go next? 5
2. Declaring and validating bean constraints 7
2.1. Declaring bean constraints 7
2.1.1. Field-level constraints 7
2.1.2. Property-level constraints 8
2.1.3. Class-level constraints 9
2.1.4. Constraint inheritance 10
2.1.5. Object graphs 11
2.2. Validating bean constraints 12
2.2.1. Obtaining a Validator instance 12
2.2.2. Validator methods 13
2.2.3. ConstraintViolation methods 14
2.3. Built-in constraints 15
2.3.1. Bean Validation constraints 15
2.3.2. Additional constraints 18
3. Declaring and validating method constraints 27
3.1. Declaring method constraints 28
3.1.1. Parameter constraints 28
3.1.2. Return value constraints 30
3.1.3. Cascaded validation 31
3.1.4. Method constraints in inheritance hierarchies 32
3.2. Validating method constraints 35
3.2.1. Obtaining an ExecutableValidator instance 35
3.2.2. ExecutableValidator methods 36
3.2.3. ConstraintViolation methods for method validation 39
3.3. Built-in method constraints 40
4. Interpolating constraint error messages 41
4.1. Default message interpolation 41
4.1.1. Special characters 42
4.1.2. Interpolation with message expressions 42
4.1.3. Examples 43
4.2. Custom message interpolation 45
4.2.1. ResourceBundleLocator 45
5. Grouping constraints 47
5.1. Requesting groups 47
5.2. Defining group sequences 51
5.3. Redefining the default group sequence 52

5.3.1. @GroupSequence 52
5.3.2. @GroupSequenceProvider 53
5.4. Group conversion 54
6. Creating custom constraints 59
6.1. Creating a simple constraint 59
6.1.1. The constraint annotation 59
6.1.2. The constraint validator 61
6.1.3. The error message 64
6.1.4. Using the constraint 64
6.2. Class-level constraints 65
6.2.1. Custom property paths 66
6.3. Cross-parameter constraints 67
6.4. Constraint composition 70
7. Configuring via XML 73
7.1. Configuring the validator factory in validation.xml 73
7.2. Mapping constraints via constraint-mappings 76
8. Bootstrapping 83
8.1. Retrieving ValidatorFactory and Validator 83
8.1.1. ValidationProviderResolver 84
8.2. Configuring a ValidatorFactory 85
8.2.1. MessageInterpolator 85
8.2.2. TraversableResolver 86
8.2.3. ConstraintValidatorFactory 87
8.2.4. ParameterNameProvider 88
8.2.5. Adding mapping streams 89
8.2.6. Provider-specific settings 89
8.3. Configuring a Validator 90
9. Using constraint metadata 91
9.1. BeanDescriptor 92
9.2. PropertyDescriptor 95
9.3. MethodDescriptor and ConstructorDescriptor 95
9.4. ElementDescriptor 97
9.5. GroupConversionDescriptor 100
9.6. ConstraintDescriptor 100
10. Integrating with other frameworks 103
10.1. ORM integration 103
10.1.1. Database schema-level validation 103
10.1.2. Hibernate event-based validation 103
10.1.3. JPA 104
10.2. JSF & Seam 105
10.3. CDI 106
10.3.1. Dependency injection 106
10.3.2. Method validation 108
10.4. Java EE 111

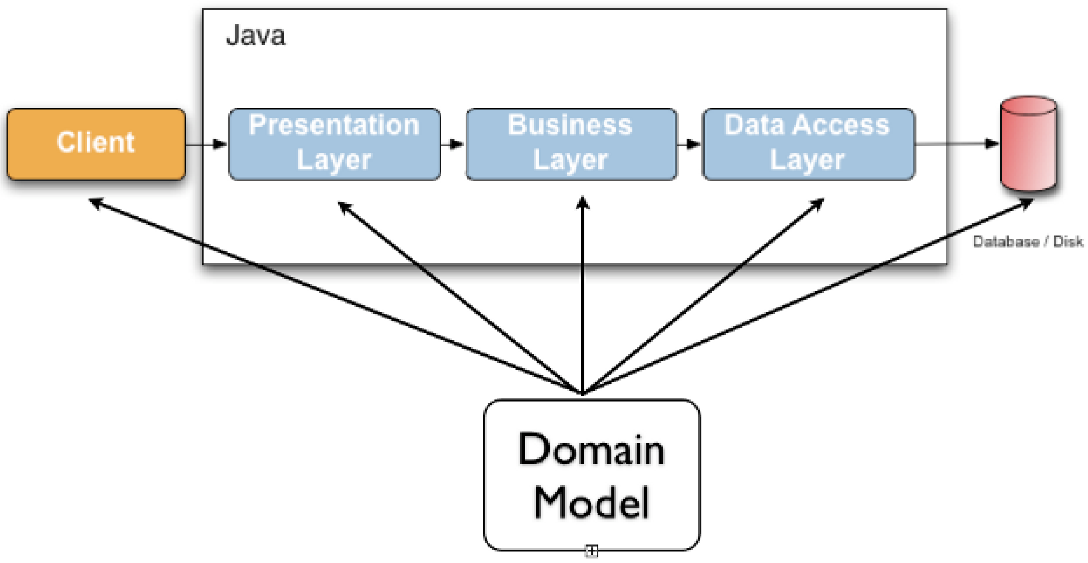
11. Hibernate Validator Specifics 113
11.1. Public API 113
11.2. Fail fast mode 114
11.3. Programmatic constraint declaration 116
11.4. Boolean composition of constraints 119
11.5. ResourceBundleLocator 119
11.6. Custom contexts 120
11.6.1. HibernateConstraintValidatorContext 120
11.6.2. HibernateMessageInterpolatorContext 121
11.7. ParaNamer based ParameterNameProvider 121
11.8. Unwrapping values prior to validation 122
12. Annotation Processor 125
12.1. Prerequisites 125
12.2. Features 125
12.3. Options 126
12.4. Using the Annotation Processor 126
12.4.1. Command line builds 126
12.4.2. IDE builds 128
12.5. Known issues 131
13. Further reading 133

Preface

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



JSR 349 - Bean Validation 1.1 - defines a metadata model and API for entity and method validation. The default metadata source are annotations, with the ability to override and extend the meta-data through the use of XML. The API is not tied to a specific application tier nor programming model. It is specifically not tied to either web or persistence tier, and is available for both server-side application programming, as well as rich client Swing application developers.



Hibernate Validator is the reference implementation of this JSR 349. The implementation itself as well as the Bean Validation API and TCK are all provided and distributed under the *Apache Software License 2.0* [http://www.apache.org/licenses/LICENSE-2.0].

Getting started

This chapter will show you how to get started with Hibernate Validator, the reference implementation (RI) of Bean Validation. For the following quickstart you need:

- A JDK >= 6
- *Apache Maven* [http://maven.apache.org/]
- An Internet connection (Maven has to download all required libraries)

1.1. Project set up

In order to use Hibernate Validator within a Maven project, simply add the following dependency to your `pom.xml`:

Example 1.1. Hibernate Validator Maven dependency

```
<dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-validator</artifactId>
                <version>5.1.3.Final</version>
                </dependency>
```

This transitively pulls in the dependency to the Bean Validation API (`javax.validation:validation-api:1.1.0.Final`).

1.1.1. Unified EL

Hibernate Validator requires an implementation of the Unified Expression Language (*JSR 341* [jcp.org/en/jsr/detail?id=341]) for evaluating dynamic expressions in constraint violation messages (see*Section 4.1, "Default message interpolation"*). When your application runs in a Java EE container such as JBoss AS, an EL implementation is already provided by the container. In a Java SE environment, however, you have to add an implementation as dependency to your POM file. For instance you can add the following two dependencies to use the JSR 341*reference implementation* [http://uel.java.net/]:

Example 1.2. Maven dependencies for Unified EL reference implementation

```
<dependency>
                    <groupId>javax.el</groupId>
                    <artifactId>javax.el-api</artifactId>
                    <version>2.2.4</version>
                    </dependency>
                    <dependency>
                    <groupId>org.glassfish.web</groupId>
                    <artifactId>javax.el</artifactId>
```

```
                    <version>2.2.4</version>
                    </dependency>
```

1.1.2. CDI

Bean Validation defines integration points with CDI (Contexts and Dependency Injection for Java TM EE,*JSR 346* [http://jcp.org/en/jsr/detail?id=346]). If your application runs in an environment which does not provide this integration out of the box, you may use the Hibernate Validator CDI portable extension by adding the following Maven dependency to your POM:

Example 1.3. Hibernate Validator CDI portable extension Maven dependency

```
<dependency>
                    <groupId>org.hibernate</groupId>
                    <artifactId>hibernate-validator-cdi</artifactId>
                    <version>5.1.3.Final</version>
                    </dependency>
```

Note that adding this dependency is usually not required for applications running on a Java EE application server. You can learn more about the integration of Bean Validation and CDI in*Section 10.3, “CDI”*.

1.2. Applying constraints

Lets dive directly into an example to see how to apply constraints.

Example 1.4. Class `Car` annotated with constraints

```
package org.hibernate.validator.referenceguide.chapter01;

                import javax.validation.constraints.Min;
                import javax.validation.constraints.NotNull;
                import javax.validation.constraints.Size;

                public class Car {

                @NotNull
                private String manufacturer;

                @NotNull
                @Size(min = 2, max = 14)
                private String licensePlate;

                @Min(2)
                private int seatCount;

                public Car(String manufacturer, String licencePlate, int seatCount) {
                this.manufacturer = manufacturer;
```

```
                this.licensePlate = licencePlate;
                this.seatCount = seatCount;
                }

                //getters and setters ...
                }
```

The@NotNull, @Size and @Min annotations are used to declare the constraints which should be applied to the fields of a Car instance:

- manufacturer must never be null
- licensePlate must never be null and must be between 2 and 14 characters long
- seatCount must be at least 2

> **Tip**
>
> You can find the complete source code of all examples used in this reference guide in the Hibernate Validator *source repository* [https://github.com/hibernate/hibernate-validator/tree/master/documentation/src/test] on GitHub.

1.3. Validating constraints

To perform a validation of these constraints, you use a `Validator` instance. Let's have a look at a unit test for`Car`:

Example 1.5. Class `CarTest` showing validation examples

```
package org.hibernate.validator.referenceguide.chapter01;

                import java.util.Set;
                import javax.validation.ConstraintViolation;
                import javax.validation.Validation;
                import javax.validation.Validator;
                import javax.validation.ValidatorFactory;

                import org.junit.BeforeClass;
                import org.junit.Test;

                import static org.junit.Assert.assertEquals;

                public class CarTest {

                private static Validator validator;

                @BeforeClass
                public static void setUp() {
                ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
                validator = factory.getValidator();
```

```
}

@Test
public void manufacturerIsNull() {
Car car = new Car( null, "DD-AB-123", 4 );

Set<ConstraintViolation<Car>> constraintViolations =
validator.validate( car );

assertEquals( 1, constraintViolations.size() );
                                             assertEquals(    "may    not    be
null", constraintViolations.iterator().next().getMessage() );
}

@Test
public void licensePlateTooShort() {
Car car = new Car( "Morris", "D", 4 );

Set<ConstraintViolation<Car>> constraintViolations =
validator.validate( car );

assertEquals( 1, constraintViolations.size() );
assertEquals(
"size must be between 2 and 14",
constraintViolations.iterator().next().getMessage()
);
}

@Test
public void seatCountTooLow() {
Car car = new Car( "Morris", "DD-AB-123", 1 );

Set<ConstraintViolation<Car>> constraintViolations =
validator.validate( car );

assertEquals( 1, constraintViolations.size() );
assertEquals(
"must be greater than or equal to 2",
constraintViolations.iterator().next().getMessage()
);
}

@Test
public void carIsValid() {
Car car = new Car( "Morris", "DD-AB-123", 2 );

Set<ConstraintViolation<Car>> constraintViolations =
validator.validate( car );

assertEquals( 0, constraintViolations.size() );
}
}
```

In the `setUp()` method a `Validator` object is retrieved from the `ValidatorFactory`. A `Validator` instance is thread-safe and may be reused multiple times. It thus can safely be stored in a static field and be used in the test methods to validate the different `Car` instances.

The `validate()` method returns a set of `ConstraintViolation` instances, which you can iterate over in order to see which validation errors occurred. The first three test methods show some expected constraint violations:

- The `@NotNull` constraint on manufacturer is violated in `manufacturerIsNull()`
- The `@Size` constraint on licensePlate is violated in `licensePlateTooShort()`
- The `@Min` constraint on seatCount is violated in `seatCountTooLow()`

If the object validates successfully, `validate()` returns an empty set as you can see in `carIsValid()`.

Note that only classes from the package javax.validation are used. These are provided from the Bean Validation API. No classes from Hibernate Validator are directly referenced, resulting in portable code.

1.4. Where to go next?

That concludes the 5 minute tour through the world of Hibernate Validator and Bean Validation. Continue exploring the code examples or look at further examples referenced in*Chapter 13, Further reading*.

To learn more about the validation of beans and properties, just continue reading*Chapter 2, Declaring and validating bean constraints*. If you are interested in using Bean Validation for the validation of method pre- and postcondition refer to*Chapter 3, Declaring and validating method constraints*. In case your application has specific validation requirements have a look at *Chapter 6, Creating custom constraints*.

Declaring and validating bean constraints

In this chapter you will learn how to declare (see *Section 2.1, "Declaring bean constraints"*) and validate (see *Section 2.2, "Validating bean constraints"*) bean constraints. *Section 2.3, "Built-in constraints"* provides an overview of all built-in constraints coming with Hibernate Validator.

If you are interested in applying constraints to method parameters and return values, refer to *Chapter 3, Declaring and validating method constraints*.

2.1. Declaring bean constraints

Constraints in Bean Validation are expressed via Java annotations. In this section you will learn how to enhance an object model with these annotations. There are the following three types of bean constraints:

- field constraints
- property constraints
- class constraints

> **Note**
>
> Not all constraints can be placed on all of these levels. In fact, none of the default constraints defined by Bean Validation can be placed at class level. The `java.lang.annotation.Target` annotation in the constraint annotation itself determines on which elements a constraint can be placed. See *Chapter 6, Creating custom constraints* for more information.

2.1.1. Field-level constraints

Constraints can be expressed by annotating a field of a class. *Example 2.1, "Field-level constraints"* shows a field level configuration example:

Example 2.1. Field-level constraints

```
package org.hibernate.validator.referenceguide.chapter02.fieldlevel;

public class Car {

    @NotNull
    private String manufacturer;

    @AssertTrue
```

```
    private boolean isRegistered;

    public Car(String manufacturer, boolean isRegistered) {
        this.manufacturer = manufacturer;
        this.isRegistered = isRegistered;
    }

    //getters and setters...
}
```

When using field-level constraints field access strategy is used to access the value to be validated. This means the validation engine directly accesses the instance variable and does not invoke the property accessor method even if such an accessor exists.

Constraints can be applied to fields of any access type (public, private etc.). Constraints on static fields are not supported, though.

> **Tip**
>
> When validating byte code enhanced objects property level constraints should be used, because the byte code enhancing library won't be able to determine a field access via reflection.

2.1.2. Property-level constraints

If your model class adheres to the *JavaBeans* [http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp] standard, it is also possible to annotate the properties of a bean class instead of its fields. *Example 2.2, “Property-level constraints”* uses the same entity as in *Example 2.1, “Field-level constraints”*, however, property level constraints are used.

Example 2.2. Property-level constraints

```
package org.hibernate.validator.referenceguide.chapter02.propertylevel;

public class Car {

    private String manufacturer;

    private boolean isRegistered;

    public Car(String manufacturer, boolean isRegistered) {
        this.manufacturer = manufacturer;
        this.isRegistered = isRegistered;
    }

    @NotNull
    public String getManufacturer() {
        return manufacturer;
    }
```

```
    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

    @AssertTrue
    public boolean isRegistered() {
        return isRegistered;
    }

    public void setRegistered(boolean isRegistered) {
        this.isRegistered = isRegistered;
    }
}
```

Note

The property's getter method has to be annotated, not its setter. That way also read-only properties can be constrained which have no setter method.

When using property level constraints property access strategy is used to access the value to be validated, i.e. the validation engine accesses the state via the property accessor method.

Tip

It is recommended to stick either to field *or* property annotations within one class. It is not recommended to annotate a field *and* the accompanying getter method as this would cause the field to be validated twice.

2.1.3. Class-level constraints

Last but not least, a constraint can also be placed on the class level. In this case not a single property is subject of the validation but the complete object. Class-level constraints are useful if the validation depends on a correlation between several properties of an object.

The `Car` class in *Example 2.3, "Class-level constraint"* has the two attributes `seatCount` and `passengers` and it should be ensured that the list of passengers has not more entries than seats are available. For that purpose the `@ValidPassengerCount` constraint is added on the class level. The validator of that constraint has access to the complete `Car` object, allowing to compare the numbers of seats and passengers.

Refer to *Section 6.2, "Class-level constraints"* to learn in detail how to implement this custom constraint.

Example 2.3. Class-level constraint

```
package org.hibernate.validator.referenceguide.chapter02.classlevel;
```

```
@ValidPassengerCount
public class Car {

    private int seatCount;

    private List<Person> passengers;

    //...
}
```

2.1.4. Constraint inheritance

When a class implements an interface or extends another class, all constraint annotations declared on the supertype apply in the same manner as the constraints specified on the class itself. To make things clearer let's have a look at the following example:

Example 2.4. Constraint inheritance

```
package org.hibernate.validator.referenceguide.chapter02.inheritance;

public class Car {

    private String manufacturer;

    @NotNull
    public String getManufacturer() {
        return manufacturer;
    }

    //...
}
```

```
package org.hibernate.validator.referenceguide.chapter02.inheritance;

public class RentalCar extends Car {

    private String rentalStation;

    @NotNull
    public String getRentalStation() {
        return rentalStation;
    }

    //...
}
```

Here the class `RentalCar` is a subclass of `Car` and adds the property rentalStation. If an instance of `RentalCar` is validated, not only the `@NotNull` constraint on rentalStation is evaluated, but also the constraint on manufacturer from the parent class.

The same would be true, if `Car` was not a superclass but an interface implemented by `RentalCar`.

Constraint annotations are aggregated if methods are overridden. So if `RentalCar` overrode the `getManufacturer()` method from `Car`, any constraints annotated at the overriding method would be evaluated in addition to the `@NotNull` constraint from the superclass.

2.1.5. Object graphs

The Bean Validation API does not only allow to validate single class instances but also complete object graphs (cascaded validation). To do so, just annotate a field or property representing a reference to another object with `@Valid` as demonstrated in *Example 2.5, “Cascaded validation”*.

Example 2.5. Cascaded validation

```
package org.hibernate.validator.referenceguide.chapter02.objectgraph;

public class Car {

    @NotNull
    @Valid
    private Person driver;

    //...
}
```

```
package org.hibernate.validator.referenceguide.chapter02.objectgraph;

public class Person {

    @NotNull
    private String name;

    //...
}
```

If an instance of `Car` is validated, the referenced `Person` object will be validated as well, as the driver field is annotated with `@Valid`. Therefore the validation of a `Car` will fail if the name field of the referenced `Person` instance is `null`.

The validation of object graphs is recursive, i.e. if a reference marked for cascaded validation points to an object which itself has properties annotated with `@Valid`, these references will be followed up by the validation engine as well. The validation engine will ensure that no infinite loops occur during cascaded validation, for example if two objects hold references to each other.

Note that `null` values are getting ignored during cascaded validation.

Object graph validation also works for collection-typed fields. That means any attributes that

- are arrays

- implement `java.lang.Iterable` (especially `Collection`, `List` and `Set`)
- implement `java.util.Map`

can be annotated with `@Valid`, which will cause each contained element to be validated, when the parent object is validated.

Example 2.6. Cascaded validation of a collection

```
package org.hibernate.validator.referenceguide.chapter02.objectgraph.list;

public class Car {

    @NotNull
    @Valid
    private List<Person> passengers = new ArrayList<Person>();

    //...
}
```

So when validating an instance of the `Car` class shown in *Example 2.6, “Cascaded validation of a collection”*, a `ConstraintViolation` will be created, if any of the `Person` objects contained in the passengers list has a `null` name.

2.2. Validating bean constraints

The `Validator` interface is the most important object in Bean Validation. The next section shows how to obtain an `Validator` instance. Afterwards you'll learn how to use the different methods of the `Validator` interface.

2.2.1. Obtaining a `Validator` instance

The first step towards validating an entity instance is to get hold of a `Validator` instance. The road to this instance leads via the `Validation` class and a `ValidatorFactory`. The easiest way is to use the static method `Validation#buildDefaultValidatorFactory()`:

Example 2.7. Validation#buildDefaultValidatorFactory()

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

This bootstraps a validator in the default configuration. Refer to *Chapter 8, Bootstrapping* to learn more about the different bootstrapping methods and how to obtain a specifically configured `Validator` instance.

2.2.2. Validator methods

The `Validator` interface contains three methods that can be used to either validate entire entities or just single properties of the entity.

All three methods return a `Set<ConstraintViolation>`. The set is empty, if the validation succeeds. Otherwise a `ConstraintViolation` instance is added for each violated constraint.

All the validation methods have a var-args parameter which can be used to specify, which validation groups shall be considered when performing the validation. If the parameter is not specified the default validation group (`javax.validation.groups.Default`) is used. The topic of validation groups is discussed in detail in *Chapter 5, Grouping constraints*.

2.2.2.1. `Validator#validate()`

Use the `validate()` method to perform validation of all constraints of a given bean. *Example 2.8, “Using Validator#validate()”* shows the validation of an instance of the `Car` class from *Example 2.2, “Property-level constraints”* which fails to satisfy the `@NotNull` constraint on the `manufacturer` property. The validation call therefore returns one `ConstraintViolation` object.

Example 2.8. Using `Validator#validate()`

```
Car car = new Car( null, true );

Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );

assertEquals( 1, constraintViolations.size() );
assertEquals( "may not be null", constraintViolations.iterator().next().getMessage() );
```

2.2.2.2. `Validator#validateProperty()`

With help of the `validateProperty()` you can validate a single named property of a given object. The property name is the JavaBeans property name.

Example 2.9. Using `Validator#validateProperty()`

```
Car car = new Car( null, true );

Set<ConstraintViolation<Car>> constraintViolations = validator.validateProperty(
        car,
        "manufacturer"
);

assertEquals( 1, constraintViolations.size() );
assertEquals( "may not be null", constraintViolations.iterator().next().getMessage() );
```

2.2.2.3. `Validator#validateValue()`

By using the `validateValue()` method you can check whether a single property of a given class can be validated successfully, if the property had the specified value:

Example 2.10. Using `Validator#validateValue()`

```
Set<ConstraintViolation<Car>> constraintViolations = validator.validateValue(
        Car.class,
        "manufacturer",
        null
);

assertEquals( 1, constraintViolations.size() );
assertEquals( "may not be null", constraintViolations.iterator().next().getMessage() );
```

> **Note**
>
> `@Valid` is not honored by `validateProperty()` or `validateValue()`.

`Validator#validateProperty()` is for example used in the integration of Bean Validation into JSF 2 (see *Section 10.2, “JSF & Seam”*) to perform a validation of the values entered into a form before they are propagated to the model.

2.2.3. `ConstraintViolation` methods

Now it is time to have a closer look at what a `ConstraintViolation` is. Using the different methods of `ConstraintViolation` a lot of useful information about the cause of the validation failure can be determined. *Table 2.1, “The various ConstraintViolation methods”* gives an overview of these methods. The values in the "Example" column refer to *Example 2.8, “Using Validator#validate()”*.

Table 2.1. The various `ConstraintViolation` methods

Method	Usage	Example
`getMessage()`	The interpolated error message	"may not be null"
`getMessageTemplate()`	The non-interpolated error message	"{... NotNull.message}"
`getRootBean()`	The root bean being validated	`car`
`getRootBeanClass()`	The class of the root bean being validated	`Car.class`
`getLeafBean()`	If a bean constraint, the bean instance the constraint is applied on; If a property	`car`

Method	Usage	Example
	constraint, the bean instance hosting the property the constraint is applied on	
`getPropertyPath()`	The property path to the validated value from root bean	contains one node with kind `PROPERTY` and name "manufacturer"
`getInvalidValue()`	The value failing to pass the constraint	`null`
`getConstraintDescriptor()`	Constraint metadata reported to fail	descriptor for `@NotNull`

2.3. Built-in constraints

Hibernate Validator comprises a basic set of commonly used constraints. These are foremost the constraints defined by the Bean Validation specification (see *Table 2.2, "Bean Validation constraints"*). Additionally, Hibernate Validator provides useful custom constraints (see *Table 2.3, "Custom constraints"* and *Table 2.4, "Custom country specific constraints"*).

