HIBERNATE

Hibernate Validator 7.0.5.Final - Jakarta Bean Validation Reference Implementation

Reference Guide

Hardy Ferentschik, Gunnar Morling, Guillaume Smet

2022-08-03

Table of Contents

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.

Jakarta Bean Validation 3.0 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 Jakarta Bean Validation. The implementation itself as well as the Jakarta Bean Validation API and TCK are all provided and distributed under the Apache Software License 2.0.

Hibernate Validator 7 and Jakarta Bean Validation 3.0 require Java 8 or later.

Chapter 1. Getting started

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

- A JDK 8

- Apache Maven

- 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.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>7.0.5.Final</version>
</dependency>
```

This transitively pulls in the dependency to the Jakarta Bean Validation API (`jakarta.validation:jakarta.validation-api:3.0.0`).

1.1.1. Unified EL

Hibernate Validator requires an implementation of Jakarta Expression Language for evaluating dynamic expressions in constraint violation messages (see Section 4.1, "Default message interpolation"). When your application runs in a Jakarta EE container such as WildFly/JBoss EAP, 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 dependency to use the Jakarta EL reference implementation:

Example 1.2: Maven dependencies for Unified EL reference implementation

```
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>jakarta.el</artifactId>
    <version>4.0.1</version>
</dependency>
```

For environments where one cannot provide a EL implementation Hibernate Validator is offering a Section 12.10, "ParameterMessageInterpolator". However, the use of this interpolator is not Jakarta Bean Validation specification compliant.

1.1.2. CDI

Jakarta Bean Validation defines integration points with CDI (Contexts and Dependency Injection for Jakarta EE). 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.validator</groupId>
    <artifactId>hibernate-validator-cdi</artifactId>
    <version>7.0.5.Final</version>
</dependency>
```

Note that adding this dependency is usually not required for applications running on a Jakarta EE application server. You can learn more about the integration of Jakarta Bean Validation and CDI in Section 11.3, "CDI".

1.1.3. Running with a security manager

Hibernate Validator supports running with a security manager being enabled. To do so, you must assign several permissions to the code bases of Hibernate Validator, the Jakarta Bean Validation API, Classmate and JBoss Logging and also to the code base calling Jakarta Bean Validation. The following shows how to do this via a policy file as processed by the Java default policy implementation:

Example 1.4: Policy file for using Hibernate Validator with a security manager

```
grant codeBase "file:path/to/hibernate-validator-7.0.5.Final.jar" {
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
    permission java.lang.RuntimePermission "accessDeclaredMembers";
    permission java.lang.RuntimePermission "setContextClassLoader";

    permission org.hibernate.validator.HibernateValidatorPermission "accessPrivateMembers";

    // Only needed when working with XML descriptors (validation.xml or XML constraint
mappings)
    permission java.util.PropertyPermission "mapAnyUriToUri", "read";
};

grant codeBase "file:path/to/jakarta.validation-api-3.0.0.jar" {
    permission java.io.FilePermission "path/to/hibernate-validator-7.0.5.Final.jar",
"read";
};

grant codeBase "file:path/to/jboss-logging-3.4.1.Final.jar" {
    permission java.util.PropertyPermission "org.jboss.logging.provider", "read";
    permission java.util.PropertyPermission "org.jboss.logging.locale", "read";
};

grant codeBase "file:path/to/classmate-1.5.1.jar" {
    permission java.lang.RuntimePermission "accessDeclaredMembers";
};

grant codeBase "file:path/to/validation-caller-x.y.z.jar" {
    permission org.hibernate.validator.HibernateValidatorPermission "accessPrivateMembers";
};
```

1.1.4. Updating Hibernate Validator in WildFly

The WildFly application server contains Hibernate Validator out of the box. In order to update the server modules for Jakarta Bean Validation API and Hibernate Validator to the latest and greatest, the patch mechanism of WildFly can be used.