2.3.1. Bean Validation constraints

Table 2.2, "Bean Validation constraints" shows purpose and supported data types of all constraints specified in the Bean Validation API. All these constraints apply to the field/property level, there are no class-level constraints defined in the Bean Validation specification. If you are using the Hibernate object-relational mapper, some of the constraints are taken into account when creating the DDL for your model (see column "Hibernate metadata impact").

Note

Hibernate Validator allows some constraints to be applied to more data types than required by the Bean Validation specification (e.g. `@Max` can be applied to `Strings`). Relying on this feature can impact portability of your application between Bean Validation providers.

Table 2.2. Bean Validation constraints

Annotation	Supported data types	Use	Hibernate metadata impact
`@AssertFalse`	`Boolean`, `boolean`	Checks that the annotated element is `false`	None
`@AssertTrue`	`Boolean`, `boolean`	Checks that the annotated element is `true`	None

Annotation	Supported data types	Use	Hibernate metadata impact
`@DecimalMax(value=, inclusive=)`	`BigDecimal`, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types; Additionally supported by HV: any sub-type of `Number`	Checks whether the annotated value is less than the specified maximum, when *`inclusive=false`*. Otherwise whether the value is less than or equal to the specified maximum. The parameter *`value`* is the string representation of the max value according to the `BigDecimal` string representation.	None
`@DecimalMin(value=, inclusive=)`	`BigDecimal`, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types; Additionally supported by HV: any sub-type of `Number`	Checks whether the annotated value is larger than the specified minimum, when *`inclusive=false`*. Otherwise whether the value is larger than or equal to the specified minimum. The parameter *`value`* is the string representation of the min value according to the `BigDecimal` string representation.	None
`@Digits(integer=, fraction=)`	`BigDecimal`, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types; Additionally supported by HV: any sub-type of `Number`	Checks whether the annoted value is a number having up to `integer` digits and `fraction` fractional digits	Defines column precision and scale
`@Future`	`java.util.Date`, `java.util.Calendar`; Additionally supported by HV, if the *Joda Time* [http://joda-time.sourceforge.net/] date/time API is on the class path: any implementations of `ReadablePartial` and `ReadableInstant`	Checks whether the annotated date is in the future	None

Annotation	Supported data types	Use	Hibernate metadata impact
`@Max(value=)`	`BigDecimal`, `BigInteger`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types; Additionally supported by HV: any sub-type of `CharSequence` (the numeric value represented by the character sequence is evaluated), any sub-type of `Number`	Checks whether the annotated value is less than or equal to the specified maximum	Adds a check constraint on the column
`@Min(value=)`	`BigDecimal`, `BigInteger`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types; Additionally supported by HV: any sub-type of `CharSequence` (the numeric value represented by the char sequence is evaluated), any sub-type of `Number`	Checks whether the annotated value is higher than or equal to the specified minimum	Adds a check constraint on the column
`@NotNull`	Any type	Checks that the annotated value is not `null.`	Column(s) are not nullable
`@Null`	Any type	Checks that the annotated value is `null`	None
`@Past`	`java.util.Date`, `java.util.Calendar`; Additionally supported by HV, if the *Joda Time* [http://joda-time.sourceforge.net/] date/time API is on the class path: any implementations of `ReadablePartial` and `ReadableInstant`	Checks whether the annotated date is in the past	None
`@Pattern(regex=, flag=)`	`CharSequence`	Checks if the annotated string matches the regular expression *`regex`* considering the given flag *`match`*	None

Annotation	Supported data types	Use	Hibernate metadata impact
`@Size(min=, max=)`	`CharSequence`, `Collection`, `Map` and `arrays`	Checks if the annotated element's size is between min and max (inclusive)	Column length will be set to *max*
`@Valid`	Any non-primitive type	Performs validation recursively on the associated object. If the object is a collection or an array, the elements are validated recursively. If the object is a map, the value elements are validated recursively.	None

> **Note**
>
> On top of the parameters indicated in *Table 2.2, "Bean Validation constraints"* each constraint has the parameters *`message`*, *`groups`* and *`payload`*. This is a requirement of the Bean Validation specification.

2.3.2. Additional constraints

In addition to the constraints defined by the Bean Validation API Hibernate Validator provides several useful custom constraints which are listed in *Table 2.3, "Custom constraints"*. With one exception also these constraints apply to the field/property level, only `@ScriptAssert` is a class-level constraint.

Table 2.3. Custom constraints

Annotation	Supported data types	Use	Hibernate metadata impact
`@CreditCardNumber(ignoreNonDigitCharacters=)`	`CharSequence`	Checks that the annotated character sequence passes the Luhn checksum test. Note, this validation aims to check for user mistakes, not credit card validity! See also *Anatomy of Credit Card Numbers* [http://www.merriampark.com/	None

Annotation	Supported data types	Use	Hibernate metadata impact
		anatomycc.htm]. *ignoreNonDigitCharacters* allows to ignore non digit characters. The default is `false`.	
`@EAN`	`CharSequence`	Checks that the annotated character sequence is a valid *EAN* [http://en.wikipedia.org/wiki/International_Article_Number_%28EAN%29] barcode. `type` determines the type of barcode. The default is EAN-13.	None
`@Email`	`CharSequence`	Checks whether the specified character sequence is a valid email address. The optional parameters *regexp* and *flags* allow to specify an additional regular expression (including regular expression flags) which the email must match.	None
`@Length(min=, max=)`	`CharSequence`	Validates that the annotated character sequence is between *min* and *max* included	Column length will be set to *max*
`@LuhnCheck(startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=)`	`CharSequence`	Checks that the digits within the annotated character sequence pass the Luhn checksum algorithm (see also *Luhn algorithm* [http://en.wikipedia.org/wiki/Luhn_algorithm]).	None

Annotation	Supported data types	Use	Hibernate metadata impact
		`startIndex` and `endIndex` allow to only run the algorithm on the specified sub-string. `checkDigitIndex` allows to use an arbitrary digit within the character sequence as the check digit. If not specified it is assumed that the check digit is part of the specified range. Last but not least, `ignoreNonDigitCharacters` allows to ignore non digit characters.	
`@Mod10Check(multiplier=, weight=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=)`	`CharSequence`	Checks that the digits within the annotated character sequence pass the generic mod 10 checksum algorithm. `multiplier` determines the multiplier for odd numbers (defaults to 3), `weight` the weight for even numbers (defaults to 1). `startIndex` and `endIndex` allow to only run the algorithm on the specified sub-string. `checkDigitIndex` allows to use an arbitrary digit within the character sequence as the	None

Annotation	Supported data types	Use	Hibernate metadata impact
		check digit. If not specified it is assumed that the check digit is part of the specified range. Last but not least, `ignoreNonDigitCharacters` allows to ignore non digit characters.	
`@Mod11Check(threshold=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=, treatCheck10As=, treatCheck11As=)`	`CharSequence`	Checks that the digits within the annotated character sequence pass the mod 11 checksum algorithm. `threshold` specifies the threshold for the mod11 multiplier growth; if no value is specified the multiplier will grow indefinitely. `treatCheck10As` and `treatCheck11As` specify the check digits to be used when the mod 11 checksum equals 10 or 11, respectively. Default to X and 0, respectively. `startIndex`, `endIndex` `acheckDigitIndex` and `ignoreNonDigitCharacters` carry the same semantics as in `@Mod10Check`.	None
`@NotBlank`	`CharSequence`	Checks that the annotated character sequence is not null	None

Annotation	Supported data types	Use	Hibernate metadata impact
		and the trimmed length is greater than 0. The difference to `@NotEmpty` is that this constraint can only be applied on strings and that trailing whitespaces are ignored.	
`@NotEmpty`	`CharSequence`, `Collection`, `Map` and arrays	Checks whether the annotated element is not `null` nor empty	None
`@Range(min=, max=)`	`BigDecimal`, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types	Checks whether the annotated value lies between (inclusive) the specified minimum and maximum	None
`@SafeHtml(whitelistType=, additionalTags=, additionalTagsWithAttributes=)`	`CharSequence`	Checks whether the annotated value contains potentially malicious fragments such as `<script/>`. In order to use this constraint, the *jsoup* [http://jsoup.org/] library must be part of the class path. With the *`whitelistType`* attribute a predefined whitelist type can be chosen which can be refined via `additionalTags` or `additionalTagsWithAttributes`. The former allows to add tags without any attributes, whereas the latter	None

Annotation	Supported data types	Use	Hibernate metadata impact
		allows to specify tags and optionally allowed attributes using the annotation `@SafeHtml.Tag`.	
`@ScriptAssert(lang=, script=, alias=)`	Any type	Checks whether the given script can successfully be evaluated against the annotated element. In order to use this constraint, an implementation of the Java Scripting API as defined by JSR 223 ("Scripting for the Java^{TM} Platform") must part of the class path. The expressions to be evaluated can be written in any scripting or expression language, for which a JSR 223 compatible engine can be found in the class path.	None
`@URL(protocol=, host=, port= regexp=, flags=)`	`CharSequence`	Checks if the annotated character sequence is a valid URL according to RFC2396. If any of the optional parameters *`protocol`*, *`host`* or *`port`* are specified, the corresponding URL fragments must match the specified values. The optional	None

Annotation	Supported data types	Use	Hibernate metadata impact
		parameters *regexp* and *flags* allow to specify an additional regular expression (including regular expression flags) which the URL must match.	

2.3.2.1. Country specific constraints

Hibernate Validator offers also some country specific constraints, e.g. for the validation of social security numbers.

> **Note**
>
> If you have to implement a country specific constraint, consider making it a contribution to Hibernate Validator!

Table 2.4. Custom country specific constraints

Annotation	Supported data types	Use	Country	Hibernate metadata impact
`@CNPJ`	`CharSequence`	Checks that the annotated character sequence represents a Brazilian corporate tax payer registry number (Cadastro de Pessoa Juríeddica)	Brazil	None
`@CPF`	`CharSequence`	Checks that the annotated character sequence represents	Brazil	None

Annotation	Supported data types	Use	Country	Hibernate metadata impact
		a Brazilian individual taxpayer registry number (Cadastro de Pessoa Fídsica)		
`@TituloEleitoral`	`CharSequence`	Checks that the annotated character sequence represents a Brazilian voter ID card number (*Título Eleitoral* [http://ghiorzi.org/cgcancpf.htm])	Brazil	None

Tip

In some cases neither the Bean Validation constraints nor the custom constraints provided by Hibernate Validator will fulfill your requirements. In this case you can easily write your own constraint. You can find more information in *Chapter 6, Creating custom constraints*.

Declaring and validating method constraints

As of Bean Validation 1.1, constraints can not only be applied to JavaBeans and their properties, but also to the parameters and return values of the methods and constructors of any Java type. That way Bean Validation constraints can be used to specify

- the preconditions that must be satisfied by the caller before a method or constructor may be invoked (by applying constraints to the parameters of an executable)
- the postconditions that are guaranteed to the caller after a method or constructor invocation returns (by applying constraints to the return value of an executable)

Note

For the purpose of this reference guide, the term *method constraint* refers to both, method and constructor constraints, if not stated otherwise. Ocassionally, the term *executable* is used when referering to methods and constructors.

This approach has several advantages over traditional ways of checking the correctness of parameters and return values:

- the checks don't have to be performed manually (e.g. by throwing `IllegalArgumentExceptions` or similar), resulting in less code to write and maintain
- an executable's pre- and postconditions don't have to be expressed again in its documentation, since the constraint annotations will automatically be included in the generated JavaDoc. This avoids redundancies and reduces the chance of inconsistencies between implementation and documentation

Tip

In order to make annotations show up in the JavaDoc of annoted elements, the annotation types themselves must be annotated with the meta annotation `@Documented`. This is the case for all built-in constraints and is considered a best practice for any custom constraints.

In the remainder of this chapter you will learn how to declare parameter and return value constraints and how to validate them using the `ExecutableValidator` API.

3.1. Declaring method constraints

3.1.1. Parameter constraints

You specify the preconditions of a method or constructor by adding constraint annotations to its parameters as demonstrated in *Example 3.1, “Declaring method and constructor parameter constraints”*.

Example 3.1. Declaring method and constructor parameter constraints

```
package org.hibernate.validator.referenceguide.chapter03.parameter;

public class RentalStation {

    public RentalStation(@NotNull String name) {
        //...
    }

    public void rentCar(
            @NotNull Customer customer,
            @NotNull @Future Date startDate,
            @Min(1) int durationInDays) {
        //...
    }
}
```

The following preconditions are declared here:

- The name passed to the `RentalCar` constructor must not be `null`
- When invoking the `rentCar()` method, the given customer must not be `null`, the rental's start date must not be `null` and must be in the future and the rental duration must be at least one day

Note that declaring method or constructor constraints itself does not automatically cause their validation upon invocation of the executable. Instead, the ExecutableValidator API (see *Section 3.2, “Validating method constraints”*) must be used to perform the validation, which is often done using a method interception facility such as AOP, proxy objects etc.

Constraints may only be applied to instance methods, i.e. declaring constraints on static methods is not supported. Depending on the interception facility you use for triggering method validation, additional restrictions may apply, e.g. with respect to the visibility of methods supported as target of interception. Refer to the documentation of the interception technology to find out whether any such limitations exist.

3.1.1.1. Cross-parameter constraints

Sometimes validation does not only depend on a single parameter but on several or even all parameters of a method or constructor. This kind of requirement can be fulfilled with help of a cross-parameter constraint.

Cross-parameter constraints can be considered as the method validation equivalent to class-level constraints. Both can be used to implement validation requirements which are based on several elements. While class-level constraints apply to several properties of a bean, cross-parameter constraints apply to several parameters of an executable.

In contrast to single-parameter constraints, cross-parameter constraints are declared on the method or constructor as you can see in *Example 3.2, “Declaring a cross-parameter constraint”*. Here the cross-parameter constraint `@LuggageCountMatchesPassengerCount` declared on the `load()` method is used to ensure that no passenger has more than two pieces of luggage.

Example 3.2. Declaring a cross-parameter constraint

```
package org.hibernate.validator.referenceguide.chapter03.crossparameter;

public class Car {

    @LuggageCountMatchesPassengerCount(piecesOfLuggagePerPassenger = 2)
    public void load(List<Person> passengers, List<PieceOfLuggage> luggage) {
        //...
    }
}
```

As you will learn in the next section, return value constraints are also declared on the method level. In order to distinguish cross-parameter constraints from return value constraints, the constraint target is configured in the `ConstraintValidator` implementation using the `@SupportedValidationTarget` annotation. You can find out about the details in *Section 6.3, “Cross-parameter constraints”* which shows how to implement your own cross-parameter constraint.

In some cases a constraint can be applied to an executable's parameters (i.e. it is a cross-parameter constraint), but also to the return value. One example for this are custom constraints which allow to specify validation rules using expression or script languages.

Such constraints must define a member `validationAppliesTo()` which can be used at declaration time to specify the constraint target. As shown in *Example 3.3, “Specifying a constraint's target”* you apply the constraint to an executable's parameters by specifying `validationAppliesTo = ConstraintTarget.PARAMETERS`, while `ConstraintTarget.RETURN_VALUE` is used to apply the constraint to the executable return value.

Example 3.3. Specifying a constraint's target

```
package org.hibernate.validator.referenceguide.chapter03.crossparameter.constrainttarget;

public class Garage {

    @ELAssert(expression = "...", validationAppliesTo = ConstraintTarget.PARAMETERS)
    public Car buildCar(List<Part> parts) {
        //...
```

```
    }

    @ELAssert(expression = "...", validationAppliesTo = ConstraintTarget.RETURN_VALUE)
    public Car paintCar(int color) {
        //...
    }
}
```

Although such a constraint is applicable to the parameters and return value of an executable, the target can often be inferred automatically. This is the case, if the constraint is declared on

- a void method with parameters (the constraint applies to the parameters)
- an executable with return value but no parameters (the constraint applies to the return value)
- neither a method nor a constructor, but a field, parameter etc. (the constraint applies to the annotated element)

In these situations you don't have to specify the constraint target. It is still recommended to do so if it increases readability of the source code. If the constraint target is not specified in situations where it can't be determined automatically, a `ConstraintDeclarationException` is raised.

3.1.2. Return value constraints

The postconditions of a method or constructor are declared by adding constraint annotations to the executable as shown in *Example 3.4, “Declaring method and constructor return value constraints”*.

Example 3.4. Declaring method and constructor return value constraints

```
package org.hibernate.validator.referenceguide.chapter03.returnvalue;

public class RentalStation {

    @ValidRentalStation
    public RentalStation() {
        //...
    }

    @NotNull
    @Size(min = 1)
    public List<Customer> getCustomers() {
        //...
    }
}
```

The following constraints apply to the executables of `RentalStation`:

- Any newly created `RentalStation` object must satisfy the `@ValidRentalStation` constraint

- The customer list returned by `getCustomers()` must not be `null` and must contain at least on element

3.1.3. Cascaded validation

Similar to the cascaded validation of JavaBeans properties (see *Section 2.1.5, "Object graphs"*), the `@Valid` annotation can be used to mark executable parameters and return values for cascaded validation. When validating a parameter or return value annotated with `@Valid`, the constraints declared on the parameter or return value object are validated as well.

In *Example 3.5, "Marking executable parameters and return values for cascaded validation"*, the `car` parameter of the method `Garage#checkCar()` as well as the return value of the `Garage` constructor are marked for cascaded validation.

Example 3.5. Marking executable parameters and return values for cascaded validation

```
package org.hibernate.validator.referenceguide.chapter03.cascaded;

public class Garage {

    @NotNull
    private String name;

    @Valid
    public Garage(String name) {
        this.name = name;
    }

    public boolean checkCar(@Valid @NotNull Car car) {
        //...
    }
}
```

```
package org.hibernate.validator.referenceguide.chapter03.cascaded;

public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    public Car(String manufacturer, String licencePlate) {
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
    }

    //getters and setters ...
```

```
}
```

When validating the arguments of the `checkCar()` method, the constraints on the properties of the passed `Car` object are evaluated as well. Similarly, the `@NotNull` constraint on the `name` field of `Garage` is checked when validating the return value of the `Garage` constructor.

Generally, the cascaded validation works for executables in exactly the same way as it does for JavaBeans properties.

In particular, `null` values are ignored during cascaded validation (naturally this can't happen during constructor return value validation) and cascaded validation is performed recursively, i.e. if a parameter or return value object which is marked for cascaded validation itself has properties marked with `@Valid`, the constraints declared on the referenced elements will be validated as well.

Cascaded validation can not only be applied to simple object references but also to collection-typed parameters and return values. This means when putting the `@Valid` annotation to a parameter or return value which

- is an array
- implements `java.lang.Iterable`
- or implements `java.util.Map`

each contained element gets validated. So when validating the arguments of the `checkCars()` method in *Example 3.6, “List-typed method parameter marked for cascaded validation”*, each element instance of the passed list will be validated and a `ConstraintViolation` created when any of the contained `Car` objects is invalid.

Example 3.6. List-typed method parameter marked for cascaded validation

```
package org.hibernate.validator.referenceguide.chapter03.cascaded.collection;

public class Garage {

    public boolean checkCars(@Valid @NotNull List<Car> cars) {
        //...
    }
}
```

3.1.4. Method constraints in inheritance hierarchies

When declaring method constraints in inheritance hierarchies, it is important to be aware of the following rules:

- The preconditions to be satisified by the caller of a method may not be strengthened in subtypes

- The postconditions guaranteed to the caller of a method may not be weakened in subtypes

These rules are motivated by the concept of *behavioral subtyping* which requires that wherever a type `T` is used, also a subtype `S` of `T` may be used without altering the program's behavior.

As an example, consider a class invoking a method on an object with the static type `T`. If the runtime type of that object was `S` and `S` imposed additional preconditions, the client class might fail to satisfy these preconditions as is not aware of them. The rules of behavioral subtyping are also known as the *Liskov substitution principle* [http://en.wikipedia.org/wiki/Liskov_substitution_principle].

The Bean Validation specification implements the first rule by disallowing parameter constraints on methods which override or implement a method declared in a supertype (superclass or interface). *Example 3.7, "Illegal method parameter constraint in subtype"* shows a violation of this rule.

Example 3.7. Illegal method parameter constraint in subtype

```
package org.hibernate.validator.referenceguide.chapter03.inheritance.parameter;

public interface Vehicle {

    void drive(@Max(75) int speedInMph);
}
```

```
package org.hibernate.validator.referenceguide.chapter03.inheritance.parameter;

public class Car implements Vehicle {

    @Override
    public void drive(@Max(55) int speedInMph) {
        //...
    }
}
```

The `@Max` constraint on `Car#drive()` is illegal since this method implements the interface method `Vehicle#drive()`. Note that parameter constraints on overriding methods are also disallowed, if the supertype method itself doesn't declare any parameter constraints.

Furthermore, if a method overrides or implements a method declared in several parallel supertypes (e.g. two interfaces not extending each other or a class and an interface not implemented by that class), no parameter constraints may be specified for the method in any of the involved types. The types in *Example 3.8, "Illegal method parameter constraint in parallel types of a hierarchy"* demonstrate a violation of that rule. The method `RacingCar#drive()` overrides `Vehicle#drive()` as well as `Car#drive()`. Therefore the constraint on `Vehicle#drive()` is illegal.

Example 3.8. Illegal method parameter constraint in parallel types of a hierarchy

```
package org.hibernate.validator.referenceguide.chapter03.inheritance.parallel;

public interface Vehicle {

    void drive(@Max(75) int speedInMph);
}
```

```
package org.hibernate.validator.referenceguide.chapter03.inheritance.parallel;

public interface Car {

    public void drive(int speedInMph);
}
```

```
package org.hibernate.validator.referenceguide.chapter03.inheritance.parallel;

public class RacingCar implements Car, Vehicle {

    @Override
    public void drive(int speedInMph) {
        //...
    }
}
```

The previously described restrictions only apply to parameter constraints. In contrast, return value constraints may be added in methods overriding or implementing any supertype methods.

In this case, all the method's return value constraints apply for the subtype method, i.e. the constraints declared on the subtype method itself as well as any return value constraints on overridden/implemented supertype methods. This is legal as putting additional return value constraints in place may never represent a weakening of the postconditions guaranteed to the caller of a method.

So when validating the return value of the method `Car#getPassengers()` shown in *Example 3.9, “Return value constraints on supertype and subtype method”*, the `@Size` constraint on the method itself as well as the `@NotNull` constraint on the implemented interface method `Vehicle#getPassengers()` apply.

Example 3.9. Return value constraints on supertype and subtype method

```
package org.hibernate.validator.referenceguide.chapter03.inheritance.returnvalue;

public interface Vehicle {
```

```
    @NotNull
    List<Person> getPassengers();
}
```

```
package org.hibernate.validator.referenceguide.chapter03.inheritance.returnvalue;

public class Car implements Vehicle {

    @Override
    @Size(min = 1)
    public List<Person> getPassengers() {
        //...
    }
}
```

If the validation engine detects a violation of any of the aforementioned rules, a `ConstraintDeclarationException` will be raised.

> **Note**
>
> The rules described in this section only apply to methods but not constructors. By definition, constructors never override supertype constructors. Therefore, when validating the parameters or the return value of a constructor invocation only the constraints declared on the constructor itself apply, but never any constraints declared on supertype constructors.

3.2. Validating method constraints

The validation of method constraints is done using the `ExecutableValidator` interface.

In *Section 3.2.1, “Obtaining an ExecutableValidator instance”* you will learn how to obtain an `ExecutableValidator` instance while *Section 3.2.2, “ExecutableValidator methods”* shows how to use the different methods offered by this interface.

Instead of calling the ExecutableValidator methods directly from within application code, they are usually invoked via a method interception technology such as AOP, proxy objects, etc. This causes executable constraints to be validated automatically and transparently upon method or constructor invocation. Typically a `ConstraintViolationException` is raised by the integration layer in case any of the constraints is violated.

3.2.1. Obtaining an `ExecutableValidator` instance

You can retrieve an `ExecutableValidator` instance via `Validator#forExecutables()` as shown in *Example 3.10, “Obtaining an ExecutableValidator”*.

Example 3.10. Obtaining an ExecutableValidator

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
executableValidator = factory.getValidator().forExecutables();
```

In the example the executable validator is retrieved from the default validator factory, but if required you could also bootstrap a specifically configured factory as described in *Chapter 8, Bootstrapping*, for instance in order to use a specific parameter name provider (see *Section 8.2.4, "ParameterNameProvider"*).

3.2.2. `ExecutableValidator` methods

The `ExecutableValidator` interface offers altogether four methods:

- `validateParameters()` and `validateReturnValue()` for method validation
- `validateConstructorParameters()` and `validateConstructorReturnValue()` for constructor validation

Just as the methods on `Validator`, all these methods return a `Set<ConstraintViolation>` which contains a `ConstraintViolation` instance for each violated constraint and which is empty if the validation succeeds. Also all the methods have a var-args `groups` parameter by which you can pass the validation groups to be considered for validation.

The examples in the following sections are based on the methods on constructors of the `Car` class shown in *Example 3.11, "Class Car with constrained methods and constructors"*.

Example 3.11. Class `Car` with constrained methods and constructors

```
package org.hibernate.validator.referenceguide.chapter03.validation;

public class Car {

    public Car(@NotNull String manufacturer) {
        //...
    }

    @ValidRacingCar
    public Car(String manufacturer, String team) {
        //...
    }

    public void drive(@Max(75) int speedInMph) {
        //...
    }

    @Size(min = 1)
    public List<Passenger> getPassengers() {
        //...
```

```
    }
}
```

3.2.2.1. `ExecutableValidator#validateParameters()`

The method `validateParameters()` is used to validate the arguments of a method invocation. *Example 3.12, "Using ExecutableValidator#validateParameters()"* shows an example. The validation results in a violation of the `@Max` constraint on the parameter of the `drive()` method.

Example 3.12. Using `ExecutableValidator#validateParameters()`

```
Car object = new Car( "Morris" );
Method method = Car.class.getMethod( "drive", int.class );
Object[] parameterValues = { 80 };
Set<ConstraintViolation<Car>> violations = executableValidator.validateParameters(
        object,
        method,
        parameterValues
);

assertEquals( 1, violations.size() );
Class<? extends Annotation> constraintType = violations.iterator()
        .next()
        .getConstraintDescriptor()
        .getAnnotation()
        .annotationType();
assertEquals( Max.class, constraintType );
```

Note that `validateParameters()` validates all the parameter constraints of a method, i.e. constraints on individual parameters as well as cross-parameter constraints.

3.2.2.2. `ExecutableValidator#validateReturnValue()`

Using `validateReturnValue()` the return value of a method can can be validated. The validation in *Example 3.13, "Using ExecutableValidator#validateReturnValue()"* yields one constraint violation since the `getPassengers()` method is expect to return at least one `Passenger` object.

Example 3.13. Using `ExecutableValidator#validateReturnValue()`

```
Car object = new Car( "Morris" );
Method method = Car.class.getMethod( "getPassengers" );
Object returnValue = Collections.<Passenger>emptyList();
Set<ConstraintViolation<Car>> violations = executableValidator.validateReturnValue(
        object,
        method,
        returnValue
);

assertEquals( 1, violations.size() );
Class<? extends Annotation> constraintType = violations.iterator()
```

```
        .next()
        .getConstraintDescriptor()
        .getAnnotation()
        .annotationType();
assertEquals( Size.class, constraintType );
```

3.2.2.3. `ExecutableValidator#validateConstructorParameters()`

The arguments of constructor invocations can be validated with `validateConstructorParameters()` as shown in method *Example 3.14, “Using ExecutableValidator#validateConstructorParameters()”*. Due to the `@NotNull` constraint on the `manufacturer` parameter, the validation call returns one constraint violation.

Example 3.14. Using `ExecutableValidator#validateConstructorParameters()`

```
Constructor<Car> constructor = Car.class.getConstructor( String.class );
Object[] parameterValues = { null };
Set<ConstraintViolation<Car>> violations = executableValidator.validateConstructorParameters(
        constructor,
        parameterValues
);

assertEquals( 1, violations.size() );
Class<? extends Annotation> constraintType = violations.iterator()
        .next()
        .getConstraintDescriptor()
        .getAnnotation()
        .annotationType();
assertEquals( NotNull.class, constraintType );
```

3.2.2.4. `ExecutableValidator#validateConstructorReturnValue()`

Finally, by using `validateConstructorReturnValue()` you can valide a constructor's return value. In *Example 3.15, “Using ExecutableValidator#validateConstructorReturnValue()”*, `validateConstructorReturnValue()` returns one constraint violation, since the `Car` object returned by the constructor doesn't satisfy the `@ValidRacingCar` constraint (not shown).

Example 3.15. Using `ExecutableValidator#validateConstructorReturnValue()`

```
//constructor for creating racing cars
Constructor<Car> constructor = Car.class.getConstructor( String.class, String.class );
Car createdObject = new Car( "Morris", null );
Set<ConstraintViolation<Car>> violations = executableValidator.validateConstructorReturnValue(
        constructor,
        createdObject
);

assertEquals( 1, violations.size() );
Class<? extends Annotation> constraintType = violations.iterator()
        .next()
```

```
        .getConstraintDescriptor()
        .getAnnotation()
        .annotationType();
assertEquals( ValidRacingCar.class, constraintType );
```

3.2.3. `ConstraintViolation` methods for method validation

In addition to the methods introduced in *Section 2.2.3, "ConstraintViolation methods"*, `ConstraintViolation` provides two more methods specific to the validation of executable parameters and return values.

`ConstraintViolation#getExecutableParameters()` returns the validated parameter array in case of method or constructor parameter validation, while `ConstraintViolation#getExecutableReturnValue()` provides access to the validated object in case of return value validation.

All the other `ConstraintViolation` methods generally work for method validation in the same way as for validation of beans. Refer to the *JavaDoc* [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/index.html?javax/validation/metadata/BeanDescriptor.html] to learn more about the behavior of the individual methods and their return values during bean and method validation.

Note that `getPropertyPath()` can be very useful in order to obtain detailed information about the validated parameter or return value, e.g. for logging purposes. In particular, you can retrieve name and argument types of the concerned method as well as the index of the concerned parameter from the path nodes. How this can be done is shown in *Example 3.16, "Retrieving method and parameter information"*.

Example 3.16. Retrieving method and parameter information

```
Car object = new Car( "Morris" );
Method method = Car.class.getMethod( "drive", int.class );
Object[] parameterValues = { 80 };
Set<ConstraintViolation<Car>> violations = executableValidator.validateParameters(
        object,
        method,
        parameterValues
);

assertEquals( 1, violations.size() );
Iterator<Node> propertyPath = violations.iterator()
        .next()
        .getPropertyPath()
        .iterator();

MethodNode methodNode = propertyPath.next().as( MethodNode.class );
assertEquals( "drive", methodNode.getName() );
assertEquals( Arrays.<Class<?>>asList( int.class ), methodNode.getParameterTypes() );

ParameterNode parameterNode = propertyPath.next().as( ParameterNode.class );
assertEquals( "arg0", parameterNode.getName() );
```

```
assertEquals( 0, parameterNode.getParameterIndex() );
```

The parameter name is determined using the current `ParameterNameProvider` (see *Section 8.2.4, “ParameterNameProvider”*) and defaults to `arg0`, `arg1` etc.

3.3. Built-in method constraints

In addition to the built-in bean and property-level constraints discussed in *Section 2.3, “Built-in constraints”*, Hibernate Validator currently provides one method-level constraint, `@ParameterScriptAssert`. This is a generic cross-parameter constraint which allows to implement validation routines using any JSR 223 compatible ("Scripting for the Java™ Platform") scripting language, provided an engine for this language is available on the classpath.

To refer to the executable's parameters from within the expression, use their name as obtained from the active parameter name provider (see *Section 8.2.4, “ParameterNameProvider”*). *Example 3.17, “Using @ParameterScriptAssert”* shows how the validation logic of the `@LuggageCountMatchesPassengerCount` constraint from *Example 3.2, “Declaring a cross-parameter constraint”* could be expressed with the help of `@ParameterScriptAssert`.

Example 3.17. Using `@ParameterScriptAssert`

```
package org.hibernate.validator.referenceguide.chapter03.parametersscriptassert;

public class Car {

    @ParameterScriptAssert(lang = "javascript", script = "arg1.size() <= arg0.size() * 2")
    public void load(List<Person> passengers, List<PieceOfLuggage> luggage) {
        //...
    }
}
```

Interpolating constraint error messages

Message interpolation is the process of creating error messages for violated Bean Validation constraints. In this chapter you will learn how such messages are defined and resolved and how you can plug in custom message interpolators in case the default algorithm is not sufficient for your requirements.

4.1. Default message interpolation

Constraint violation messages are retrieved from so called message descriptors. Each constraint defines its default message descriptor using the `message` attribute. At declaration time, the default descriptor can be overridden with a specific value as shown in *Example 4.1, “Specifying a message descriptor using the message attribute”*.

Example 4.1. Specifying a message descriptor using the `message` attribute

```
package org.hibernate.validator.referenceguide.chapter04;

public class Car {

    @NotNull(message = "The manufacturer name must not be null")
    private String manufacturer;

    //constructor, getters and setters ...
}
```

If a constraint is violated, its descriptor will be interpolated by the validation engine using the currently configured `MessageInterpolator`. The interpolated error message can then be retrieved from the resulting constraint violation by calling `ConstraintViolation#getMessage()`.

Message descriptors can contain *message parameters* as well as *message expressions* which will be resolved during interpolation. Message parameters are string literals enclosed in `{}`, while message expressions are string literals enclosed in `${}`. The following algorithm is applied during method interpolation:

1. Resolve any message parameters by using them as key for the resource bundle `ValidationMessages`. If this bundle contains an entry for a given message parameter, that parameter will be replaced in the message with the corresponding value from the bundle. This step will be executed recursively in case the replaced value again contains message parameters. The resource bundle is expected to be provided by the application developer, e.g. by adding a file named `ValidationMessages.properties` to the classpath. You can also create localized error messages by providing locale specific variations of this bundle,

such as `ValidationMessages_en_US.properties`. By default, the JVM's default locale (`Locale#getDefault()`) will be used when looking up messages in the bundle.

2. Resolve any message parameters by using them as key for a resource bundle containing the standard error messages for the built-in constraints as defined in Appendix B of the Bean Validation specification. In the case of Hibernate Validator, this bundle is named `org.hibernate.validator.ValidationMessages`. If this step triggers a replacement, step 1 is executed again, otherwise step 3 is applied.

3. Resolve any message parameters by replacing them with the value of the constraint annotation member of the same name. This allows to refer to attribute values of the constraint (e.g. `Size#min()`) in the error message (e.g. "must be at least ${min}").

4. Resolve any message expressions by evaluating them as expressions of the Unified Expression Language. See *Section 4.1.2, “Interpolation with message expressions”* to learn more about the usage of Unified EL in error messages.

Tip

You can find the formal definition of the interpolation algorithm in section *5.3.1.1* [http://beanvalidation.org/1.1/spec/#default-resolution-algorithm] of the Bean Validation specification.

4.1.1. Special characters

Since the characters `{`, `}` and `$` have a special meaning in message descriptors they need to be escaped if you want to use them literally. The following rules apply:

- `\{` is considered as the literal `{`
- `\}` is considered as the literal `}`
- `\$` is considered as the literal `$`
- `\\` is considered as the literal `\`

4.1.2. Interpolation with message expressions

As of Hibernate Validator 5 (Bean Validation 1.1) it is possible to use the Unified Expression Language (as defined by *JSR 341* [http://jcp.org/en/jsr/detail?id=341]) in constraint violation messages. This allows to define error messages based on conditional logic and also enables advanced formatting options. The validation engine makes the following objects available in the EL context:

- the attribute values of the constraint mapped to the attribute names
- the currently validated value (property, bean, method parameter etc.) under the name `validatedValue`

- a bean mapped to the name `formatter` exposing the var-arg method `format(String format, Object... args)` which behaves like `java.util.Formatter.format(String format, Object... args)`.

The following section provides several examples for using EL expressions in error messages.

4.1.3. Examples

Example 4.2, “Specifying message descriptors” shows how to make use of the different options for specifying message descriptors.

Example 4.2. Specifying message descriptors

```
package org.hibernate.validator.referenceguide.chapter04.complete;

public class Car {

    @NotNull
    private String manufacturer;

    @Size(
            min = 2,
            max = 14,
             message = "The license plate '${validatedValue}' must be between {min} and {max}
 characters long"
    )
    private String licensePlate;

    @Min(
            value = 2,
            message = "There must be at least {value} seat${value > 1 ? 's' : ''}"
    )
    private int seatCount;

    @DecimalMax(
            value = "350",
           message = "The top speed ${formatter.format('%1$.2f', validatedValue)} is higher " +
                    "than {value}"
    )
    private double topSpeed;

    @DecimalMax(value = "100000", message = "Price must not be higher than ${value}")
    private BigDecimal price;

    public Car(
            String manufacturer,
            String licensePlate,
            int seatCount,
            double topSpeed,
            BigDecimal price) {
        this.manufacturer = manufacturer;
        this.licensePlate = licensePlate;
        this.seatCount = seatCount;
        this.topSpeed = topSpeed;
        this.price = price;
```

```
    }

    //getters and setters ...
}
```

Validating an invalid `Car` instance yields constraint violations with the messages shown by the assertions in *Example 4.3, "Expected error messages"*:

- the `@NotNull` constraint on the `manufacturer` field causes the error message "may not be null", as this is the default message defined by the Bean Validation specification and no specific descriptor is given in the `message` attribute
- the `@Size` constraint on the `licensePlate` field shows the interpolation of message parameters (`{min}`, `{max}`) and how to add the validated value to the error message using the EL expression `${validatedValue}`
- the `@Min` constraint on `seatCount` demonstrates how use an EL expression with a ternery expression to dynamically chose singular or plural form, depending on an attribute of the constraint ("There must be at least 1 seat" vs. "There must be at least 2 seats")
- the message for the `@DecimalMax` constraint on `topSpeed` shows how to format the validated value using the `formatter` object
- finally, the `@DecimalMax` constraint on `price` shows that parameter interpolation has precedence over expression evaluation, causing the `$` sign to show up in front of the maximum price

Tip

Only actual constraint attributes can be interpolated using message parameters in the form `{attributeName}`. When referring to the validated value or custom expression variables added to the interpolation context (see *Section 11.6.1, "HibernateConstraintValidatorContext"*), an EL expression in the form `${attributeName}` must be used.

Example 4.3. Expected error messages

```
Car car = new Car( null, "A", 1, 400.123456, BigDecimal.valueOf( 200000 ) );

String message = validator.validateProperty( car, "manufacturer" )
        .iterator()
        .next()
        .getMessage();
assertEquals( "may not be null", message );

message = validator.validateProperty( car, "licensePlate" )
        .iterator()
```

```
        .next()
        .getMessage();
assertEquals(
        "The license plate must be between 2 and 14 characters long",
        message
);

message = validator.validateProperty( car, "seatCount" ).iterator().next().getMessage();
assertEquals( "There must be at least 2 seats", message );

message = validator.validateProperty( car, "topSpeed" ).iterator().next().getMessage();
assertEquals( "The top speed 400.12 is higher than 350", message );

message = validator.validateProperty( car, "price" ).iterator().next().getMessage();
assertEquals( "Price must not be higher than $100000", message );
```

4.2. Custom message interpolation

If the default message interpolation algorithm does not fit your requirements it is also possible to plug in a custom `MessageInterpolator` implementation.

Custom interpolators must implement the interface `javax.validation.MessageInterpolator`. Note that implementations must be thread-safe. It is recommended that custom message interpolators delegate final implementation to the default interpolator, which can be obtained via `Configuration#getDefaultMessageInterpolator()`.

In order to use a custom message interpolator it must be registered either by configuring it in the Bean Validation XML descriptor `META-INF/validation.xml` (see *Section 7.1, “Configuring the validator factory in validation.xml”*) or by passing it when bootstrapping a `ValidatorFactory` or `Validator` (see *Section 8.2.1, “MessageInterpolator”* and *Section 8.3, “Configuring a Validator”*, respectively).

4.2.1. `ResourceBundleLocator`

In some use cases you want to use the message interpolation algorithm as defined by the Bean Validation specification, but retrieve error messages from other resource bundles than `ValidationMessages`. In this situation Hibernate Validator's `ResourceBundleLocator` SPI can help.

The default message interpolator in Hibernate Validator, `ResourceBundleMessageInterpolator`, delegates retrieval of resource bundles to that SPI. Using an alternative bundle only requires passing an instance of `PlatformResourceBundleLocator` with the bundle name when bootstrapping the `ValidatorFactory` as shown in *Example 4.4, “Using a specific resource bundle”*.

Example 4.4. Using a specific resource bundle

```
Validator validator = Validation.byDefaultProvider()
        .configure()
```

```
        .messageInterpolator(
                new ResourceBundleMessageInterpolator(
                        new PlatformResourceBundleLocator( "MyMessages" )
                )
        )
        .buildValidatorFactory()
        .getValidator();
```

Of course you also could implement a completely different `ResourceBundleLocator`, which for instance returns bundles backed by records in a database. In this case you can obtain the default locator via `HibernateValidatorConfiguration#getDefaultResourceBundleLocator()`, which you e.g. could use as fallback for your custom locator.

Besides `PlatformResourceBundleLocator`, Hibernate Validator provides another resource bundle locator implementation out of the box, namely `AggregateResourceBundleLocator`, which allows to retrieve error messages from more than one resource bundle. You could for instance use this implementation in a multi-module application where you want to have one message bundle per module. *Example 4.5, “Using AggregateResourceBundleLocator”* shows how to use `AggregateResourceBundleLocator`.

Example 4.5. Using `AggregateResourceBundleLocator`

```
Validator validator = Validation.byDefaultProvider()
        .configure()
        .messageInterpolator(
                new ResourceBundleMessageInterpolator(
                        new AggregateResourceBundleLocator(
                                Arrays.asList(
                                        "MyMessages",
                                        "MyOtherMessages"
                                )
                        )
                )
        )
        .buildValidatorFactory()
        .getValidator();
```

Note that the bundles are processed in the order as passed to the constructor. That means if several bundles contain an entry for a given message key, the value will be taken from the first bundle in the list containing the key.

Grouping constraints

All validation methods on `Validator` and `ExecutableValidator` discussed in earlier chapters also take a var-arg argument *`groups`*. So far we have been ignoring this parameter, but it is time to have a closer look.

5.1. Requesting groups

Groups allow you to restrict the set of constraints applied during validation. One use case for validation groups are UI wizards where in each step only a specified subset of constraints should get validated. The groups targeted are passed as var-arg parameters to the appropriate validate method.

Let's have a look at an example. The class `Person` in *Example 5.1, "Person"* has a `@NotNull` constraint on name. Since no group is specified for this annotation the default group `javax.validation.groups.Default` is assumed.

Note

When more than one group is requested, the order in which the groups are evaluated is not deterministic. If no group is specified the default group `javax.validation.groups.Default` is assumed.

Example 5.1. Person

```
package org.hibernate.validator.referenceguide.chapter05;

public class Person {

    @NotNull
    private String name;

    public Person(String name) {
        this.name = name;
    }

    // getters and setters ...
}
```

The class `Driver` in *Example 5.2, "Driver"* extends `Person` and adds the properties age and hasDrivingLicense. Drivers must be at least 18 years old (`@Min(18)`) and have a driving license (`@AssertTrue`). Both constraints defined on these properties belong to the group `DriverChecks` which is just a simple tagging interface.

Tip

Using interfaces makes the usage of groups type-safe and allows for easy refactoring. It also means that groups can inherit from each other via class inheritance.

Example 5.2. Driver

```
package org.hibernate.validator.referenceguide.chapter05;

public class Driver extends Person {

    @Min(
            value = 18,
            message = "You have to be 18 to drive a car",
            groups = DriverChecks.class
    )
    public int age;

    @AssertTrue(
            message = "You first have to pass the driving test",
            groups = DriverChecks.class
    )
    public boolean hasDrivingLicense;

    public Driver(String name) {
        super( name );
    }

    public void passedDrivingTest(boolean b) {
        hasDrivingLicense = b;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```
package org.hibernate.validator.referenceguide.chapter05;

public interface DriverChecks {
}
```

Finally the class Car (*Example 5.3, “Car”*) has some constraints which are part of the default group as well as `@AssertTrue` in the group `CarChecks` on the property passedVehicleInspection which indicates whether a car passed the road worthy tests.

Example 5.3. Car

```
package org.hibernate.validator.referenceguide.chapter05;

public class Car {
    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    @AssertTrue(
            message = "The car has to pass the vehicle inspection first",
            groups = CarChecks.class
    )
    private boolean passedVehicleInspection;

    @Valid
    private Driver driver;

    public Car(String manufacturer, String licencePlate, int seatCount) {
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    // getters and setters ...
}
```

```
package org.hibernate.validator.referenceguide.chapter05;

public interface CarChecks {
}
```

Overall three different groups are used in the example:

- The constraints on Person.name, Car.manufacturer, Car.licensePlate and Car.seatCount all belong to the `Default` group
- The constraints on Driver.age and Driver.hasDrivingLicense belong to `DriverChecks`
- The constraint on Car.passedVehicleInspection belongs to the group `CarChecks`

Example 5.4, “Using validation groups” shows how passing different group combinations to the `Validator#validate()` method results in different validation results.

Example 5.4. Using validation groups

```
// create a car and check that everything is ok with it.
Car car = new Car( "Morris", "DD-AB-123", 2 );
Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );
assertEquals( 0, constraintViolations.size() );

// but has it passed the vehicle inspection?
constraintViolations = validator.validate( car, CarChecks.class );
assertEquals( 1, constraintViolations.size() );
assertEquals(
        "The car has to pass the vehicle inspection first",
        constraintViolations.iterator().next().getMessage()
);

// let's go to the vehicle inspection
car.setPassedVehicleInspection( true );
assertEquals( 0, validator.validate( car ).size() );

// now let's add a driver. He is 18, but has not passed the driving test yet
Driver john = new Driver( "John Doe" );
john.setAge( 18 );
car.setDriver( john );
constraintViolations = validator.validate( car, DriverChecks.class );
assertEquals( 1, constraintViolations.size() );
assertEquals(
        "You first have to pass the driving test",
        constraintViolations.iterator().next().getMessage()
);

// ok, John passes the test
john.passedDrivingTest( true );
assertEquals( 0, validator.validate( car, DriverChecks.class ).size() );

// just checking that everything is in order now
assertEquals(
        0, validator.validate(
        car,
        Default.class,
        CarChecks.class,
        DriverChecks.class
).size()
);
```

The first `validate()` call in *Example 5.4, “Using validation groups”* is done using no explicit group. There are no validation errors, even though the property passedVehicleInspection is per default `false`. However, the constraint defined on this property does not belong to the default group.

The next validation using the `CarChecks` group fails until the car passes the vehicle inspection. Adding a driver to the car and validating against `DriverChecks` again yields one constraint violation due to the fact that the driver has not yet passed the driving test. Only after setting passedDrivingTest to `true` the validation against `DriverChecks` passes.

The last validate() call finally shows that all constraints are passing by validating against all defined groups.

5.2. Defining group sequences

By default, constraints are evaluated in no particular order, regardless of which groups they belong to. In some situations, however, it is useful to control the order constraints are evaluated.

In the example from *Example 5.4, "Using validation groups"* it could for instance be required that first all default car constraints are passing before checking the road worthiness of the car. Finally, before driving away, the actual driver constraints should be checked.

In order to implement such a validation order you just need to define an interface and annotate it with `@GroupSequence`, defining the order in which the groups have to be validated (see *Example 5.5, "Defining a group sequence"*). If at least one constraint fails in a sequenced group none of the constraints of the following groups in the sequence get validated.

Example 5.5. Defining a group sequence

```
package org.hibernate.validator.referenceguide.chapter05;

@GroupSequence({ Default.class, CarChecks.class, DriverChecks.class })
public interface OrderedChecks {
}
```

Warning

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

You then can use the new sequence as shown in in *Example 5.6, "Using a group sequence"*.

Example 5.6. Using a group sequence

```
Car car = new Car( "Morris", "DD-AB-123", 2 );
car.setPassedVehicleInspection( true );

Driver john = new Driver( "John Doe" );
john.setAge( 18 );
john.passedDrivingTest( true );
car.setDriver( john );
```

```
assertEquals( 0, validator.validate( car, OrderedChecks.class ).size() );
```

5.3. Redefining the default group sequence

5.3.1. @GroupSequence

Besides defining group sequences, the `@GroupSequence` annotation also allows to redefine the default group for a given class. To do so, just add the `@GroupSequence` annotation to the class and specify the sequence of groups which substitute `Default` for this class within the annotation.

Example 5.7, "Class RentalCar with redefined default group" introduces a new class `RentalCar` with a redefined default group.

Example 5.7. Class `RentalCar` with redefined default group

```
package org.hibernate.validator.referenceguide.chapter05;

@GroupSequence({ RentalChecks.class, CarChecks.class, RentalCar.class })
public class RentalCar extends Car {
    @AssertFalse(message = "The car is currently rented out", groups = RentalChecks.class)
    private boolean rented;

    public RentalCar(String manufacturer, String licencePlate, int seatCount) {
        super( manufacturer, licencePlate, seatCount );
    }

    public boolean isRented() {
        return rented;
    }

    public void setRented(boolean rented) {
        this.rented = rented;
    }
}
```