You can download the patch file from SourceForge or from Maven Central using the following dependency:

Example 1.5: Maven dependency for WildFly 26.0.1.Final patch file

```xml
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-modules</artifactId>
    <version>7.0.5.Final</version>
    <classifier>wildfly-26.0.1.Final-patch</classifier>
    <type>zip</type>
</dependency>
```

We also provide a patch for WildFly :

Example 1.6: Maven dependency for WildFly patch file

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-modules</artifactId>
    <version>7.0.5.Final</version>
    <classifier>wildfly--patch</classifier>
    <type>zip</type>
</dependency>
```

Having downloaded the patch file, you can apply it to WildFly by running this command:

Example 1.7: Applying the WildFly patch

```
$JBOSS_HOME/bin/jboss-cli.sh patch apply hibernate-validator-modules-7.0.5.Final-wildfly-
26.0.1.Final-patch.zip
```

In case you want to undo the patch and go back to the version of Hibernate Validator originally coming with the server, run the following command:

Example 1.8: Rolling back the WildFly patch

```
$JBOSS_HOME/bin/jboss-cli.sh patch rollback --reset-configuration=true
```

You can learn more about the WildFly patching infrastructure in general here and here.

1.1.5. Running on Java 11+

Support for Java 11 and the Java Platform Module System (JPMS) is present in a preliminary form. There are no JPMS module descriptors provided, but Hibernate Validator is usable as automatic modules.

These are the module names as declared using the `Automatic-Module-Name` header:

- Jakarta Bean Validation API: `jakarta.validation`

- Hibernate Validator core: `org.hibernate.validator`

- Hibernate Validator CDI extension: `org.hibernate.validator.cdi`

- Hibernate Validator test utilities: `org.hibernate.validator.testutils`

- Hibernate Validator annotation processor: `org.hibernate.validator.annotationprocessor`

These module names are preliminary and may be changed when providing real module descriptors in a future release.

1.2. Applying constraints

Let's dive directly into an example to see how to apply constraints.

Example 1.9: Class Car annotated with constraints

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

import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotNull;
import jakarta.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

> You can find the complete source code of all examples used in this reference guide in the Hibernate Validator source repository 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 1.10: Class CarTest showing validation examples

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

```java
import java.util.Set;
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.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 setUpValidator() {
        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( "must 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 `jakarta.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 minutes tour through the world of Hibernate Validator and Jakarta Bean Validation. Continue exploring the code examples or look at further examples referenced in Chapter 14, *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 Jakarta 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*.

Chapter 2. 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 Jakarta Bean Validation are expressed via Java annotations. In this section you will learn how to enhance an object model with these annotations. There are four types of bean constraints:

- field constraints

- property constraints

- container element constraints

- class constraints

> Not all constraints can be placed on all of these levels. In fact, none of the default constraints defined by Jakarta 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.

> 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 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;
    }
}
```

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.

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. Container element constraints

It is possible to specify constraints directly on the type argument of a parameterized type: these constraints are called container element constraints.

This requires that `ElementType.TYPE_USE` is specified via `@Target` in the constraint definition. As of Jakarta Bean Validation 2.0, built-in Jakarta Bean Validation as well as Hibernate Validator specific constraints specify `ElementType.TYPE_USE` and can be used directly in this context.

Hibernate Validator validates container element constraints specified on the following standard Java containers:

- implementations of `java.util.Iterable` (e.g. `List`s, `Set`s),

- implementations of `java.util.Map`, with support for keys and values,

- `java.util.Optional`, `java.util.OptionalInt`, `java.util.OptionalDouble`, `java.util.OptionalLong`,

- the various implementations of JavaFX's `javafx.beans.observable.ObservableValue`.

It also supports container element constraints on custom container types (see Chapter 7, *Value extraction*).

> In versions prior to 6, a subset of container element constraints were supported. A `@Valid` annotation was required at the container level to enable them. This is not required anymore as of Hibernate Validator 6.

We present below a couple of examples illustrating container element constraints on various Java types.

In these examples, `@ValidPart` is a custom constraint allowed to be used in the `TYPE_USE` context.

2.1.3.1. With `Iterable`

When applying constraints on an `Iterable` type argument, Hibernate Validator will validate each element. Example 2.3, "Container element constraint on `Set`" shows an example of a `Set` with a container element constraint.

Example 2.3: Container element constraint on Set

```java
package org.hibernate.validator.referenceguide.chapter02.containerelement.set;

public class Car {

    private Set<@ValidPart String> parts = new HashSet<>();

    public void addPart(String part) {
        parts.add( part );
    }

    //...