```
package org.hibernate.validator.referenceguide.chapter05;

public interface RentalChecks {
}
```

With this definition you can evaluate the constraints belonging to `RentalChecks`, `CarChecks` and `RentalCar` by just requesting the `Default` group as seen in *Example 5.8, "Validating an object with redefined default group"*.

Example 5.8. Validating an object with redefined default group

```
RentalCar rentalCar = new RentalCar( "Morris", "DD-AB-123", 2 );
```

```
rentalCar.setPassedVehicleInspection( true );
rentalCar.setRented( true );

Set<ConstraintViolation<RentalCar>> constraintViolations = validator.validate( rentalCar );

assertEquals( 1, constraintViolations.size() );
assertEquals(
        "Wrong message",
        "The car is currently rented out",
        constraintViolations.iterator().next().getMessage()
);

rentalCar.setRented( false );
constraintViolations = validator.validate( rentalCar );

assertEquals( 0, constraintViolations.size() );
```

Note

Since there must no cyclic dependency in the group and group sequence definitions one cannot just add `Default` to the sequence redefining `Default` for a class. Instead the class itself has to be added!

The `Default` group sequence overriding is local to the class it is defined on and is not propagated to associated objects. For the example this means that adding `DriverChecks` to the default group sequence of `RentalCar` would not have any effects. Only the group `Default` will be propagated to the driver association.

Note that you can control the propagated group(s) by declaring a group conversion rule (see *Section 5.4, “Group conversion”*).

5.3.2. `@GroupSequenceProvider`

In addition to statically redefining default group sequences via `@GroupSequence`, Hibernate Validator also provides an SPI for the dynamic redefinition of default group sequences depending on the object state.

For that purpose you need to implement the interface `DefaultGroupSequenceProvider` and register this implementation with the target class via the `@GroupSequenceProvider` annotation. In the rental car scenario you could for instance dynamically add the `CarChecks` as seen in *Example 5.9, “Implementing and using a default group sequence provider”*.

Example 5.9. Implementing and using a default group sequence provider

```
package org.hibernate.validator.referenceguide.chapter05.groupsequenceprovider;

public class RentalCarGroupSequenceProvider
        implements DefaultGroupSequenceProvider<RentalCar> {
```

```
    @Override
    public List<Class<?>> getValidationGroups(RentalCar car) {
        List<Class<?>> defaultGroupSequence = new ArrayList<Class<?>>();
        defaultGroupSequence.add( RentalCar.class );

        if ( car != null && !car.isRented() ) {
            defaultGroupSequence.add( CarChecks.class );
        }

        return defaultGroupSequence;
    }
}
```

```
package org.hibernate.validator.referenceguide.chapter05.groupsequenceprovider;

@GroupSequenceProvider(RentalCarGroupSequenceProvider.class)
public class RentalCar extends Car {

    @AssertFalse(message = "The car is currently rented out", groups = RentalChecks.class)
    private boolean rented;

    public RentalCar(String manufacturer, String licencePlate, int seatCount) {
        super( manufacturer, licencePlate, seatCount );
    }

    public boolean isRented() {
        return rented;
    }

    public void setRented(boolean rented) {
        this.rented = rented;
    }
}
```

5.4. Group conversion

What if you wanted to validate the car related checks together with the driver checks? Of course you could pass the required groups to the validate call explicitly, but what if you wanted to make these validations occur as part of the `Default` group validation? Here `@ConvertGroup` comes into play which allows you during cascaded validation to use a different group than the originally requested one.

Let's have a look at *Example 5.10, “@ConvertGroup usage”*. Here `@GroupSequence({ CarChecks.class, Car.class })` is used to combine the car related constraints under the `Default` group (see *Section 5.3, “Redefining the default group sequence”*). There is also a `@ConvertGroup(from = Default.class, to = DriverChecks.class)` which ensures the `Default` group gets converted to the `DriverChecks` group during cascaded validation of the driver association.

Example 5.10. `@ConvertGroup` usage

```
package org.hibernate.validator.referenceguide.chapter05.groupconversion;

public class Driver {

    @NotNull
    private String name;

    @Min(
            value = 18,
            message = "You have to be 18 to drive a car",
            groups = DriverChecks.class
    )
    public int age;

    @AssertTrue(
            message = "You first have to pass the driving test",
            groups = DriverChecks.class
    )
    public boolean hasDrivingLicense;

    public Driver(String name) {
        this.name = name;
    }

    public void passedDrivingTest(boolean b) {
        hasDrivingLicense = b;
    }

    // getters and setters ...
}
```

```
package org.hibernate.validator.referenceguide.chapter05.groupconversion;

@GroupSequence({ CarChecks.class, Car.class })
public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    @AssertTrue(
            message = "The car has to pass the vehicle inspection first",
            groups = CarChecks.class
    )
    private boolean passedVehicleInspection;

    @Valid
```

```
    @ConvertGroup(from = Default.class, to = DriverChecks.class)
    private Driver driver;

    public Car(String manufacturer, String licencePlate, int seatCount) {
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    // getters and setters ...
}
```

As a result the validation in *Example 5.11, “Test case for @ConvertGroup”* succeeds, even though the constraint on hasDrivingLicense belongs to the `DriverChecks` group and only the `Default` group is requested in the `validate()` call.

Example 5.11. Test case for @ConvertGroup

```
// create a car and validate. The Driver is still null and does not get validated
Car car = new Car( "VW", "USD-123", 4 );
car.setPassedVehicleInspection( true );
Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );
assertEquals( 0, constraintViolations.size() );

// create a driver who has not passed the driving test
Driver john = new Driver( "John Doe" );
john.setAge( 18 );

// now let's add a driver to the car
car.setDriver( john );
constraintViolations = validator.validate( car );
assertEquals( 1, constraintViolations.size() );
assertEquals(
        "The driver constraint should also be validated as part of the default group",
        constraintViolations.iterator().next().getMessage(),
        "You first have to pass the driving test"
);
```

You can define group conversions wherever `@Valid` can be used, namely associations as well as method and constructor parameters and return values. Multiple conversions can be specified using `@ConvertGroup.List`.

However, the following restrictions apply:

- `@ConvertGroup` must only be used in combination with `@Valid`. If used without, a `ConstraintDeclarationException` is thrown.
- It is not legal to have multiple conversion rules on the same element with the same *`from`* value. In this case, a `ConstraintDeclarationException` is raised.
- The *`from`* attribute must not refer to a group sequence. A `ConstraintDeclarationException` is raised in this situation.

Note

Rules are not executed recursively. The first matching conversion rule is used and subsequent rules are ignored. For example if a set of `@ConvertGroup` declarations chains group A to B and B to C, the group A will be converted to B and not to C.

Creating custom constraints

The Bean Validation API defines a whole set of standard constraint annotations such as `@NotNull`, `@Size` etc. In cases where these buit-in constraints are not sufficient, you cean easily create custom constraints tailored to your specific validation requirements.

6.1. Creating a simple constraint

To create a custom constraint, the following three steps are required:

- Create a constraint annotation
- Implement a validator
- Define a default error message

6.1.1. The constraint annotation

This section shows how to write a constraint annotation which can be used to ensure that a given string is either completely upper case or lower case. Later on this constraint will be applied to the licensePlate field of the `Car` class from *Chapter 1, Getting started* to ensure, that the field is always an upper-case string.

The first thing needed is a way to express the two case modes. While you could use `String` constants, a better approach is using a Java 5 enum for that purpose:

Example 6.1. Enum `CaseMode` to express upper vs. lower case

```
package org.hibernate.validator.referenceguide.chapter06;

public enum CaseMode {
    UPPER,
    LOWER;
}
```

The next step is to define the actual constraint annotation. If you've never designed an annotation before, this may look a bit scary, but actually it's not that hard:

Example 6.2. Defining the `@CheckCase` constraint annotation

```
package org.hibernate.validator.referenceguide.chapter06;

@Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })
@Retention(RUNTIME)
```

```
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
public @interface CheckCase {

    String message() default "{org.hibernate.validator.referenceguide.chapter06.CheckCase." +
            "message}";

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

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

    CaseMode value();

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

An annotation type is defined using the `@interface` keyword. All attributes of an annotation type are declared in a method-like manner. The specification of the Bean Validation API demands, that any constraint annotation defines

- an attribute message that returns the default key for creating error messages in case the constraint is violated
- an attribute groups that allows the specification of validation groups, to which this constraint belongs (see *Chapter 5, Grouping constraints*). This must default to an empty array of type `Class<?>`.
- an attribute `payload` that can be used by clients of the Bean Validation API to assign custom payload objects to a constraint. This attribute is not used by the API itself. An example for a custom payload could be the definition of a severity:

```
public class Severity {
    public interface Info extends Payload {
    }

    public interface Error extends Payload {
    }
}

public class ContactDetails {
    @NotNull(message = "Name is mandatory", payload = Severity.Error.class)
    private String name;

    @NotNull(message = "Phone number not specified, but not mandatory",
            payload = Severity.Info.class)
    private String phoneNumber;
```

```
    // ...
}
```

Now a client can after the validation of a `ContactDetails` instance access the severity of a constraint using `ConstraintViolation.getConstraintDescriptor().getPayload()` and adjust its behaviour depending on the severity.

Besides these three mandatory attributes there is another one, value, allowing for the required case mode to be specified. The name value is a special one, which can be omitted when using the annotation, if it is the only attribute specified, as e.g. in `@CheckCase(CaseMode.UPPER)`.

In addition, the constraint annotation is decorated with a couple of meta annotations:

- `@Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })`: Defines the supported target element types for the constraint. `@CheckCase` may be used on fields (element type `FIELD`), JavaBeans properties as well as method return values (`METHOD`) and method/constructor parameters (`PARAMETER`). The element type `ANNOTATION_TYPE` allows for the creation of composed constraints (see *Section 6.4, “Constraint composition”*) based on `@CheckCase`.

  When creating a class-level constraint (see *Section 2.1.3, “Class-level constraints”*), the element type `TYPE` would have to be used. Constraints targetting the return value of a constructor need to support the element type `CONSTRUCTOR`. Cross-parameter constraints (see *Section 6.3, “Cross-parameter constraints”*) which are used to validate all the parameters of a method or constructor together, must support `METHOD` or `CONSTRUCTOR`, respectively.

- `@Retention(RUNTIME)`: Specifies, that annotations of this type will be available at runtime by the means of reflection

- `@Constraint(validatedBy = CheckCaseValidator.class)`: Marks the annotation type as constraint annotation and specifies the validator to be used to validate elements annotated with `@CheckCase`. If a constraint may be used on several data types, several validators may be specified, one for each data type.

- `@Documented`: Says, that the use of `@CheckCase` will be contained in the JavaDoc of elements annotated with it

Finally, there is an inner annotation type named `List`. This annotation allows to specify several `@CheckCase` annotations on the same element, e.g. with different validation groups and messages. While also another name could be used, the Bean Validation specification recommends to use the name `List` and make the annotation an inner annotation of the corresponding constraint type.

6.1.2. The constraint validator

Having defined the annotation, you need to create a constraint validator, which is able to validate elements with a `@CheckCase` annotation. To do so, implement the interface `ConstraintValidator` as shown below:

Example 6.3. Implementing a constraint validator for the constraint `@CheckCase`

```
package org.hibernate.validator.referenceguide.chapter06;

public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

    @Override
    public void initialize(CheckCase constraintAnnotation) {
        this.caseMode = constraintAnnotation.value();
    }

    @Override
    public boolean isValid(String object, ConstraintValidatorContext constraintContext) {
        if ( object == null ) {
            return true;
        }

        if ( caseMode == CaseMode.UPPER ) {
            return object.equals( object.toUpperCase() );
        }
        else {
            return object.equals( object.toLowerCase() );
        }
    }
}
```

The `ConstraintValidator` interface defines two type parameters which are set in the implementation. The first one specifies the annotation type to be validated (`CheckCase`), the second one the type of elements, which the validator can handle (`String`). In case a constraint supports several data types, a `ConstraintValidator` for each allowed type has to be implemented and registered at the constraint annotation as shown above.

The implementation of the validator is straightforward. The `initialize()` method gives you access to the attribute values of the validated constraint and allows you to store them in a field of the validator as shown in the example.

The `isValid()` method contains the actual validation logic. For `@CheckCase` this is the check whether a given string is either completely lower case or upper case, depending on the case mode retrieved in `initialize()`. Note that the Bean Validation specification recommends to consider `null` values as being valid. If `null` is not a valid value for an element, it should be annotated with `@NotNull` explicitly.

6.1.2.1. The `ConstraintValidatorContext`

Example 6.3, “Implementing a constraint validator for the constraint @CheckCase” relies on the default error message generation by just returning `true` or `false` from the `isValid()` method. Using the passed `ConstraintValidatorContext` object it is possible to either add additional error messages or completely disable the default error message generation and solely define custom error messages. The `ConstraintValidatorContext` API is modeled as fluent interface and is best demonstrated with an example:

Example 6.4. Using `ConstraintValidatorContext` to define custom error messages

```
package org.hibernate.validator.referenceguide.chapter06.constraintvalidatorcontext;

public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

    @Override
    public void initialize(CheckCase constraintAnnotation) {
        this.caseMode = constraintAnnotation.value();
    }

    @Override
    public boolean isValid(String object, ConstraintValidatorContext constraintContext) {
        if ( object == null ) {
            return true;
        }

        boolean isValid;
        if ( caseMode == CaseMode.UPPER ) {
            isValid = object.equals( object.toUpperCase() );
        }
        else {
            isValid = object.equals( object.toLowerCase() );
        }

        if ( !isValid ) {
            constraintContext.disableDefaultConstraintViolation();
            constraintContext.buildConstraintViolationWithTemplate(
                    "{org.hibernate.validator.referenceguide.chapter03." +
                    "constraintvalidatorcontext.CheckCase.message}"
            )
            .addConstraintViolation();
        }

        return isValid;
    }
}
```

Example 6.4, "Using ConstraintValidatorContext to define custom error messages" shows how you can disable the default error message generation and add a custom error message using a specified message template. In this example the use of the `ConstraintValidatorContext` results in the same error message as the default error message generation.

> **Tip**
>
> It is important to add each configured constraint violation by calling `addConstraintViolation()`. Only after that the new constraint violation will be created.

Refer to *Section 6.2.1, “Custom property paths”* to learn how to use the `ConstraintValidatorContext` API to control the property path of constraint violations for class-level constraints.

6.1.3. The error message

The last missing building block is an error message which should be used in case a `@CheckCase` constraint is violated. To define this, create a file `ValidationMessages.properties` with the following contents (see also *Section 4.1, “Default message interpolation”*):

Example 6.5. Defining a custom error message for the `CheckCase` constraint

```
org.hibernate.validator.referenceguide.chapter06.CheckCase.message=Case mode must be {value}.
```

If a validation error occurs, the validation runtime will use the default value, that you specified for the message attribute of the `@CheckCase` annotation to look up the error message in this resource bundle.

6.1.4. Using the constraint

You can now use the constraint in the `Car` class from the *Chapter 1, Getting started* chapter to specify that the licensePlate field should only contain upper-case strings:

Example 6.6. Applying the `@CheckCase` constraint

```
package org.hibernate.validator.referenceguide.chapter06;

public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    @CheckCase(CaseMode.UPPER)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    public Car ( String manufacturer, String licencePlate, int seatCount ) {
```

```
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    //getters and setters ...
}
```

Finally, *Example 6.7, “Validating objects with the @CheckCase constraint”* demonstrates how validating a `Car` object with an invalid license plate causes the `@CheckCase` constraint to be violated.

Example 6.7. Validating objects with the `@CheckCase` constraint

```
//invalid license plate
Car car = new Car( "Morris", "dd-ab-123", 4 );
Set<ConstraintViolation<Car>> constraintViolations =
        validator.validate( car );
assertEquals( 1, constraintViolations.size() );
assertEquals(
        "Case mode must be UPPER.",
        constraintViolations.iterator().next().getMessage()
);

//valid license plate
car = new Car( "Morris", "DD-AB-123", 4 );

constraintViolations = validator.validate( car );

assertEquals( 0, constraintViolations.size() );
```

6.2. Class-level constraints

As discussed earlier, constraints can also be applied on the class level to validate the state of an entire object. Class-level constraints are defined in the same was as are property constraints. *Example 6.8, “Implementing a class-level constraint”* shows constraint annotation and validator of the `@ValidPassengerCount` constraint you already saw in use in *Example 2.3, “Class-level constraint”*.

Example 6.8. Implementing a class-level constraint

```
package org.hibernate.validator.referenceguide.chapter06.classlevel;

@Target({ TYPE, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = { ValidPassengerCountValidator.class })
@Documented
public @interface ValidPassengerCount {

    String message() default "{org.hibernate.validator.referenceguide.chapter06.classlevel." +
```

```
            "ValidPassengerCount.message}";

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

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

```
package org.hibernate.validator.referenceguide.chapter06.classlevel;

public class ValidPassengerCountValidator
        implements ConstraintValidator<ValidPassengerCount, Car> {

    @Override
    public void initialize(ValidPassengerCount constraintAnnotation) {
    }

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

        return car.getPassengers().size() <= car.getSeatCount();
    }
}
```

As the example demonstrates, you need to use the element type `TYPE` in the `@Target` annotation. This allows the constraint to be put on type definitions. The validator of the constraint in the example receives a `Car` in the `isValid()` method and can access the complete object state to decide whether the given instance is valid or not.

6.2.1. Custom property paths

By default the constraint violation for a class-level constraint is reported on the level of the annotated type, e.g. `Car`.

In some cases it is preferable though that the violation's property path refers to one of the involved properties. For instance you might want to report the `@ValidPassengerCount` constraint against the passengers property instead of the `Car` bean.

Example 6.9, “Adding a new ConstraintViolation with custom property path” shows how this can be done by using the constraint validator context passed to `isValid()` to build a custom constraint violation with a property node for the property passengers. Note that you also could add several property nodes, pointing to a sub-entity of the validated bean.

Example 6.9. Adding a new `ConstraintViolation` with custom property path

```
package org.hibernate.validator.referenceguide.chapter06.custompath;

public class ValidPassengerCountValidator
```

```
        implements ConstraintValidator<ValidPassengerCount, Car> {

    @Override
    public void initialize(ValidPassengerCount constraintAnnotation) {
    }

    @Override
    public boolean isValid(Car car, ConstraintValidatorContext constraintValidatorContext) {
        if ( car == null ) {
            return true;
        }

        boolean isValid = car.getPassengers().size() <= car.getSeatCount();

        if ( !isValid ) {
            constraintValidatorContext.disableDefaultConstraintViolation();
            constraintValidatorContext
                    .buildConstraintViolationWithTemplate( "{my.custom.template}" )
                    .addPropertyNode( "passengers" ).addConstraintViolation();
        }

        return isValid;
    }
}
```

6.3. Cross-parameter constraints

Bean Validation distinguishes between two different kinds of constraints.

Generic constraints (which have been discussed so far) apply to the annotated element, e.g. a type, field, method parameter or return value etc. Cross-parameter constraints, in contrast, apply to the array of parameters of a method or constructor and can be used to express validation logic which depends on several parameter values.

In order to define a cross-parameter constraint, its validator class must be annotated with `@SupportedValidationTarget(ValidationTarget.PARAMETERS)`. The type parameter `T` from the `ConstraintValidator` interface must resolve to either `Object` or `Object[]` in order to receive the array of method/constructor arguments in the `isValid()` method.

The following example shows the definition of a cross-parameter constraint which can be used to check that two `Date` parameters of a method are in the correct order:

Example 6.10. Cross-parameter constraint

```
package org.hibernate.validator.referenceguide.chapter06.crossparameter;

@Constraint(validatedBy = ConsistentDateParameterValidator.class)
@Target({ METHOD, CONSTRUCTOR, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
public @interface ConsistentDateParameters {

    String message() default "{org.hibernate.validator.referenceguide.chapter06." +
```

```
            "crossparameter.ConsistentDateParameters.message}";

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

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

The definition of a cross-parameter constraint isn't any different from defining a generic constraint, i.e. it must specify the members `message()`, `groups()` and `payload()` and be annotated with `@Constraint`. This meta annotation also specifies the corresponding validator, which is shown in *Example 6.11, "Generic and cross-parameter constraint"*. Note that besides the element types `METHOD` and `CONSTRUCTOR` also `ANNOTATION_TYPE` is specified as target of the annotation, in order to enable the creation of composed constraints based on `@ConsistentDateParameters` (see *Section 6.4, "Constraint composition"*).

> **Note**
>
> Cross-parameter constraints are specified directly on the declaration of a method or constructor, which is also the case for return value constraints. In order to improve code readability, it is therefore recommended to chose constraint names - such as `@ConsistentDateParameters` - which make the constraint target apparent.

Example 6.11. Generic and cross-parameter constraint

```
package org.hibernate.validator.referenceguide.chapter06.crossparameter;

@SupportedValidationTarget(ValidationTarget.PARAMETERS)
public class ConsistentDateParameterValidator implements
        ConstraintValidator<ConsistentDateParameters, Object[]> {

    @Override
    public void initialize(ConsistentDateParameters constraintAnnotation) {
    }

    @Override
    public boolean isValid(Object[] value, ConstraintValidatorContext context) {
        if ( value.length != 2 ) {
            throw new IllegalArgumentException( "Illegal method signature" );
        }

        //leave null-checking to @NotNull on individual parameters
        if ( value[0] == null || value[1] == null ) {
            return true;
        }

        if ( !( value[0] instanceof Date ) || !( value[1] instanceof Date ) ) {
            throw new IllegalArgumentException(
                    "Illegal method signature, expected two " +
                            "parameters of type Date."
```

```
            );
        }

        return ( (Date) value[0] ).before( (Date) value[1] );
    }
}
```

As discussed above, the validation target `PARAMETERS` must be configured for a cross-parameter validator by using the `@SupportedValidationTarget` annotation. Since a cross-parameter constraint could be applied to any method or constructor, it is considered a best practice to check for the expected number and types of parameters in the validator implementation.

As with generic constraints, `null` parameters should be considered valid and `@NotNull` on the individual parameters should be used to make sure that parameters are not `null`.

> **Tip**
>
> Similar to class-level constraints, you can create custom constraint violations on single parameters instead of all parameters when validating a cross-parameter constraint. Just obtain a node builder from the `ConstraintValidatorContext` passed to `isValid()` and add a parameter node by calling `addParameterNode()`. In the example you could use this to create a constraint violation on the end date parameter of the validated method.

In rare situations a constraint is both, generic and cross-parameter. This is the case if a constraint has a validator class which is annotated with `@SupportedValidationTarget({ValidationTarget.PARAMETERS, ValidationTarget.ANNOTATED_ELEMENT})` or if it has a generic and a cross-parameter validator class.

When declaring such a constraint on a method which has parameters and also a return value, the intended constraint target can't be determined. Constraints which are generic and cross-parameter at the same time, must therefore define a member `validationAppliesTo()` which allows the constraint user to specify the constraint's target as shown in *Example 6.12, “Generic and cross-parameter constraint”*.

Example 6.12. Generic and cross-parameter constraint

```
package org.hibernate.validator.referenceguide.chapter06.crossparameter;

@Constraint(validatedBy = {
        ScriptAssertObjectValidator.class,
        ScriptAssertParametersValidator.class
})
@Target({ TYPE, FIELD, PARAMETER, METHOD, CONSTRUCTOR, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Documented
```

```
public @interface ScriptAssert {

    String message() default "{org.hibernate.validator.referenceguide.chapter06." +
            "crossparameter.ScriptAssert.message}";

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

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

    String script();

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

The `@ScriptAssert` constraint has two validators (not shown), a generic and a cross-parameter one and thus defines the member `validationAppliesTo()`. The default value `IMPLICIT` allows to derive the target automatically in situations where this is possible (e.g. if the constraint is declared on a field or on a method which has parameters but no return value).

If the target can not be determined implicitly, it must be set by the user to either `PARAMETERS` or `RETURN_VALUE` as shown in *Example 6.13, "Specifying the target for a generic and cross-parameter constraint"*.

Example 6.13. Specifying the target for a generic and cross-parameter constraint

```
@ScriptAssert(script = "arg1.size() <= arg0", validationAppliesTo = ConstraintTarget.PARAMETERS)
public Car buildCar(int seatCount, List<Passenger> passengers) {
    //...
}
```

6.4. Constraint composition

Looking at the licensePlate field of the `Car` class in *Example 6.6, "Applying the @CheckCase constraint"*, you see three constraint annotations already. In complexer scenarios, where even more constraints could be applied to one element, this might become a bit confusing easily. Furthermore, if there was a licensePlate field in another class, you would have to copy all constraint declarations to the other class as well, violating the DRY principle.

You can address this kind of problem by creating higher level constraints, composed from several basic constraints. *Example 6.14, "Creating a composing constraint @ValidLicensePlate"* shows a composed constraint annotation which comprises the constraints `@NotNull`, `@Size` and `@CheckCase`:

Example 6.14. Creating a composing constraint `@ValidLicensePlate`

```
package org.hibernate.validator.referenceguide.chapter06.constraintcomposition;
```

```
@NotNull
@Size(min = 2, max = 14)
@CheckCase(CaseMode.UPPER)
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = { })
@Documented
public @interface ValidLicensePlate {

    String message() default "{org.hibernate.validator.referenceguide.chapter06." +
            "constraintcomposition.ValidLicensePlate.message}";

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

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

To create a composed constraint, simply annotate the constraint declaration with its comprising constraints. If the composed constraint itself requires a validator, this validator is to be specified within the `@Constraint` annotation. For composed constraints which don't need an additional validator such as `@ValidLicensePlate`, just set `validatedBy()` to an empty array.

Using the new composed constraint at the licensePlate field is fully equivalent to the previous version, where the three constraints were declared directly at the field itself:

Example 6.15. Application of composing constraint `ValidLicensePlate`

```
package org.hibernate.validator.referenceguide.chapter06.constraintcomposition;

public class Car {

    @ValidLicensePlate
    private String licensePlate;

    //...
}
```

The set of `ConstraintViolations` retrieved when validating a `Car` instance will contain an entry for each violated composing constraint of the `@ValidLicensePlate` constraint. If you rather prefer a single `ConstraintViolation` in case any of the composing constraints is violated, the `@ReportAsSingleViolation` meta constraint can be used as follows:

Example 6.16. Using `@ReportAsSingleViolation`

```
//...
@ReportAsSingleViolation
public @interface ValidLicensePlate {

    String message() default "{org.hibernate.validator.referenceguide.chapter06." +
```

```
            "constraintcomposition.ValidLicensePlate.message}";

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

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

Configuring via XML

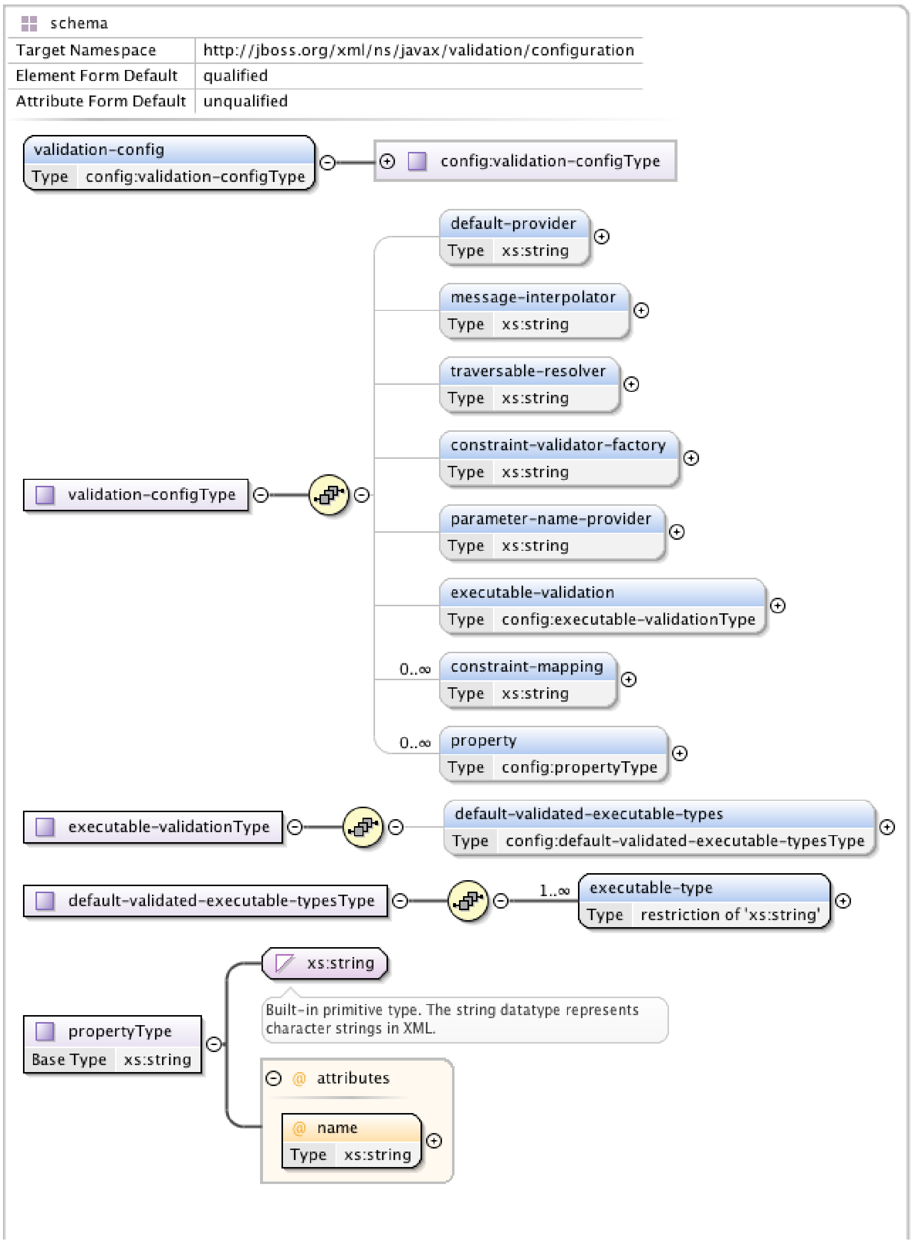
So far we have used the default configuration source for Bean Validation, namely annotations. However, there also exist two kinds of XML descriptors allowing configuration via XML. The first descriptor describes general Bean Validation behaviour and is provided as `META-INF/validation.xml`. The second one describes constraint declarations and closely matches the constraint declaration approach via annotations. Let's have a look at these two document types.

> **Note**
>
> The XSD files are available via *http://www.jboss.org/xml/ns/javax/validation/configuration* [http://www.jboss.org/xml/ns/javax/validation/configuration/] and *http://www.jboss.org/xml/ns/javax/validation/mapping*.

7.1. Configuring the validator factory in `validation.xml`

The key to enable XML configuration for Hibernate Validator is the file `META-INF/validation.xml`. If this file exists on the classpath its configuration will be applied when the `ValidatorFactory` gets created. *Example 7.1, "validation-configuration-1.1.xsd"* shows a model view of the XML schema to which `validation.xml` has to adhere.

Example 7.1. validation-configuration-1.1.xsd



Example 7.2, “validation.xml” shows the several configuration options of `validation.xml`. All settings are optional and the same configuration options are also available programmatically through `javax.validation.Configuration`. In fact the XML configuration will be overridden by values explicitly specified via the programmatic API. It is even possible to ignore the XML configuration completely via `Configuration#ignoreXmlConfiguration()`. See also *Section 8.2, “Configuring a ValidatorFactory”*.

Example 7.2. validation.xml

```
<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">

    <default-provider>com.acme.ValidationProvider</default-provider>

    <message-interpolator>com.acme.MessageInterpolator</message-interpolator>
    <traversable-resolver>com.acme.TraversableResolver</traversable-resolver>
    <constraint-validator-factory>
        com.acme.ConstraintValidatorFactory
    </constraint-validator-factory>
    <parameter-name-provider>com.acme.ParameterNameProvider</parameter-name-provider>

    <executable-validation enabled="true">
        <default-validated-executable-types>
            <executable-type>CONSTRUCTORS</executable-type>
            <executable-type>NON_GETTER_METHODS</executable-type>
            <executable-type>GETTER_METHODS</executable-type>
        </default-validated-executable-types>
    </executable-validation>

    <constraint-mapping>META-INF/validation/constraints-car.xml</constraint-mapping>

    <property name="hibernate.validator.fail_fast">false</property>
</validation-config>
```

Warning

There must only be one file named `META-INF/validation.xml` on the classpath. If more than one is found an exception is thrown.

The node default-provider allows to choose the Bean Validation provider. This is useful if there is more than one provider on the classpath. message-interpolator, traversable-resolver, constraint-validator-factory and parameter-name-provider allow to customize the used implementations for the interfaces `MessageInterpolator`, `TraversableResolver`, `ConstraintValidatorFactory` and `ParameterNameProvider` defined in the `javax.validation` package. See the sub-sections of *Section 8.2, “Configuring a ValidatorFactory”* for more information about these interfaces.

executable-validation and its subnodes define defaults for method validation. The Bean Validation specification defines constructor and non getter methods as defaults. The enabled attribute acts as global switch to turn method validation on and off (see also *Chapter 3, Declaring and validating method constraints*).

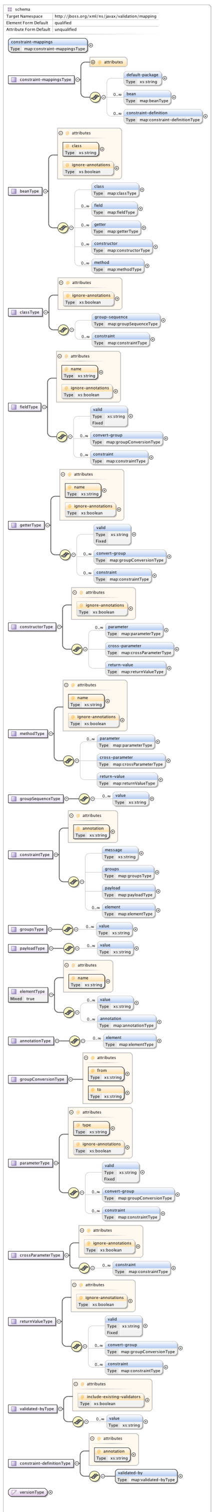
Via the constraint-mapping element you can list an arbitrary number of additional XML files containing the actual constraint configuration. Mapping file names must be specified using their fully-qualified name on the classpath. Details on writing mapping files can be found in the next section.

Last but not least, you can specify provider specific properties via the property nodes. In the example we are using the Hibernate Validator specific hibernate.validator.fail_fast property (see *Section 11.2, “Fail fast mode”*).

7.2. Mapping constraints via constraint-mappings

Expressing constraints in XML is possible via files adhering to the schema seen in *Example 7.3, “validation-mapping-1.1.xsd”*. Note that these mapping files are only processed if listed via constraint-mapping in `validation.xml`.

Example 7.3. validation-mapping-1.1.xsd



Example 7.4, “Bean constraints configured via XML” shows how the classes `Car` and `RentalCar` from *Example 5.3, “Car”* resp. *Example 5.7, “Class RentalCar with redefined default group”* could be mapped in XML.

Example 7.4. Bean constraints configured via XML

```
<constraint-mappings
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/mapping   validation-
mapping-1.1.xsd"
    xmlns="http://jboss.org/xml/ns/javax/validation/mapping" version="1.1">

    <default-package>org.hibernate.validator.referenceguide.chapter05</default-package>
    <bean class="Car" ignore-annotations="true">
        <field name="manufacturer">
            <constraint annotation="javax.validation.constraints.NotNull"/>
        </field>
        <field name="licensePlate">
            <constraint annotation="javax.validation.constraints.NotNull"/>
        </field>
        <field name="seatCount">
            <constraint annotation="javax.validation.constraints.Min">
                <element name="value">2</element>
            </constraint>
        </field>
        <field name="driver">
            <valid/>
        </field>
        <getter name="passedVehicleInspection" ignore-annotations="true">
            <constraint annotation="javax.validation.constraints.AssertTrue">
                <message>The car has to pass the vehicle inspection first</message>
                <groups>
                    <value>CarChecks</value>
                </groups>
                <element name="max">10</element>
            </constraint>
        </getter>
    </bean>
    <bean class="RentalCar" ignore-annotations="true">
        <class ignore-annotations="true">
            <group-sequence>
                <value>RentalCar</value>
                <value>CarChecks</value>
            </group-sequence>
        </class>
    </bean>
    <constraint-definition annotation="org.mycompany.CheckCase">
        <validated-by include-existing-validators="false">
            <value>org.mycompany.CheckCaseValidator</value>
        </validated-by>
    </constraint-definition>
</constraint-mappings>
```

Example 7.5, “Method constraints configured via XML” shows how the constraints from *Example 3.1, “Declaring method and constructor parameter constraints”*, *Example 3.4, “Declaring method and constructor return value constraints”* and *Example 3.3, “Specifying a constraint's target”* can be expressed in XML.

Example 7.5. Method constraints configured via XML

```
<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-1.1.xsd" version="1.1">

    <default-package>org.hibernate.validator.referenceguide.chapter07</default-package>

    <bean class="RentalStation" ignore-annotations="true">
        <constructor>
            <return-value>
                <constraint annotation="ValidRentalStation"/>
            </return-value>
        </constructor>

        <constructor>
            <parameter type="java.lang.String">
                <constraint annotation="javax.validation.constraints.NotNull"/>
            </parameter>
        </constructor>

        <method name="getCustomers">
            <return-value>
                <constraint annotation="javax.validation.constraints.NotNull"/>
                <constraint annotation="javax.validation.constraints.Size">
                    <element name="min">1</element>
                </constraint>
            </return-value>
        </method>

        <method name="rentCar">
            <parameter type="Customer">
                <constraint annotation="javax.validation.constraints.NotNull"/>
            </parameter>
            <parameter type="java.util.Date">
                <constraint annotation="javax.validation.constraints.NotNull"/>
                <constraint annotation="javax.validation.constraints.Future"/>
            </parameter>
            <parameter type="int">
                <constraint annotation="javax.validation.constraints.Min">
                    <element name="value">1</element>
                </constraint>
            </parameter>
        </method>
    </bean>

    <bean class="Garage" ignore-annotations="true">
        <method name="buildCar">
            <parameter type="java.util.List"/>
            <cross-parameter>
                <constraint annotation="ELAssert">
                    <element name="expression">...</element>
                    <element name="validationAppliesTo">PARAMETERS</element>
                </constraint>
```

```
            </cross-parameter>
        </method>
        <method name="paintCar">
            <parameter type="int"/>
            <return-value>
                <constraint annotation="ELAssert">
                    <element name="expression">...</element>
                    <element name="validationAppliesTo">RETURN_VALUE</element>
                </constraint>
            </return-value>
        </method>
    </bean>

</constraint-mappings>
```

The XML configuration is closely mirroring the programmatic API. For this reason it should suffice to just add some comments. default-package is used for all fields where a class name is expected. If the specified class is not fully qualified the configured default package will be used. Every mapping file can then have several bean nodes, each describing the constraints on the entity with the specified class name.

> **Warning**
>
> A given entity can only be configured once across all configuration files. The same applies for constraint definitions for a given constraint annotation. It can only occur in one mapping file. If these rules are violated a `ValidationException` is thrown.

Setting ignore-annotations to `true` means that constraint annotations placed on the configured bean are ignored. The default for this value is `true`. ignore-annotations is also available for the nodes class, fields, getter, constructor, method, parameter, cross-parameter and return-value. If not explicitly specified on these levels the configured bean value applies.

The nodes class, field, getter, constructor and method (and its sub node parameter) determine on which level the constraint gets placed. The constraint node is then used to add a constraint on the corresponding level. Each constraint definition must define the class via the annotation attribute. The constraint attributes required by the Bean Validation specification (message, groups and payload) have dedicated nodes. All other constraint specific attributes are configured using the element node.

The class node also allows to reconfigure the default group sequence (see *Section 5.3, “Redefining the default group sequence”*) via the group-sequence node. Not shown in the example is the use of convert-group to specify group conversions (see *Section 5.4, “Group conversion”*). This node is available on field, getter, parameter and return-value and specifies a from and to attribute to specify the groups.

Last but not least, the list of `ConstraintValidators` associated to a given constraint can be altered via the constraint-definition node. 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`, validators defined on the constraint annotation are ignored. If set to `true`, the list of constraint validators described in XML is concatenated to the list of validators specified on the annotation.

Bootstrapping

In *Section 2.2.1, “Obtaining a Validator instance”* you already saw one way for creating a `Validator` instance - via `Validation#buildDefaultValidatorFactory()`. In this chapter you will learn how to use the other methods in `javax.validation.Validation` in order to bootstrap specifically configured validators.

8.1. Retrieving `ValidatorFactory` and `Validator`

You obtain a `Validator` by retrieving a `ValidatorFactory` via one of the static methods on `javax.validation.Validation` and calling `getValidator()` on the factory instance.

Example 8.1, “Bootstrapping default ValidatorFactory and Validator” shows how to obtain a validator from the default validator factory:

Example 8.1. Bootstrapping default `ValidatorFactory` and `Validator`

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

> **Tip**
>
> The generated `ValidatorFactory` and `Validator` instances are thread-safe and can be cached. As Hibernate Validator uses the factory as context for caching constraint metadata it is recommended to work with one factory instance within an application.

Bean Validation supports working with several providers such as Hibernate Validator within one application. If more than one provider is present on the classpath, it is not guaranteed which one is chosen when creating a factory via `buildDefaultValidatorFactory()`.

In this case you can explicitly specify the provider to use via `Validation#byProvider()`, passing the provider's `ValidationProvider` class as shown in *Example 8.2, “Bootstrapping ValidatorFactory and Validator using a specific provider”*.

Example 8.2. Bootstrapping `ValidatorFactory` and `Validator` using a specific provider

```
ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
        .configure()
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

Note that the configuration object returned by `configure()` allows to specifically customize the factory before calling `buildValidatorFactory()`. The available options are discussed later in this chapter.

Similarly you can retrieve the default validator factory for configuration which is demonstrated in *Example 8.3, “Retrieving the default ValidatorFactory for configuration”*.

Example 8.3. Retrieving the default `ValidatorFactory` for configuration

```
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .configure()
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

> **Note**
>
> If a `ValidatorFactory` instance is no longer in use, it should be disposed by calling `ValidatorFactory#close()`. This will free any resources possibly allocated by the factory.

8.1.1. `ValidationProviderResolver`

By default, available Bean Validation providers are discovered using the *Java Service Provider* [http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html#Service%20Provider] mechanism.

For that purpose, each provider includes the file `META-INF/services/javax.validation.spi.ValidationProvider`, containing the fully qualified classname of its `ValidationProvider` implementation. In the case of Hibernate Validator this is `org.hibernate.validator.HibernateValidator`.

Depending on your environment and its classloading specifics, provider discovery via the Java's service loader mechanism might not work. In this case you can plug in a custom `ValidationProviderResolver` implementation which performs the provider retrieval. An example is OSGi, where you could implement a provider resolver which uses OSGi services for provider discovery.

To use a custom provider resolver pass it via `providerResolver()` as shown shown in *Example 8.4, “Using a custom ValidationProviderResolver”*.

Example 8.4. Using a custom `ValidationProviderResolver`

```
package org.hibernate.validator.referenceguide.chapter08;

public class OsgiServiceDiscoverer implements ValidationProviderResolver {
```

```
    @Override
    public List<ValidationProvider<?>> getValidationProviders() {
        //...
    }
}
```

```
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .providerResolver( new OsgiServiceDiscoverer() )
        .configure()
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

8.2. Configuring a `ValidatorFactory`

By default validator factories retrieved from `Validation` and any validators they create are configured as per the XML descriptor `META-INF/validation.xml` (see *Chapter 7, Configuring via XML*), if present.

If you want to disable the XML based configuration, you can do so by invoking `Configuration#ignoreXmlConfiguration()`.

The different values of the XML configuration can be accessed via `Configuration#getBootstrapConfiguration()`. This can for instance be helpful if you want to integrate Bean Validation into a managed environment and want to create managed instances of the objects configured via XML.

Using the fluent configuration API, you can override one or more of the settings when bootstrapping the factory. The following sections show how to make use of the different options. Note that the `Configuration` class exposes the default implementations of the different extension points which can be useful if you want to use these as delegates for your custom implementations.

8.2.1. `MessageInterpolator`

Message interpolators are used by the validation engine to create user readable error messages from constraint message descriptors.

In case the default message interpolation algorithm described in *Chapter 4, Interpolating constraint error messages* is not sufficient for your needs, you can pass in your own implementation of the `MessageInterpolator` interface via `Configuration#messageInterpolator()` as shown in *Example 8.5, "Using a custom MessageInterpolator"*.

Example 8.5. Using a custom `MessageInterpolator`

```
package org.hibernate.validator.referenceguide.chapter08;
```

```
public class MyMessageInterpolator implements MessageInterpolator {

    @Override
    public String interpolate(String messageTemplate, Context context) {
        //...
    }

    @Override
    public String interpolate(String messageTemplate, Context context, Locale locale) {
        //...
    }
}
```

```
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .configure()
        .messageInterpolator( new MyMessageInterpolator() )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

8.2.2. `TraversableResolver`

In some cases the validation engine should not access the state of a bean property. The most obvious example for that is a lazily loaded property or association of a JPA entity. Validating this lazy property or association would mean that its state would have to be accessed, triggering a load from the database.

Which properties can be accessed and which ones not is controlled by querying the `TraversableResolver` interface. *Example 8.6, "Using a custom TraversableResolver"* shows how to use a custom traversable resolver implementation.

Example 8.6. Using a custom `TraversableResolver`

```
package org.hibernate.validator.referenceguide.chapter08;

public class MyTraversableResolver implements TraversableResolver {

    @Override
    public boolean isReachable(
            Object traversableObject,
            Node traversableProperty,
            Class<?> rootBeanType,
            Path pathToTraversableObject,
            ElementType elementType) {
        //...
    }

    @Override
    public boolean isCascadable(
            Object traversableObject,
            Node traversableProperty,
            Class<?> rootBeanType,
```

```
            Path pathToTraversableObject,
            ElementType elementType) {
        //...
    }
}
```

```
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .configure()
        .traversableResolver( new MyTraversableResolver() )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

Hibernate Validator provides two `TraversableResolvers` out of the box which will be enabled automatically depending on your environment. The first is `DefaultTraversableResolver` which will always return `true` for `isReachable()` and `isTraversable()`. The second is `JPATraversableResolver` which gets enabled when Hibernate Validator is used in combination with JPA 2.

8.2.3. `ConstraintValidatorFactory`

`ConstraintValidatorFactory` is the extension point for customizing how constraint validators are instantiated and released.

The default `ConstraintValidatorFactory` provided by Hibernate Validator requires a public no-arg constructor to instantiate `ConstraintValidator` instances (see *Section 6.1.2, “The constraint validator”*). Using a custom `ConstraintValidatorFactory` offers for example the possibility to use dependency injection in constraint validator implementations.

To configure a custom constraint validator factory call `Configuration#constraintValidatorFactory()` (see *Example 8.7, “Using a custom ConstraintValidatorFactory”*.

Example 8.7. Using a custom `ConstraintValidatorFactory`

```
package org.hibernate.validator.referenceguide.chapter08;

public class MyConstraintValidatorFactory implements ConstraintValidatorFactory {

    @Override
    public <T extends ConstraintValidator<?, ?>> T getInstance(Class<T> key) {
        //...
    }

    @Override
    public void releaseInstance(ConstraintValidator<?, ?> instance) {
        //...
    }
}
```

```
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .configure()
        .constraintValidatorFactory( new MyConstraintValidatorFactory() )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

> **Warning**
>
> Any constraint implementations relying on `ConstraintValidatorFactory` behaviors specific to an implementation (dependency injection, no no-arg constructor and so on) are not considered portable.

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

8.2.4. `ParameterNameProvider`

In case a method or constructor parameter constraint is violated, the `ParameterNameProvider` interface is used to retrieve the parameter's name and make it available to the user via the constraint violation's property path.

The default implementation returns parameter names in the form `arg0`, `arg1` etc., while custom implementations could e.g. be based on parameter annotations, debug symbols or a feature for retrieving parameter names at runtime possibly provided by future Java versions.

Custom `ParameterNameProvider` implementations are used as demonstrated in *Example 8.8, “Using a custom ParameterNameProvider”*.

Example 8.8. Using a custom `ParameterNameProvider`

```
package org.hibernate.validator.referenceguide.chapter08;

public class MyParameterNameProvider implements ParameterNameProvider {

    @Override
    public List<String> getParameterNames(Constructor<?> constructor) {
        //...
    }

    @Override
    public List<String> getParameterNames(Method method) {
        //...
```

```
    }
}
```

```
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .configure()
        .parameterNameProvider( new MyParameterNameProvider() )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

> **Tip**
>
> Hibernate Validator comes with a custom `ParameterNameProvider` implementation based on the *ParaNamer* [http://paranamer.codehaus.org/] library which provides several ways for obtaining parameter names at runtime. Refer to *Section 11.7, “ParaNamer based ParameterNameProvider”* to learn more about this specific implementation.

8.2.5. Adding mapping streams

As discussed earlier you can configure the constraints applying for your Java beans using XML based constraint mappings.

Besides the mapping files specified in `META-INF/validation.xml` you can add further mappings via `Configuration#addMapping()` (see *Example 8.9, “Adding constraint mapping streams”*). Note that the passed input stream(s) must adhere to the XML schema for constraint mappings presented in *Section 7.2, “Mapping constraints via constraint-mappings”*.

Example 8.9. Adding constraint mapping streams

```
InputStream constraintMapping1 = ...;
InputStream constraintMapping2 = ...;
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .configure()
        .addMapping( constraintMapping1 )
        .addMapping( constraintMapping2 )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

You should close any passed input stream after the validator factory has been created.

8.2.6. Provider-specific settings

Via the configuration object returned by `Validation#byProvider()` provider specific options can be configured.

In case of Hibernate Validator this e.g. allows you to enable the fail fast mode and pass one or more programmatic constraint mappings as demonstrated in *Example 8.10, “Setting Hibernate Validator specific options”*.

Example 8.10. Setting Hibernate Validator specific options

```
ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
        .configure()
        .failFast( true )
        .addMapping( (ConstraintMapping) null )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

Alternatively, provider-specific options can be passed via `Configuration#addProperty()`. Hibernate Validator supports enabling the fail fast mode that way, too:

Example 8.11. Enabling a Hibernate Validator specific option via `addProperty()`

```
ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
        .configure()
        .addProperty( "hibernate.validator.fail_fast", "true" )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

Refer to *Section 11.2, “Fail fast mode”* and *Section 11.3, “Programmatic constraint declaration”* to learn more about the fail fast mode and the constraint declaration API.

8.3. Configuring a `Validator`

When working with a configured validator factory it can occasionally be required to apply a different configuration to a single `Validator` instance. *Example 8.12, “Configuring a Validator via usingContext()”* shows how this can be achieved by calling `ValidatorFactory#usingContext()`.

Example 8.12. Configuring a `Validator` via `usingContext()`

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

Validator validator = validatorFactory.usingContext()
        .messageInterpolator( new MyMessageInterpolator() )
        .traversableResolver( new MyTraversableResolver() )
        .getValidator();
```

Using constraint metadata

The Bean Validation specification provides not only a validation engine, but also an API for retrieving constraint metadata in a uniform way, no matter whether the constraints are declared using annotations or via XML mappings. Read this chapter to learn more about this API and its possibilities. You can find all the metadata API types in the package javax.validation.metadata.

The examples presented in this chapter are based on the classes and constraint declarations shown in *Example 9.1, "Example classes"*.

Example 9.1. Example classes

```
package org.hibernate.validator.referenceguide.chapter07;

public class Person {

    public interface Basic {
    }

    @NotNull
    private String name;

    //getters and setters ...
}
```

```
public interface Vehicle {

    public interface Basic {
    }

    @NotNull(groups = Vehicle.Basic.class)
    String getManufacturer();
}
```

```
@ValidCar
public class Car implements Vehicle {

    public interface SeverityInfo extends Payload {
    }

    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    private Person driver;

    private String modelName;
```

```
    public Car() {
    }

    public Car(
            @NotNull String manufacturer,
            String licencePlate,
            Person driver,
            String modelName) {

        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.driver = driver;
        this.modelName = modelName;
    }

    public void driveAway(@Max(75) int speed) {
        //...
    }

    @LuggageCountMatchesPassengerCount(
            piecesOfLuggagePerPassenger = 2,
            validationAppliesTo = ConstraintTarget.PARAMETERS,
            payload = SeverityInfo.class,
            message = "There must not be more than {piecesOfLuggagePerPassenger} pieces of " +
                    "luggage per passenger."
    )
    public void load(List<Person> passengers, List<PieceOfLuggage> luggage) {
        //...
    }

    @Override
    @Size(min = 3)
    public String getManufacturer() {
        return manufacturer;
    }

    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

    @Valid
    @ConvertGroup(from = Default.class, to = Person.Basic.class)
    public Person getDriver() {
        return driver;
    }

    //further getters and setters...
}
```

9.1. `BeanDescriptor`

The entry point into the metadata API is the method `Validator#getConstraintsForClass()`, which returns an instance of the *`BeanDescriptor`* [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/index.html?javax/validation/metadata/BeanDescriptor.html] interface. Using this descriptor, you can obtain metadata for constraints declared directly on the bean itself (class-or property-level), but also retrieve metadata descriptors representing single properties, methods and constructors.

Example 9.2, “Using BeanDescriptor” demonstrates how to retrieve a `BeanDescriptor` for the `Car` class and how to use this descriptor in form of assertions.

> **Tip**
>
> If a constraint declaration hosted by the requested class is invalid, a `ValidationException` is thrown.

Example 9.2. Using `BeanDescriptor`

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

BeanDescriptor carDescriptor = validator.getConstraintsForClass( Car.class );

assertTrue( carDescriptor.isBeanConstrained() );

//one class-level constraint
assertEquals( 1, carDescriptor.getConstraintDescriptors().size() );

//manufacturer, licensePlate, driver
assertEquals( 3, carDescriptor.getConstrainedProperties().size() );

//property has constraint
assertNotNull( carDescriptor.getConstraintsForProperty( "licensePlate" ) );

//property is marked with @Valid
assertNotNull( carDescriptor.getConstraintsForProperty( "driver" ) );

//constraints from getter method in interface and implementation class are returned
assertEquals(
        2,
        carDescriptor.getConstraintsForProperty( "manufacturer" )
                .getConstraintDescriptors()
                .size()
);

//property is not constrained
assertNull( carDescriptor.getConstraintsForProperty( "modelName" ) );

//driveAway(int), load(List<Person>, List<PieceOfLuggage>)
assertEquals( 2, carDescriptor.getConstrainedMethods( MethodType.NON_GETTER ).size() );

//driveAway(int), getManufacturer(), getDriver(), load(List<Person>, List<PieceOfLuggage>)
assertEquals(
        4,
        carDescriptor.getConstrainedMethods( MethodType.NON_GETTER, MethodType.GETTER )
                .size()
);

//driveAway(int)
```

```
assertNotNull( carDescriptor.getConstraintsForMethod( "driveAway", int.class ) );

//getManufacturer()
assertNotNull( carDescriptor.getConstraintsForMethod( "getManufacturer" ) );

//setManufacturer() is not constrained
assertNull( carDescriptor.getConstraintsForMethod( "setManufacturer", String.class ) );

//Car(String, String, Person, String)
assertEquals( 1, carDescriptor.getConstrainedConstructors().size() );

//Car(String, String, Person, String)
assertNotNull(
        carDescriptor.getConstraintsForConstructor(
                String.class,
                String.class,
                Person.class,
                String.class
        )
);
```

You can determine whether the specified class hosts any class- or property-level constraints via `isBeanConstrained()`. Method or constructor constraints are not considered by `isBeanConstrained()`.

The method `getConstraintDescriptors()` is common to all descriptors derived from `ElementDescriptor` (see *Section 9.4, "ElementDescriptor"*) and returns a set of descriptors representing the constraints directly declared on the given element. In case of `BeanDescriptor`, the bean's class-level constraints are returned. More details on `ConstraintDescriptor` can be found in *Section 9.6, "ConstraintDescriptor"*.

Via `getConstraintsForProperty()`, `getConstraintsForMethod()` and `getConstraintsForConstructor()` you can obtain a descriptor representing one given property or executable element, identified by its name and, in case of methods and constructors, parameter types. The different descriptor types returned by these methods are described in the following sections.

Note that these methods consider constraints declared at super-types according to the rules for constraint inheritance as described in *Section 2.1.4, "Constraint inheritance"*. An example is the descriptor for the `manufacturer` property, which provides access to all constraints defined on `Vehicle#getManufacturer()` and the implementing method `Car#getManufacturer()`. `null` is returned in case the specified element does not exist or is not constrained.

The methods `getConstrainedProperties()`, `getConstrainedMethods()` and `getConstrainedConstructors()` return (potentially empty) sets with all constrained properties, methods and constructors, respectively. An element is considered constrained, if it has at least one constraint or is marked for cascaded validation. When invoking `getConstrainedMethods()`, you can specify the type of the methods to be returned (getters, non-getters or both).

9.2. `PropertyDescriptor`

The interface *`PropertyDescriptor`* [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/index.html?javax/validation/metadata/PropertyDescriptor.html] represents one given property of a class. It is transparent whether constraints are declared on a field or a property getter, provided the JavaBeans naming conventions are respected. *Example 9.3, “Using PropertyDescriptor”* shows how to use the `PropertyDescriptor` interface.

Example 9.3. Using `PropertyDescriptor`

```
PropertyDescriptor licensePlateDescriptor = carDescriptor.getConstraintsForProperty(
        "licensePlate"
);

//"licensePlate" has two constraints, is not marked with @Valid and defines no group conversions
assertEquals( "licensePlate", licensePlateDescriptor.getPropertyName() );
assertEquals( 2, licensePlateDescriptor.getConstraintDescriptors().size() );
assertTrue( licensePlateDescriptor.hasConstraints() );
assertFalse( licensePlateDescriptor.isCascaded() );
assertTrue( licensePlateDescriptor.getGroupConversions().isEmpty() );

PropertyDescriptor driverDescriptor = carDescriptor.getConstraintsForProperty( "driver" );

//"driver" has no constraints, is marked with @Valid and defines one group conversion
assertEquals( "driver", driverDescriptor.getPropertyName() );
assertTrue( driverDescriptor.getConstraintDescriptors().isEmpty() );
assertFalse( driverDescriptor.hasConstraints() );
assertTrue( driverDescriptor.isCascaded() );
assertEquals( 1, driverDescriptor.getGroupConversions().size() );
```

Using `getConstrainedDescriptors()`, you can retrieve a set of `ConstraintDescriptors` providing more information on the individual constraints of a given property. The method `isCascaded()` returns `true`, if the property is marked for cascaded validation (either using the `@Valid` annotation or via XML), `false` otherwise. Any configured group conversions are returned by `getGroupConversions()`. See *Section 9.5, “GroupConversionDescriptor”* for more details on `GroupConversionDescriptor`.

9.3. `MethodDescriptor` and `ConstructorDescriptor`

Constrained methods and constructors are represented by the interfaces *`MethodDescriptor`* [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/index.html?javax/validation/metadata/MethodDescriptor.html] and *`ConstructorDescriptor`* [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/index.html?javax/validation/metadata/ConstructorDescriptor.html], respectively. *Example 9.4, “Using MethodDescriptor and ConstructorDescriptor”* demonstrates how to work with these descriptors.

Example 9.4. Using `MethodDescriptor` and `ConstructorDescriptor`

```
//driveAway(int) has a constrained parameter and an unconstrained return value
MethodDescriptor driveAwayDescriptor = carDescriptor.getConstraintsForMethod(
        "driveAway",
        int.class
);
assertEquals( "driveAway", driveAwayDescriptor.getName() );
assertTrue( driveAwayDescriptor.hasConstrainedParameters() );
assertFalse( driveAwayDescriptor.hasConstrainedReturnValue() );

//always returns an empty set; constraints are retrievable by navigating to
//one of the sub-descriptors, e.g. for the return value
assertTrue( driveAwayDescriptor.getConstraintDescriptors().isEmpty() );

ParameterDescriptor speedDescriptor = driveAwayDescriptor.getParameterDescriptors()
        .get( 0 );

//The "speed" parameter is located at index 0, has one constraint and is not cascaded
//nor does it define group conversions
assertEquals( "arg0", speedDescriptor.getName() );
assertEquals( 0, speedDescriptor.getIndex() );
assertEquals( 1, speedDescriptor.getConstraintDescriptors().size() );
assertFalse( speedDescriptor.isCascaded() );
assert speedDescriptor.getGroupConversions().isEmpty();

//getDriver() has no constrained parameters but its return value is marked for cascaded
//validation and declares one group conversion
MethodDescriptor getDriverDescriptor = carDescriptor.getConstraintsForMethod(
        "getDriver"
);
assertFalse( getDriverDescriptor.hasConstrainedParameters() );
assertTrue( getDriverDescriptor.hasConstrainedReturnValue() );

ReturnValueDescriptor returnValueDescriptor = getDriverDescriptor.getReturnValueDescriptor();
assertTrue( returnValueDescriptor.getConstraintDescriptors().isEmpty() );
assertTrue( returnValueDescriptor.isCascaded() );
assertEquals( 1, returnValueDescriptor.getGroupConversions().size() );

//load(List<Person>, List<PieceOfLuggage>) has one cross-parameter constraint
MethodDescriptor loadDescriptor = carDescriptor.getConstraintsForMethod(
        "load",
        List.class,
        List.class
);
assertTrue( loadDescriptor.hasConstrainedParameters() );
assertFalse( loadDescriptor.hasConstrainedReturnValue() );
assertEquals(
        1,
        loadDescriptor.getCrossParameterDescriptor().getConstraintDescriptors().size()
);

//Car(String, String, Person, String) has one constrained parameter
ConstructorDescriptor constructorDescriptor = carDescriptor.getConstraintsForConstructor(
        String.class,
        String.class,
        Person.class,
```

```
        String.class
);

assertEquals( "Car", constructorDescriptor.getName() );
assertFalse( constructorDescriptor.hasConstrainedReturnValue() );
assertTrue( constructorDescriptor.hasConstrainedParameters() );
assertEquals(
        1,
        constructorDescriptor.getParameterDescriptors()
                .get( 0 )
                .getConstraintDescriptors()
                .size()
);
```

`getName()` returns the name of the given method or constructor. The methods `hasConstrainedParameters()` and `hasConstrainedReturnValue()` can be used to perform a quick check whether an executable element has any parameter constraints (either constraints on single parameters or cross-parameter constraints) or return value constraints.

Note that any constraints are not directly exposed on `MethodDescriptor` and `ConstructorDescriptor`, but rather on dedicated descriptors representing an executable's parameters, its return value and its cross-parameter constraints. To get hold of one of these descriptors, invoke `getParameterDescriptors()`, `getReturnValueDescriptor()` or `getCrossParameterDescriptor()`, respectively.

These descriptors provide access to the element's constraints (`getConstraintDescriptors()`) and, in case of parameters and return value, to its configuration for cascaded validation (`isValid()` and `getGroupConversions()`). For parameters, you also can retrieve the index and the name, as returned by the currently used parameter name provider (see *Section 8.2.4, "ParameterNameProvider"*) via `getName()` and `getIndex()`.

Tip

Getter methods following the JavaBeans naming conventions are considered as bean properties but also as constrained methods.

That means you can retrieve the related metadata either by obtaining a `PropertyDescriptor` (e.g. `BeanDescriptor.getConstraintsForProperty("foo")`) or by examining the return value descriptor of the getter's `MethodDescriptor` (e.g. `BeanDescriptor.getConstraintsForMethod("getFoo").getReturnValueDescri`

9.4. `ElementDescriptor`

The *`ElementDiscriptor`* [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/index.html?javax/validation/metadata/ElementDescriptor.html] interface is the common base class for the individual descriptor types such as `BeanDescriptor`, `PropertyDescriptor` etc. Besides `getConstraintDescriptors()` it provides some more methods common to all descriptors.

`hasConstraints()` allows for a quick check whether an element has any direct constraints (e.g. class-level constraints in case of `BeanDescriptor`). `getElementClass()` returns the Java type of the element represented by a given descriptor. More specifically, the method returns

- the object type when invoked on `BeanDescriptor`,
- the type of a property or parameter when invoked on `PropertyDescriptor` or `ParameterDescriptor` respectively,
- `Object[].class` when invoked on `CrossParameterDescriptor`,
- the return type when invoked on `ConstructorDescriptor`, `MethodDescriptor` or `ReturnValueDescriptor`. `void.class` will be returned for methods which don't have a return value.

Example 9.5, “Using ElementDescriptor methods” shows how these methods are used.

Example 9.5. Using `ElementDescriptor` methods

```
PropertyDescriptor manufacturerDescriptor = carDescriptor.getConstraintsForProperty(
        "manufacturer"
);

assertTrue( manufacturerDescriptor.hasConstraints() );
assertEquals( String.class, manufacturerDescriptor.getElementClass() );

CrossParameterDescriptor loadCrossParameterDescriptor = carDescriptor.getConstraintsForMethod(
        "load",
        List.class,
        List.class
).getCrossParameterDescriptor();

assertTrue( loadCrossParameterDescriptor.hasConstraints() );
assertEquals( Object[].class, loadCrossParameterDescriptor.getElementClass() );
```

Finally, `ElementDescriptor` offers access to the `ConstraintFinder` API which allows you to query for constraint metadata in a fine grained way. *Example 9.6, “Usage of ConstraintFinder”* shows how to retrieve a `ConstraintFinder` instance via `findConstraints()` and use the API to query for constraint metadata.

Example 9.6. Usage of ConstraintFinder

```
PropertyDescriptor manufacturerDescriptor = carDescriptor.getConstraintsForProperty(
        "manufacturer"
);

//"manufacturer" constraints are declared on the getter, not the field
```

```
assertTrue(
        manufacturerDescriptor.findConstraints()
                .declaredOn( ElementType.FIELD )
                .getConstraintDescriptors()
                .isEmpty()
);

//@NotNull on Vehicle#getManufacturer() is part of another group
assertEquals(
        1,
        manufacturerDescriptor.findConstraints()
                .unorderedAndMatchingGroups( Default.class )
                .getConstraintDescriptors()
                .size()
);

//@Size on Car#getManufacturer()
assertEquals(
        1,
        manufacturerDescriptor.findConstraints()
                .lookingAt( Scope.LOCAL_ELEMENT )
                .getConstraintDescriptors()
                .size()
);

//@Size on Car#getManufacturer() and @NotNull on Vehicle#getManufacturer()
assertEquals(
        2,
        manufacturerDescriptor.findConstraints()
                .lookingAt( Scope.HIERARCHY )
                .getConstraintDescriptors()
                .size()
);

//Combining several filter options
assertEquals(
        1,
        manufacturerDescriptor.findConstraints()
                .declaredOn( ElementType.METHOD )
                .lookingAt( Scope.HIERARCHY )
                .unorderedAndMatchingGroups( Vehicle.Basic.class )
                .getConstraintDescriptors()
                .size()
);
```

Via `declaredOn()` you can search for `ConstraintDescriptors` declared on certain element types. This is useful to find property constraints declared on either fields or getter methods.

`unorderedAndMatchingGroups()` restricts the resulting constraints to those matching the given validation group(s).

`lookingAt()` allows to distinguish between constraints directly specified on the element (`Scope.LOCAL_ELEMENT`) or constraints belonging to the element but hosted anywhere in the class hierarchy (`Scope.HIERARCHY`).

You can also combine the different options as shown in the last example.

> **Warning**
>
> Order is not respected by `unorderedAndMatchingGroups()`, but group inheritance and inheritance via sequence are.

9.5. `GroupConversionDescriptor`

All those descriptor types that represent elements which can be subject of cascaded validation (i.e., `PropertyDescriptor`, `ParameterDescriptor` and `ReturnValueDescriptor`) provide access to the element's group conversions via `getGroupConversions()`. The returned set contains a *`GroupConversionDescriptor`* [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/index.html?javax/validation/metadata/GroupConversionDescriptor.html] for each configured conversion, allowing to retrieve source and target groups of the conversion. *Example 9.7, "Using GroupConversionDescriptor"* shows an example.

Example 9.7. Using `GroupConversionDescriptor`

```
PropertyDescriptor driverDescriptor = carDescriptor.getConstraintsForProperty( "driver" );

Set<GroupConversionDescriptor> groupConversions = driverDescriptor.getGroupConversions();
assertEquals( 1, groupConversions.size() );

GroupConversionDescriptor groupConversionDescriptor = groupConversions.iterator()
        .next();
assertEquals( Default.class, groupConversionDescriptor.getFrom() );
assertEquals( Person.Basic.class, groupConversionDescriptor.getTo() );
```

9.6. `ConstraintDescriptor`

Last but not least, the *`ConstraintDescriptor`* [http://docs.jboss.org/hibernate/beanvalidation/spec/1.1/api/index.html?javax/validation/metadata/ConstraintDescriptor.html] interface describes a single constraint together with its composing constraints. Via an instance of this interface you get access to the constraint annotation and its parameters.

Example 9.8, "Using ConstraintDescriptor" shows how to retrieve default constraint attributes (such as message template, groups etc.) as well as custom constraint attributes (`piecesOfLuggagePerPassenger`) and other metadata such as the constraint's annotation type and its validators from a `ConstraintDescriptor`.

Example 9.8. Using `ConstraintDescriptor`

```
//descriptor for the @LuggageCountMatchesPassengerCount constraint on the
//load(List<Person>, List<PieceOfLuggage>) method
ConstraintDescriptor<?> constraintDescriptor = carDescriptor.getConstraintsForMethod(
        "load",
```

```
        List.class,
        List.class
).getCrossParameterDescriptor().getConstraintDescriptors().iterator().next();

//constraint type
assertEquals(
        LuggageCountMatchesPassengerCount.class,
        constraintDescriptor.getAnnotation().annotationType()
);

//standard constraint attributes
assertEquals( SeverityInfo.class, constraintDescriptor.getPayload().iterator().next() );
assertEquals(
        ConstraintTarget.PARAMETERS,
        constraintDescriptor.getValidationAppliesTo()
);
assertEquals( Default.class, constraintDescriptor.getGroups().iterator().next() );
assertEquals(
        "There must not be more than {piecesOfLuggagePerPassenger} pieces of luggage per " +
        "passenger.",
        constraintDescriptor.getMessageTemplate()
);

//custom constraint attribute
assertEquals(
        2,
        constraintDescriptor.getAttributes().get( "piecesOfLuggagePerPassenger" )
);

//no composing constraints
assertTrue( constraintDescriptor.getComposingConstraints().isEmpty() );

//validator class
assertEquals(
        Arrays.<Class<?>>asList( LuggageCountMatchesPassengerCount.Validator.class ),
        constraintDescriptor.getConstraintValidatorClasses()
);
```

Integrating with other frameworks

Hibernate Validator is intended to be used to implement multi-layered data validation, where constraints are expressed in a single place (the annotated domain model) and checked in various different layers of the application. For this reason there are multiple integration points with other technologies.

10.1. ORM integration

Hibernate Validator integrates with both Hibernate and all pure Java Persistence providers.

> **Tip**
>
> When lazy loaded associations are supposed to be validated it is recommended to place the constraint on the getter of the association. Hibernate replaces lazy loaded associations with proxy instances which get initialized/loaded when requested via the getter. If, in such a case, the constraint is placed on field level the actual proxy instance is used which will lead to validation errors.

10.1.1. Database schema-level validation

Out of the box, Hibernate (as of version 3.5.x) will translate the constraints you have defined for your entities into mapping metadata. For example, if a property of your entity is annotated `@NotNull`, its columns will be declared as `not null` in the DDL schema generated by Hibernate.

If, for some reason, the feature needs to be disabled, set `hibernate.validator.apply_to_ddl` to `false`. See also *Table 2.2, “Bean Validation constraints”* and *Table 2.3, “Custom constraints”*.

You can also limit the DDL constraint generation to a subset of the defined constraints by setting the property org.hibernate.validator.group.ddl. The property specifies the comma-separated, fully specified class names of the groups a constraint has to be part of in order to be considered for DDL schema generation.

10.1.2. Hibernate event-based validation

Hibernate Validator has a built-in Hibernate event listener - *`org.hibernate.cfg.beanvalidation.BeanValidationEventListener`* [http://fisheye.jboss.org/browse/Hibernate/core/trunk/annotations/src/main/java/org/hibernate/cfg/beanvalidation/BeanValidationEventListener.java] - which is part of Hibernate Annotations (as of Hibernate 3.5.x). Whenever a `PreInsertEvent`, `PreUpdateEvent` or `PreDeleteEvent` occurs, the listener will verify all constraints of the entity instance and throw an exception if any constraint is violated. Per default objects will be checked before any inserts or updates are made by Hibernate. Pre deletion events will per default not trigger a validation. You can configure the groups to be validated per event type using the properties javax.persistence.validation.group.pre-persist, javax.persistence.validation.group.pre-update and javax.persistence.validation.group.pre-remove. The values of these properties are the comma-separated, fully specified class names of the groups to validate. *Example 10.1, “Manual configuration of BeanValidationEvenListener”* shows the default values for these properties. In this case they could also be omitted.

On constraint violation, the event will raise a runtime `ConstraintViolationException` which contains a set of `ConstraintViolations` describing each failure.

If Hibernate Validator is present in the classpath, Hibernate Annotations (or Hibernate EntityManager) will use it transparently. To avoid validation even though Hibernate Validator is in the classpath set javax.persistence.validation.mode to `none`.

> **Note**
>
> If the beans are not annotated with validation annotations, there is no runtime performance cost.

In case you need to manually set the event listeners for Hibernate Core, use the following configuration in `hibernate.cfg.xml`:

Example 10.1. Manual configuration of `BeanValidationEvenListener`

```
<hibernate-configuration>
    <session-factory>
        ...
        <property name="javax.persistence.validation.group.pre-persist">
            javax.validation.groups.Default
        </property>
        <property name="javax.persistence.validation.group.pre-update">
            javax.validation.groups.Default
        </property>
        <property name="javax.persistence.validation.group.pre-remove"></property>
        ...
        <event type="pre-update">
            <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
        </event>
        <event type="pre-insert">
            <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
        </event>
        <event type="pre-delete">
            <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

10.1.3. JPA

If you are using JPA 2 and Hibernate Validator is in the classpath the JPA2 specification requires that Bean Validation gets enabled. The properties javax.persistence.validation.group.pre-persist, javax.persistence.validation.group.pre-update and javax.persistence.validation.group.pre-remove as described in *Section 10.1.2, “Hibernate event-based validation”* can in this case be configured in `persistence.xml`. `persistence.xml` also defines a node validation-mode which can be set to `AUTO`, `CALLBACK`, `NONE`. The default is `AUTO`.

In a JPA 1 you will have to create and register Hibernate Validator yourself. In case you are using Hibernate EntityManager you can add a customized version of the `BeanValidationEventListener` described in *Section 10.1.2, “Hibernate event-based validation”* to your project and register it manually.

10.2. JSF & Seam

When working with JSF2 or JBoss Seam™ and Hibernate Validator (Bean Validation) is present in the runtime environment, validation is triggered for every field in the application. *Example 10.2, “Usage of Bean Validation within JSF2”* shows an example of the f:validateBean tag in a JSF page. The validationGroups attribute is optional and can be used to specify a comma seperated list of validation groups. The default is `javax.validation.groups.Default`. For more information refer to the Seam documentation or the JSF 2 specification.

Example 10.2. Usage of Bean Validation within JSF2

```
<h:form>
  <f:validateBean validationGroups="javax.validation.groups.Default">
    <h:inputText value=##{model.property}# />
    <h:selectOneRadio value=##{model.radioProperty}# > ... </h:selectOneRadio>
    <!-- other input components here -->
  </f:validateBean>
</h:form>
```

Tip

The integration between JSF 2 and Bean Validation is described in the "Bean Validation Integration" chapter of *JSR-314* [http://jcp.org/en/jsr/detail?id=314]. It is interesting to know that JSF 2 implements a custom `MessageInterpolator` to ensure ensure proper localization. To encourage the use of the Bean Validation message facility, JSF 2 will per default only display the generated Bean Validation message. This can, however, be configured via the application resource bundle by providing the following configuration (*{0}* is replaced with the Bean Validation message and *{1}* is replaced with the JSF component label):

```
javax.faces.validator.BeanValidator.MESSAGE={1}: {0}
```

The default is:

```
javax.faces.validator.BeanValidator.MESSAGE={0}
```

10.3. CDI

As of version 1.1, Bean Validation is integrated with CDI (Contexts and Dependency Injection for Java™ EE).

This integration provides CDI managed beans for `Validator` and `ValidatorFactory` and enables dependency injection in constraint validators as well as custom message interpolators, traversable resolvers, constraint validator factories and parameter name providers.

Furthermore, parameter and return value constraints on the methods and constructors of CDI managed beans will automatically be validated upon invocation.

When your application runs on a Jave EE container, this integration is enabled by default. When working with CDI in a Servlet container or in a pure Java SE environment, you can use the CDI portable extension provided by Hibernate Validator. To do so, add the portable extension to your class path as described in *Section 1.1.2, “CDI”*.

10.3.1. Dependency injection

CDI's dependency injection mechanism makes it very easy to retrieve `ValidatorFactory` and `Validator` instances and use them in your managed beans. Just annotate instance fields of your bean with `@javax.inject.Inject` as shown in *Example 10.3, “Retrieving validator factory and validator via @Inject”*.

Example 10.3. Retrieving validator factory and validator via `@Inject`

```
package org.hibernate.validator.referenceguide.chapter10.cdi.validator;

@ApplicationScoped
public class RentalStation {

    @Inject
    private ValidatorFactory validatorFactory;

    @Inject
    private Validator validator;

    //...
}
```

The injected beans are the default validator factory and validator instances. In order to configure them - e.g. to use a custom message interpolator - you can use the Bean Validation XML descriptors as discussed in *Chapter 7, Configuring via XML*.

If you are working with several Bean Validation providers you can make sure that factory and validator from Hibernate Validator are injected by annotating the injection points with the `@HibernateValidator` qualifier which is demonstrated in *Example 10.4, “Using the @HibernateValidator qualifier annotation”*.

Example 10.4. Using the @HibernateValidator qualifier annotation

```
package org.hibernate.validator.referenceguide.chapter10.cdi.validator.qualifier;

@ApplicationScoped
public class RentalStation {

    @Inject
    @HibernateValidator
    private ValidatorFactory validatorFactory;

    @Inject
    @HibernateValidator
    private Validator validator;

    //...
}
```

> **Tip**
>
> The fully-qualified name of the qualifier annotation is `org.hibernate.validator.cdi.HibernateValidator`. Be sure to not import `org.hibernate.validator.HibernateValidator` instead which is the `ValidationProvider` implementation used for selecting Hibernate Validator when working with the bootstrapping API (see *Section 8.1, “Retrieving ValidatorFactory and Validator”*).

Via `@Inject` you also can inject dependencies into constraint validators and other Bean Validation objects such as `MessageInterpolator` implementations etc.

Example 10.5, “Constraint validator with injected bean” demonstrates how an injected CDI bean is used in a `ConstraintValidator` implementation to determine whether the given constraint is valid or not. As the example shows, you also can work with the `@PostConstruct` and `@PreDestroy` callbacks to implement any required construction and destruction logic.

Example 10.5. Constraint validator with injected bean

```
package org.hibernate.validator.referenceguide.chapter10.cdi.injection;

public class ValidLicensePlateValidator
        implements ConstraintValidator<ValidLicensePlate, String> {
```

```
    @Inject
    private VehicleRegistry vehicleRegistry;

    @PostConstruct
    public void postConstruct() {
        //do initialization logic...
    }

    @PreDestroy
    public void preDestroy() {
        //do destruction logic...
    }

    @Override
    public void initialize(ValidLicensePlate constraintAnnotation) {
    }

    @Override
    public boolean isValid(String licensePlate, ConstraintValidatorContext constraintContext) {
        return vehicleRegistry.isValidLicensePlate( licensePlate );
    }
}
```

10.3.2. Method validation

The method interception facilities of CDI allow for a very tight integration with Bean Validation's method validation functionality. Just put constraint annotations to the parameters and return values of the executables of your CDI beans and they will be validated automatically before (parameter constraints) and after (return value constraints) a method or constructor is invoked.

Note that no explicit interceptor binding is required, instead the required method validation interceptor will automatically be registered for all managed beans with constrained methods and constructors.

You can see an example in *Example 10.6, “CDI managed beans with method-level constraints”*.

Example 10.6. CDI managed beans with method-level constraints

```
package org.hibernate.validator.referenceguide.chapter10.cdi.methodvalidation;

@ApplicationScoped
public class RentalStation {

    @Valid
    public RentalStation() {
        //...
    }

    @NotNull
    @Valid
    public Car rentCar(
            @NotNull Customer customer,
            @NotNull @Future Date startDate,
```

```
            @Min(1) int durationInDays) {
        //...
    }

    @NotNull
    List<Car> getAvailableCars() {
        //...
    }
}
```

```
package org.hibernate.validator.referenceguide.chapter10.cdi.methodvalidation;

@RequestScoped
public class RentCarRequest {

    @Inject
    private RentalStation rentalStation;

    public void rentCar(String customerId, Date startDate, int duration) {
        //causes ConstraintViolationException
        rentalStation.rentCar( null, null, -1 );
    }
}
```

Here the `RentalStation` bean hosts several method constraints. When invoking one of the `RentalStation` methods from another bean such as `RentCarRequest`, the constraints of the invoked method are automatically validated. If any illegal parameter values are passed as in the example, a `ConstraintViolationException` will be thrown by the method interceptor, providing detailed information on the violated constraints. The same is the case if the method's return value violates any return value constraints.

Similarly, constructor constraints are validated automatically upon invocation. In the example the `RentalStation` object returned by the constructor will be validated since the constructor return value is marked with `@Valid`.

10.3.2.1. Validated executable types

Bean Validation allows for a fine-grained control of the executable types which are automatically validated. By default, constraints on constructors and non-getter methods are validated. Therefore the `@NotNull` constraint on the method `RentalStation#getAvailableCars()` in *Example 10.6, "CDI managed beans with method-level constraints"* gets not validated when the method is invoked.

You have the following options to configure which types of executables are validated upon invocation:

- Configure the executable types globally via the XML descriptor `META-INF/validation.xml`; see *Section 7.1, "Configuring the validator factory in validation.xml"* for an example
- Use the `@ValidateOnExecution` annotation on the executable or type level

If several sources of configuration are specified for a given executable, @ValidateOnExecution on the executable level takes precedence over @ValidateOnExecution on the type level and @ValidateOnExecution generally takes precedence over the globally configured types in META-INF/validation.xml.

Example 10.7, “Using @ValidateOnExecution” shows how to use the @ValidateOnExecution annotation:

Example 10.7. Using @ValidateOnExecution

```
package org.hibernate.validator.referenceguide.chapter10.cdi.methodvalidation.configuration;

@ApplicationScoped
@ValidateOnExecution(type = ExecutableType.ALL)
public class RentalStation {

    @Valid
    public RentalStation() {
        //...
    }

    @NotNull
    @Valid
    @ValidateOnExecution(type = ExecutableType.NONE)
    public Car rentCar(
            @NotNull Customer customer,
            @NotNull @Future Date startDate,
            @Min(1) int durationInDays) {
        //...
    }

    @NotNull
    public List<Car> getAvailableCars() {
        //...
    }
}
```

Here the method rentCar() won't be validated upon invocation because it is annotated with @ValidateOnExecution(type = ExecutableType.NONE). In contrast, the constructor and the method getAvailableCars() will be validated due to @ValidateOnExecution(type = ExecutableType.ALL) being given on the type level. ExecutableType.ALL is a more compact form for explicitly specifying all the types CONSTRUCTORS, GETTER_METHODS and NON_GETTER_METHODS.

Tip

Executable validation can be turned off globally by specifying <executable-validation enabled="false"/> in META-INF/validation.xml. In this case, any @ValidateOnExecution annotations are ignored.

Note that when a method overrides or implements a super-type method the configuration will be taken from that overridden or implemented method (as given via `@ValidateOnExecution` on the method itself or on the super-type). This protects a client of the super-type method from an unexpected alteration of the configuration, e.g. disabling validation of an overridden executable in a sub-type.

In case a CDI managed bean overrides or implements a super-type method and this super-type method hosts any constraints, it can happen that the validation interceptor is not properly registered with the bean, resulting in the bean's methods not being validated upon invocation. In this case you can specify the executable type IMPLICIT on the sub-class as shown in *Example 10.8, "Using ExecutableType.IMPLICIT"*, which makes sure that all required metadata is discovered an the validation interceptor kicks in when the methods on `ExpressRentalStation` are invoked.

Example 10.8. Using `ExecutableType.IMPLICIT`

```
package org.hibernate.validator.referenceguide.chapter10.cdi.methodvalidation.implicit;

@ValidateOnExecution(type = ExecutableType.ALL)
public interface RentalStation {

    @NotNull
    @Valid
    Car rentCar(
            @NotNull Customer customer,
            @NotNull @Future Date startDate,
            @Min(1) int durationInDays);
}
```

```
package org.hibernate.validator.referenceguide.chapter10.cdi.methodvalidation.implicit;

@ApplicationScoped
@ValidateOnExecution(type = ExecutableType.IMPLICIT)
public class ExpressRentalStation implements RentalStation {

    @Override
    public Car rentCar(Customer customer, Date startDate, @Min(1) int durationInDays) {
        //...
    }
}
```

10.4. Java EE

When your application runs on a Java EE application server such as WildFly, you also can obtain `Validator` and `ValidatorFactory` instances via `@Resource` injection in managed objects such as EJBs etc., as shown in *Example 10.9, "Retrieving Validator and ValidatorFactory via @Resource injection"*.

Example 10.9. Retrieving `Validator` and `ValidatorFactory` via `@Resource` injection

```
package org.hibernate.validator.referenceguide.chapter10.javaee;

@Stateless
public class RentalStationBean {

    @Resource
    private ValidatorFactory validatorFactory;

    @Resource
    private Validator validator;

    //...
}
```

Alternatively you can obtain a validator and a validator factory from JNDI under the names "java:comp/Validator" and "java:comp/ValidatorFactory", respectively.

Similar to CDI-based injection via `@Inject`, these objects represent default validator and validator factory and thus can be configured using the XML descriptor `META-INF/validation.xml` (see *Chapter 7, Configuring via XML*).

When your application is CDI-enabled, the injected objects are CDI-aware as well and e.g. support dependency injection in constraint validators.

Hibernate Validator Specifics

In this chapter you will learn how to make use of several features provided by Hibernate Validator in addition to the functionality defined by the Bean Validation specification. This includes the fail fast mode, the API for programmatic constraint configuration and the boolean composition of constraints.

> **Note**
>
> Using the features described in the following sections may result in application code which is not portable between Bean Validation providers.

11.1. Public API

Let's start, however, with a look at the public API of Hibernate Validator. *Table 11.1, "Hibernate Validator public API"* lists all packages belonging to this API and describes their purpose. Note that when a package is part of the public this is not necessarily true for its sub-packages.

Table 11.1. Hibernate Validator public API

Packages	Description
org.hibernate.validator	Classes used by the Bean Validation bootstrap mechanism (eg. validation provider, configuration class); For more details see *Chapter 8, Bootstrapping*.
org.hibernate.validator.cfg, org.hibernate.validator.cfg.context, org.hibernate.validator.cfg.defs	Hibernate Validator's fluent API for constraint declaration; In org.hibernate.validator.cfg you will find the `ConstraintMapping` interface and in org.hibernate.validator.cfg.defs all constraint definitions. Refer to *Section 11.3, "Programmatic constraint declaration"* for the details.
org.hibernate.validator.constraints, org.hibernate.validator.constraints.br	Some useful custom constraints provided by Hibernate Validator in addition to the built-in constraints defined by the Bean Validation specification; The constraints are described in detail in *Section 2.3.2, "Additional constraints"*.
org.hibernate.validator.constraintvalidation	Extended constraint validator context which allows to set custom attributes for message interpolation. *Section 11.6.1, "HibernateConstraintValidatorContext"* describes how to make use of that feature.

Packages	Description
org.hibernate.validator.group, org.hibernate.validator.spi.group	The group sequence provider feature which allows you to define dynamic default group sequences in function of the validated object state; The specifics can be found in *Section 5.3, "Redefining the default group sequence"*.
org.hibernate.validator.messageinterpolation, org.hibernate.validator.resourceloading, org.hibernate.validator.spi.resourceloading	Classes related to constraint message interpolation; The first package contains Hibernate Validator's default message interpolator, `ResourceBundleMessageInterpolator`. The latter two packages provide the `ResourceBundleLocator` SPI for the loading of resource bundles (see *Section 4.2.1, "ResourceBundleLocator"*) and its default implementation.
org.hibernate.validator.parameternameprovider	A `ParameterNameProvider` based on the ParaNamer library, see *Section 11.7, "ParaNamer based ParameterNameProvider"*.
org.hibernate.validator.valuehandling, org.hibernate.validator.spi.valuehandling	Classes related to the processing of values prior to their validation, see *Section 11.8, "Unwrapping values prior to validation"*.

Note

The public packages of Hibernate Validator fall into two categories: while the actual API parts are intended to be *invoked* or *used* by clients (e.g. the API for programmatic constraint declaration or the custom constraints), the SPI (service provider interface) packages contain interfaces which are intended to be *implemented* by clients (e.g. `ResourceBundleLocator`).

Any packages not listed in that table are internal packages of Hibernate Validator and are not intended to be accessed by clients. The contents of these internal packages can change from release to release without notice, thus possibly breaking any client code relying on it.

11.2. Fail fast mode

Using the fail fast mode, Hibernate Validator allows to return from the current validation as soon as the first constraint violation occurs. This can be useful for the validation of large object graphs where you are only interested in a quick check whether there is any constraint violation at all.

Example 11.1, “Using the fail fast validation mode” shows how to bootstrap and use a fail fast enabled validator.

Example 11.1. Using the fail fast validation mode

```
package org.hibernate.validator.referenceguide.chapter11.failfast;

public class Car {

    @NotNull
    private String manufacturer;

    @AssertTrue
    private boolean isRegistered;

    public Car(String manufacturer, boolean isRegistered) {
        this.manufacturer = manufacturer;
        this.isRegistered = isRegistered;
    }

    //getters and setters...
}
```

```
Validator validator = Validation.byProvider( HibernateValidator.class )
        .configure()
        .failFast( true )
        .buildValidatorFactory()
        .getValidator();

Car car = new Car( null, false );

Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );

assertEquals( 1, constraintViolations.size() );
```

Here the validated object actually fails to satisfy both the constraints declared on the `Car` class, yet the validation call yields only one `ConstraintViolation` since the fail fast mode is enabled.

> **Note**
>
> There is no guarantee in which order the constraints are evaluated, i.e. it is not deterministic whether the returned violation originates from the `@NotNull` or the `@AssertTrue` constraint. If required, a deterministic evaluation order can be enforced using group sequences as described in *Section 5.2, “Defining group sequences”*.

Refer to *Section 8.2.6, “Provider-specific settings”* to learn about the different ways of enabling the fail fast mode when bootstrapping a validator.

11.3. Programmatic constraint declaration

As per the Bean Validation specification, you can declare constraints using Java annotations and XML based constraint mappings.

In addition, Hibernate Validator provides a fluent API which allows for the programmatic configuration of constraints. Use cases include the dynamic addition of constraints at runtime depending on some application state or tests where you need entities with different constraints in different scenarios but don't want to implement actual Java classes for each test case.

By default, constraints added via the fluent API are additive to constraints configured via the standard configuration capabilities. But it is also possible to ignore annotation and XML configured constraints where required.

The API is centered around the `ConstraintMapping` interface. You obtain a new mapping via `HibernateValidatorConfiguration#createConstraintMapping()` which you then can configure in a fluent manner as shown in *Example 11.2, “Programmatic constraint declaration”*.

Example 11.2. Programmatic constraint declaration

```
HibernateValidatorConfiguration configuration = Validation
        .byProvider( HibernateValidator.class )
        .configure();

ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .property( "manufacturer", FIELD )
            .constraint( new NotNullDef() )
        .property( "licensePlate", FIELD )
            .ignoreAnnotations()
            .constraint( new NotNullDef() )
            .constraint( new SizeDef().min( 2 ).max( 14 ) )
    .type( RentalCar.class )
        .property( "rentalStation", METHOD )
            .constraint( new NotNullDef() );

Validator validator = configuration.addMapping( constraintMapping )
        .buildValidatorFactory()
        .getValidator();
```

Constraints can be configured on multiple classes and properties using method chaining. The constraint definition classes `NotNullDef` and `SizeDef` are helper classes which allow to configure constraint parameters in a type-safe fashion. Definition classes exist for all built-in constraints in the `org.hibernate.validator.cfg.defs` package. By calling `ignoreAnnotations()` any constraints configured via annotations or XML are ignored for the given element.

> **Note**
>
> Each element (type, property, method etc.) may only be configured once within all the constraint mappings used to set up one validator factory. Otherwise a `ValidationException` is raised.

> **Note**
>
> It is not supported to add constraints to non-overridden supertype properties and methods by configuring a subtype. Instead you need to configure the supertype in this case.

Having configured the mapping, you must add it back to the configuration object from which you then can obtain a validator factory.

For custom constraints you can either create your own definition classes extending `ConstraintDef` or you can use `GenericConstraintDef` as seen in *Example 11.3, “Programmatic declaration of a custom constraint”*.

Example 11.3. Programmatic declaration of a custom constraint

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .property( "licensePlate", FIELD )
            .constraint( new GenericConstraintDef<CheckCase>( CheckCase.class )
                .param( "value", CaseMode.UPPER )
            );
```

By invoking `valid()` you can mark a member for cascaded validation which is equivalent to annotating it with `@Valid`. Configure any group conversions to be applied during cascaded validation using the `convertGroup()` method (equivalent to `@ConvertGroup`). An example can be seen in *Example 11.4, “Marking a property for cascaded validation”*.

Example 11.4. Marking a property for cascaded validation

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .property( "driver", FIELD )
            .constraint( new NotNullDef() )
            .valid()
```

```
            .convertGroup( Default.class ).to( PersonDefault.class )
    .type( Person.class )
        .property( "name", FIELD )
            .constraint( new NotNullDef().groups( PersonDefault.class ) );
```

You can not only configure bean constraints using the fluent API but also method and constructor constraints. As shown in *Example 11.5, “Programmatic declaration of method and constructor constraints”* constructors are identified by their parameter types and methods by their name and parameter types. Having selected a method or constructor, you can mark its parameters and/or return value for cascaded validation and add constraints as well as cross-parameter constraints.

Example 11.5. Programmatic declaration of method and constructor constraints

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .constructor( String.class )
            .parameter( 0 )
                .constraint( new SizeDef().min( 3 ).max( 50 ) )
            .returnValue()
                .valid()
        .method( "drive", int.class )
            .parameter( 0 )
                .constraint( new MaxDef().value ( 75 ) )
        .method( "load", List.class, List.class )
            .crossParameter()
                .constraint( new GenericConstraintDef<LuggageCountMatchesPassengerCount>(
                        LuggageCountMatchesPassengerCount.class ).param(
                            "piecesOfLuggagePerPassenger", 2
                        )
                )
        .method( "getDriver" )
            .returnValue()
                .constraint( new NotNullDef() )
                .valid();
```

Last but not least you can configure the default group sequence or the default group sequence provider of a type as shown in the following example.

Example 11.6. Configuration of default group sequence and default group sequence provider

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .defaultGroupSequence( Car.class, CarChecks.class )
    .type( RentalCar.class )
```

```
        .defaultGroupSequenceProviderClass( RentalCarGroupSequenceProvider.class );
```

11.4. Boolean composition of constraints

Bean Validation specifies that the constraints of a composed constraint (see *Section 6.4, “Constraint composition”*) are all combined via a logical *AND*. This means all of the composing constraints need to return `true` in order for an overall successful validation.

Hibernate Validator offers an extension to this and allows you to compose constraints via a logical *OR* or *NOT*. To do so you have to use the `ConstraintComposition` annotation and the enum `CompositionType` with its values *AND*, *OR* and *ALL_FALSE*.

Example 11.7, “OR composition of constraints” shows how to build a composed constraint `@PatternOrSize` where only one of the composing constraints needs to be valid in order to pass the validation. Either the validated string is all lower-cased or it is between two and three characters long.

Example 11.7. OR composition of constraints

```
package org.hibernate.validator.referenceguide.chapter11.booleancomposition;

@ConstraintComposition(OR)
@Pattern(regexp = "[a-z]")
@Size(min = 2, max = 3)
@ReportAsSingleViolation
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = { })
public @interface PatternOrSize {
    String message() default "{org.hibernate.validator.referenceguide.chapter11." +
            "booleancomposition.PatternOrSize.message}";

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

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

Tip

Using *ALL_FALSE* as composition type implicitly enforces that only a single violation will get reported in case validation of the constraint composition fails.

11.5. `ResourceBundleLocator`

As described in *Section 4.2, “Custom message interpolation”*, Bean Validation allows to plug in custom message interpolator implementations.

With `ResourceBundleLocator`, Hibernate Validator provides an additional SPI which allows to retrieve error messages from other resource bundles than `ValidationMessages` while still using the actual interpolation algorithm as defined by the specification. Refer to *Section 4.2.1, “ResourceBundleLocator”* to learn how to make use of that SPI.

11.6. Custom contexts

The Bean Validation specification offers at several points in its API the possibility to unwrap a given interface to a implementor specific subtype. In the case of constraint violation creation in `ConstraintValidator` implementations as well as message interpolation in `Messageinterpolator` instances, there exist `unwrap()` methods for the provided context instances - `ConstraintValidatorContext` respectively `MessageInterpolatorContext`. Hibernate Validator provides custom extensions for both of these interfaces.

11.6.1. `HibernateConstraintValidatorContext`

`HibernateConstraintValidatorContext` is a subtype of `ConstraintValidatorContext` which allows you to set arbitrary parameters for interpolation via the Expression Language message interpolation facility (see *Section 4.1.2, “Interpolation with message expressions”*). For example the default error message for the `@Future` constraint is "must be in the future". What if you would like to include the current date to make the message more explicit? *Example 11.8, “Custom @Future validator with message parameters”* shows how this could be achieved.

Example 11.8. Custom `@Future` validator with message parameters

```
public class MyFutureValidator implements ConstraintValidator<Future, Date> {

    public void initialize(Future constraintAnnotation) {
    }

    public boolean isValid(Date value, ConstraintValidatorContext context) {
        Date now = GregorianCalendar.getInstance().getTime();

        if ( value.before( now ) ) {
            HibernateConstraintValidatorContext hibernateContext =
                    context.unwrap( HibernateConstraintValidatorContext.class );

            hibernateContext.disableDefaultConstraintViolation();
            hibernateContext.addExpressionVariable( "now", now )
                    .buildConstraintViolationWithTemplate( "Must be after ${now}" )
                    .addConstraintViolation();

            return false;
        }

        return true;
    }
}
```

> **Note**
>
> Note that the parameters specified via `addExpressionVariable(String, Object)` are global and apply for all constraint violations created by this `isValid()` invocation. This includes the default constraint violation, but also all violations created by the `ConstraintViolationBuilder`. You can, however, update the parameters between invocations of `ConstraintViolationBuilder#addConstraintViolation()`.

> **Warning**
>
> This functionality is currently experimental and might change in future versions.

11.6.2. `HibernateMessageInterpolatorContext`

Hibernate Validator also offers a custom extension of `MessageInterpolatorContext`, namely `HibernateMessageInterpolatorContext` (see *Example 11.9, “HibernateMessageInterpolatorContext”*). This subtype was introduced to allow a better integration of Hibernate Validator into the Glassfish. The root bean type was in this case needed to determine the right classloader for the message resource bundle. If you have any other usecases, let us know.

Example 11.9. `HibernateMessageInterpolatorContext`

```
public interface HibernateMessageInterpolatorContext extends MessageInterpolator.Context {

    /**
     * Returns the currently validated root bean type.
     *
     * @return The currently validated root bean type.
     */
    Class<?> getRootBeanType();
}
```

11.7. ParaNamer based `ParameterNameProvider`

Hibernate Validator comes with a `ParameterNameProvider` implementation which leverages the *ParaNamer* [http://paranamer.codehaus.org/] library.

This library provides several ways for obtaining parameter names at runtime, e.g. based on debug symbols created by the Java compiler, constants with the parameter names woven into the bytecode in a post-compile step or annotations such as the `@Named` annotation from JSR 330.

In order to use `ParanamerParameterNameProvider`, either pass an instance when bootstrapping a validator as shown in *Example 8.8, “Using a custom ParameterNameProvider”* or specify org.hibernate.validator.parameternameprovider.ParanamerParameterNameProvider as value for the <parameter-name-provider> element in the `META-INF/validation.xml` file.

> **Tip**
>
> When using this parameter name provider, you need to add the ParaNamer library to your classpath. It is available in the Maven Central repository with the group id `com.thoughtworks.paranamer` and the artifact id `paranamer`.

By default `ParanamerParameterNameProvider` retrieves parameter names from constants added to the byte code at build time (via `DefaultParanamer`) and debug symbols (via `BytecodeReadingParanamer`). Alternatively you can specify a `Paranamer` implementation of your choice when creating a `ParanamerParameterNameProvider` instance.

11.8. Unwrapping values prior to validation

Sometimes it is required to unwrap values prior to the validation. E.g. in *Example 11.10, “Using@UnwrapValidatedValue”* property types as specified by *JavaFX* [http://docs.oracle.com/javafx/] are used to define an element of some domain model.

Example 11.10. Using`@UnwrapValidatedValue`

```
@Size(min = 3)
@UnwrapValidatedValue
private Property<String> name = new SimpleStringProperty( "Bob" );
```

> **Note**
>
> The concept of value unwrapping is considered experimental at this time and may evolve into more general means of value handling in future releases. Please let us know about your use cases for such functionality.

In JavaFX, bean properties are typically not of simple data types like `String` or `int`, but are wrapped in `Property` types which allows to make them observable, use them for data binding etc. When applying a constraint such as `@Size` to an element of type `Property<String>` without further preparation, an exception would be raised, indicating that no suitable validator for that constraint and data type can be found. Thus the validated value must be unwrapped from the containing property object before looking up a validator and invoking it.

To do so, put the `@UnwrapValidatedValue` annotation to the element in question. This will advice the validation engine to look for an unwrapper implementation which returns the data type to be used for constraint validator resolution and unwraps the validated value. Unwrapper types must extend the SPI class `ValidatedValueUnwrapper` as shown in *Example 11.11, “Implementing the ValidatedValueUnwrapper interface”*.

Example 11.11. Implementing the `ValidatedValueUnwrapper` interface

```
public class PropertyValueUnwrapper extends ValidatedValueUnwrapper<Property<?>> {

    @Override
    public Object handleValidatedValue(Property<?> value) {
        //...
    }

    @Override
    public Type getValidatedValueType(Type valueType) {
        //...
    }
}
```

Value unwrappers must be registered when obtaining a `Validator` instance as follows:

Example 11.12. Registering a `ValidatedValueUnwrapper`

```
Validator validator = Validation.byProvider( HibernateValidator.class )
        .configure()
        .addValidatedValueHandler( new PropertyValueUnwrapper() )
        .buildValidatorFactory()
        .getValidator();
```

Several unwrapper implementations can be registered when working with different kinds of wrapper types in one application. Note that it is not specified which of the unwrapper implementations is chosen when more than one implementation is suitable to unwrap a given element.

Alternatively, the fully-qualified names of one ore more unwrapper implementations can be specified via the configuration property hibernate.validator.validated_value_handlers which can be useful when configuring the default validator factory using the descriptor `META-INF/validation.xml` (see *Chapter 7, Configuring via XML*).

Annotation Processor

Have you ever caught yourself by unintentionally doing things like

- specifying constraint annotations at unsupported data types (e.g. by annotating a String with `@Past`)
- annotating the setter of a JavaBeans property (instead of the getter method)
- annotating static fields/methods with constraint annotations (which is not supported)?

Then the Hibernate Validator Annotation Processor is the right thing for you. It helps preventing such mistakes by plugging into the build process and raising compilation errors whenever constraint annotations are incorrectly used.

Tip

You can find the Hibernate Validator Annotation Processor as part of the distribution bundle on *Sourceforge* [http://sourceforge.net/projects/hibernate/files/hibernate-validator] or in the usual Maven repositories such as Maven Central under the GAV org.hibernate:hibernate-validator-annotation-processor:5.1.3.Final.

12.1. Prerequisites

The Hibernate Validator Annotation Processor is based on the "Pluggable Annotation Processing API" as defined by *JSR 269* [http://jcp.org/en/jsr/detail?id=269] which is part of the Java Platform since Java 6.

12.2. Features

As of Hibernate Validator 5.1.3.Final the Hibernate Validator Annotation Processor checks that:

- constraint annotations are allowed for the type of the annotated element
- only non-static fields or methods are annotated with constraint annotations
- only non-primitive fields or methods are annotated with `@Valid`
- only such methods are annotated with constraint annotations which are valid JavaBeans getter methods (optionally, see below)
- only such annotation types are annotated with constraint annotations which are constraint annotations themselves

- definition of dynamic default group sequence with `@GroupSequenceProvider` is valid

12.3. Options

The behavior of the Hibernate Validator Annotation Processor can be controlled using the *processor options* [http://java.sun.com/javase/6/docs/technotes/tools/windows/javac.html#options] listed in table *Table 12.1, “Hibernate Validator Annotation Processor options”*:

Table 12.1. Hibernate Validator Annotation Processor options

Option	Explanation
`diagnosticKind`	Controls how constraint problems are reported. Must be the string representation of one of the values from the enum `javax.tools.Diagnostic.Kind`, e.g. `WARNING`. A value of `ERROR` will cause compilation to halt whenever the AP detects a constraint problem. Defaults to `ERROR`.
`methodConstraintsSupported`	Controls whether constraints are allowed at methods of any kind. Must be set to `true` when working with method level constraints as supported by Hibernate Validator. Can be set to `false` to allow constraints only at JavaBeans getter methods as defined by the Bean Validation API. Defaults to `true`.
`verbose`	Controls whether detailed processing information shall be displayed or not, useful for debugging purposes. Must be either `true` or `false`. Defaults to `false`.

12.4. Using the Annotation Processor

This section shows in detail how to integrate the Hibernate Validator Annotation Processor into command line builds (javac, Ant, Maven) as well as IDE-based builds (Eclipse, IntelliJ IDEA, NetBeans).

12.4.1. Command line builds

12.4.1.1. javac

When compiling on the command line using *javac* [http://java.sun.com/javase/6/docs/technotes/guides/javac/index.html], specify the JAR hibernate-validator-annotation-processor-5.1.3.Final.jar using the "processorpath" option as shown in the following listing. The processor will be detected automatically by the compiler and invoked during compilation.

Example 12.1. Using the annotation processor with javac

```
javac src/main/java/org/hibernate/validator/ap/demo/Car.java \
   -cp /path/to/validation-api-1.1.0.Final.jar \
   -processorpath /path/to/hibernate-validator-annotation-processor-5.1.3.Final.jar
```

12.4.1.2. Apache Ant

Similar to directly working with javac, the annotation processor can be added as as compiler argument when invoking the *javac task* [http://ant.apache.org/manual/CoreTasks/javac.html] for *Apache Ant* [http://ant.apache.org/]:

Example 12.2. Using the annotation processor with Ant

```
<javac srcdir="src/main"
       destdir="build/classes"
       classpath="/path/to/validation-api-1.1.0.Final.jar">
       <compilerarg value="-processorpath" />
      <compilerarg value="/path/to/hibernate-validator-annotation-processor-5.1.3.Final.jar"/>
</javac>
```

12.4.1.3. Maven

There are several options for integrating the annotation processor with *Apache Maven* [http://maven.apache.org/]. Generally it is sufficient to add the Hibernate Validator Annotation Processor as dependency to your project:

Example 12.3. Adding the HV Annotation Processor as dependency

```
...
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>5.1.3.Final</version>
</dependency>
...
```

The processor will then be executed automatically by the compiler. This basically works, but comes with the disadavantage that in some cases messages from the annotation processor are not displayed (see *MCOMPILER-66* [http://jira.codehaus.org/browse/MCOMPILER-66]).

Another option is using the *Maven Annotation Plugin* [http://code.google.com/p/maven-annotation-plugin/]. To work with this plugin, disable the standard annotation processing performed by the compiler plugin and configure the annotation plugin by specifying an execution and adding the Hibernate Validator Annotation Processor as plugin dependency (that way the processor is not visible on the project's actual classpath):

Example 12.4. Configuring the Maven Annotation Plugin

```
...
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
        <compilerArgument>-proc:none</compilerArgument>
    </configuration>
</plugin>
<plugin>
    <groupId>org.bsc.maven</groupId>
    <artifactId>maven-processor-plugin</artifactId>
    <version>2.2.1</version>
    <executions>
        <execution>
            <id>process</id>
            <goals>
                <goal>process</goal>
            </goals>
            <phase>process-sources</phase>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator-annotation-processor</artifactId>
            <version>5.1.3.Final</version>
        </dependency>
    </dependencies>
</plugin>
...
```

12.4.2. IDE builds

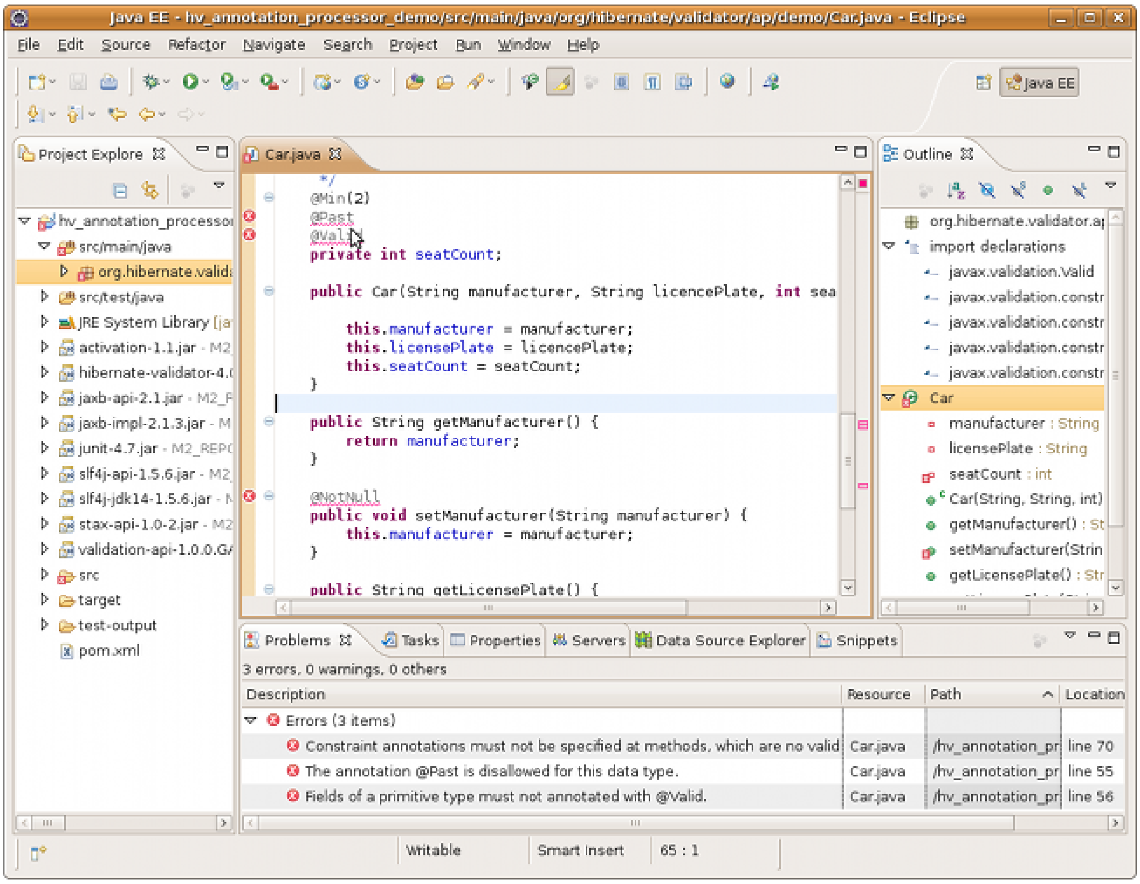
12.4.2.1. Eclipse

Do the following to use the annotation processor within the *Eclipse* [http://www.eclipse.org/] IDE:

- Right-click your project, choose "Properties"
- Go to "Java Compiler" and make sure, that "Compiler compliance level" is set to "1.6". Otherwise the processor won't be activated
- Go to "Java Compiler - Annotation Processing" and choose "Enable annotation processing"
- Go to "Java Compiler - Annotation Processing - Factory Path" and add the JAR hibernate-validator-annotation-processor-5.1.3.Final.jar

- Confirm the workspace rebuild

You now should see any annotation problems as regular error markers within the editor and in the "Problem" view:

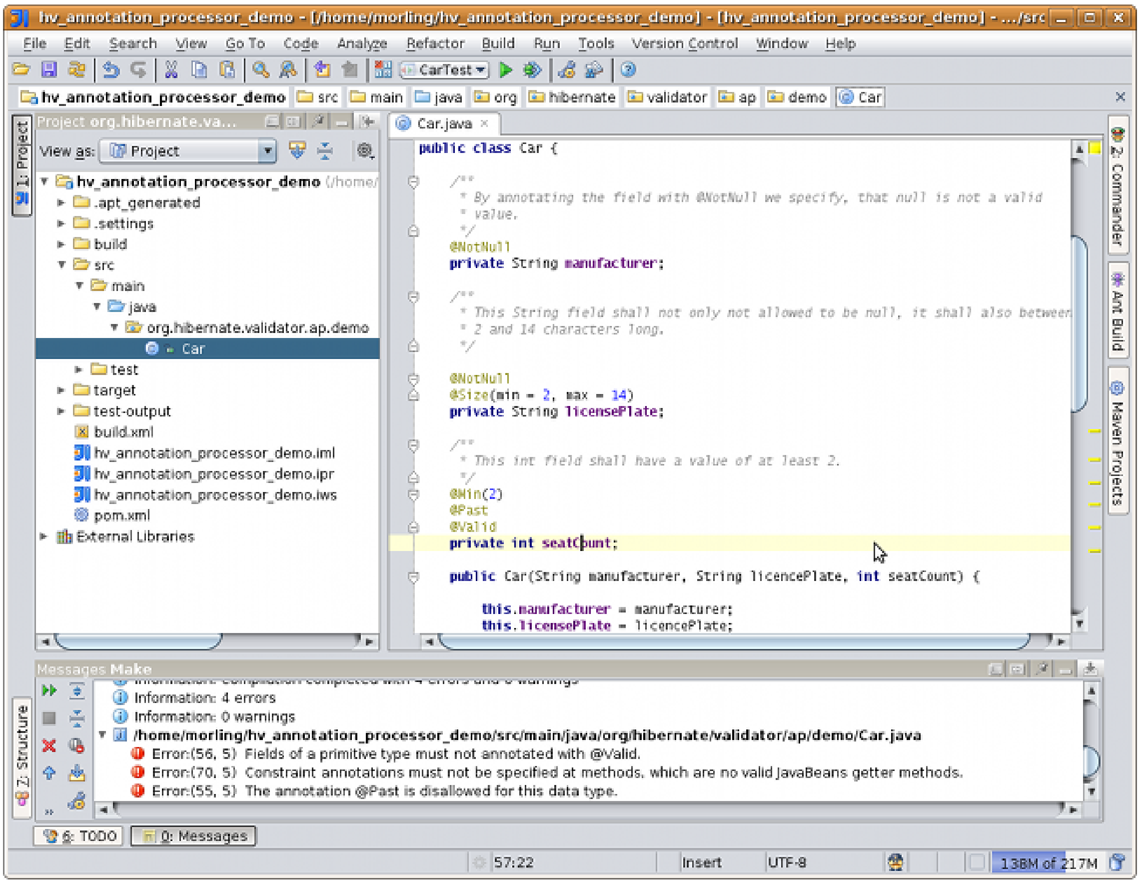


12.4.2.2. IntelliJ IDEA

The following steps must be followed to use the annotation processor within *IntelliJ IDEA* [http://www.jetbrains.com/idea/] (version 9 and above):

- Go to "File", then "Settings",
- Expand the node "Compiler", then "Annotation Processors"
- Choose "Enable annotation processing" and enter the following as "Processor path": /path/to/hibernate-validator-annotation-processor-5.1.3.Final.jar
- Add the processor's fully qualified name `org.hibernate.validator.ap.ConstraintValidationProcessor` to the "Annotation Processors" list
- If applicable add you module to the "Processed Modules" list

Rebuilding your project then should show any erronous constraint annotations:

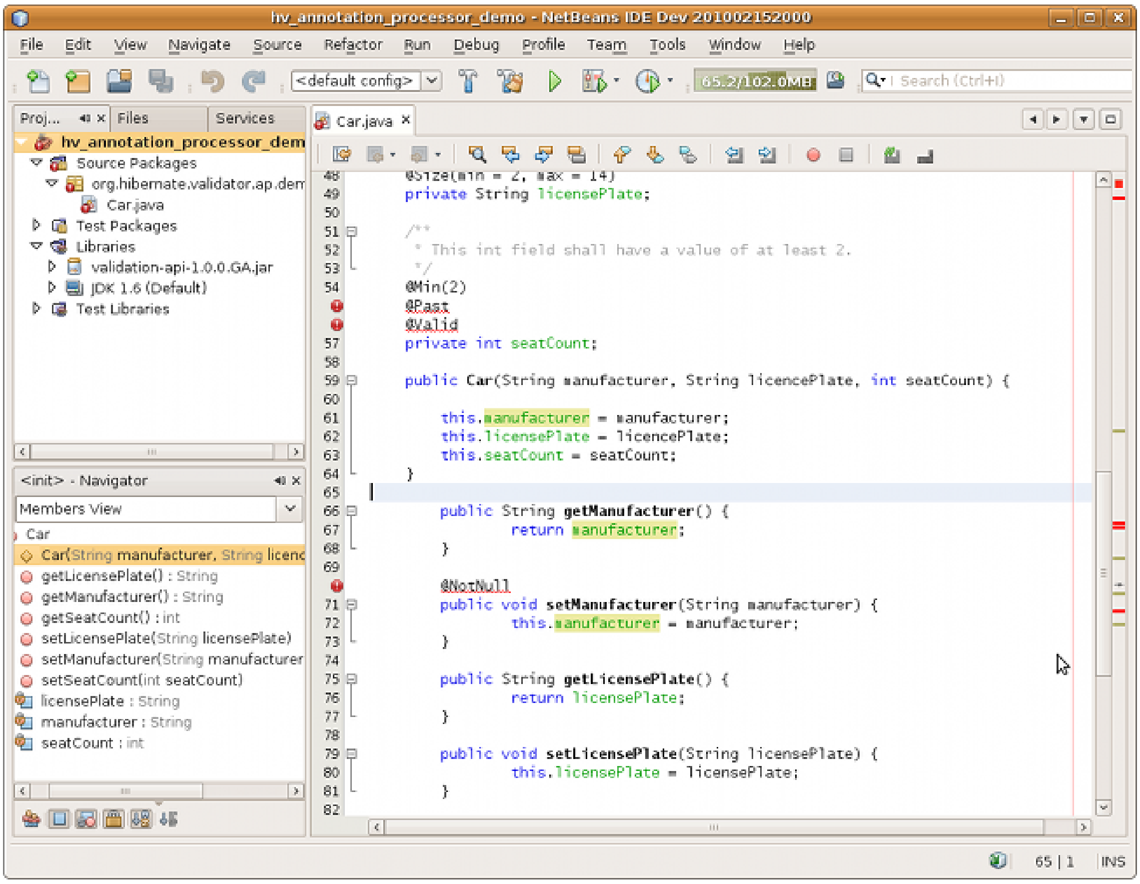


12.4.2.3. NetBeans

Starting with version 6.9, also the *NetBeans* [http://www.netbeans.org/] IDE supports using annotation processors within the IDE build. To do so, do the following:

- Right-click your project, choose "Properties"
- Go to "Libraries", tab "Processor", and add the JAR hibernate-validator-annotation-processor-5.1.3.Final.jar
- Go to "Build - Compiling", select "Enable Annotation Processing" and "Enable Annotation Processing in Editor". Add the annotation processor by specifying its fully qualified name `org.hibernate.validator.ap.ConstraintValidationProcessor`

Any constraint annotation problems will then be marked directly within the editor:



12.5. Known issues

The following known issues exist as of May 2010:

- *HV-308* [http://opensource.atlassian.com/projects/hibernate/browse/HV-308]: Additional validators registered for a constraint *using XML* [http://docs.jboss.org/hibernate/stable/validator/reference/en/html_single/#d0e1957] are not evaluated by the annotation processor.
- Sometimes custom constraints can't be *properly evaluated* [http://opensource.atlassian.com/projects/hibernate/browse/HV-293] when using the processor within Eclipse. Cleaning the project can help in these situations. This seems to be an issue with the Eclipse JSR 269 API implementation, but further investigation is required here.
- When using the processor within Eclipse, the check of dynamic default group sequence definitions doesn't work. After further investigation, it seems to be an issue with the Eclipse JSR 269 API implementation.

Further reading

Last but not least, a few pointers to further information.

A great source for examples is the Bean Validation TCK which is available for anonymous access on *GitHub* [https://github.com/beanvalidation/beanvalidation-tck]. In particular the TCK's *tests* [https://github.com/beanvalidation/beanvalidation-tck/tree/1.1.0.Final/tests] might be of interest. *The JSR 349* [http://beanvalidation.org/1.1/spec/1.1.0.cr3] specification itself is also a great way to deepen your understanding of Bean Validation resp. Hibernate Validator.

If you have any further questions to Hibernate Validator or want to share some of your use cases have a look at the *Hibernate Validator Wiki* [http://community.jboss.org/en/hibernate/validator] and the *Hibernate Validator Forum* [https://forum.hibernate.org/viewforum.php?f=9].

In case you would like to report a bug use *Hibernate's Jira* [http://opensource.atlassian.com/projects/hibernate/browse/HV] instance. Feedback is always welcome!