}
```

```java
Car car = new Car();
car.addPart( "Wheel" );
car.addPart( null );

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

assertEquals( 1, constraintViolations.size() );

ConstraintViolation<Car> constraintViolation =
        constraintViolations.iterator().next();
assertEquals(
        "'null' is not a valid car part.",
        constraintViolation.getMessage()
);
assertEquals( "parts[].<iterable element>",
        constraintViolation.getPropertyPath().toString() );
```

Note how the property path clearly states that the violation comes from an element of the iterable.

2.1.3.2. With List

When applying constraints on a List type argument, Hibernate Validator will validate each element. Example 2.4, "Container element constraint on List" shows an example of a List with a container element constraint.

Example 2.4: Container element constraint on `List`

```java
package org.hibernate.validator.referenceguide.chapter02.containerelement.list;

public class Car {

    private List<@ValidPart String> parts = new ArrayList<>();

    public void addPart(String part) {
        parts.add( part );
    }

    //...

}
```

```java
Car car = new Car();
car.addPart( "Wheel" );
car.addPart( null );

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

assertEquals( 1, constraintViolations.size() );

ConstraintViolation<Car> constraintViolation =
        constraintViolations.iterator().next();
assertEquals(
        "'null' is not a valid car part.",
        constraintViolation.getMessage()
);
assertEquals( "parts[1].<list element>",
        constraintViolation.getPropertyPath().toString() );
```

Here, the property path also contains the index of the invalid element.

2.1.3.3. With `Map`

Container element constraints are also validated on map keys and values. Example 2.5, "Container element constraint on map keys and values" shows an example of a `Map` with a constraint on the key and a constraint on the value.

Example 2.5: Container element constraint on map keys and values

```java
package org.hibernate.validator.referenceguide.chapter02.containerelement.map;

public class Car {

    public enum FuelConsumption {
        CITY,
        HIGHWAY
    }

    private Map<@NotNull FuelConsumption, @MaxAllowedFuelConsumption Integer>
fuelConsumption = new HashMap<>();

    public void setFuelConsumption(FuelConsumption consumption, int value) {
        fuelConsumption.put( consumption, value );
    }

    //...

}
```

```java
Car car = new Car();
car.setFuelConsumption( Car.FuelConsumption.HIGHWAY, 20 );

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

assertEquals( 1, constraintViolations.size() );

ConstraintViolation<Car> constraintViolation =
        constraintViolations.iterator().next();
assertEquals(
        "20 is outside the max fuel consumption.",
        constraintViolation.getMessage()
);
assertEquals(
        "fuelConsumption[HIGHWAY].<map value>",
        constraintViolation.getPropertyPath().toString()
);
```

```java
Car car = new Car();
car.setFuelConsumption( null, 5 );

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

assertEquals( 1, constraintViolations.size() );

ConstraintViolation<Car> constraintViolation =
        constraintViolations.iterator().next();
assertEquals(
        "must not be null",
        constraintViolation.getMessage()
);
assertEquals(
        "fuelConsumption<K>[].<map key>",
        constraintViolation.getPropertyPath().toString()
);
```

The property paths of the violations are particularly interesting:

- The key of the invalid element is included in the property path (in the second example, the key is `null`).

- In the first example, the violation concerns the `<map value>`, in the second one, the `<map key>`.

- In the second example, you might have noticed the presence of the type argument `<K>`, more on this later.

2.1.3.4. With `java.util.Optional`

When applying a constraint on the type argument of `Optional`, Hibernate Validator will automatically unwrap the type and validate the internal value. Example 2.6, "Container element constraint on Optional" shows an example of an `Optional` with a container element constraint.

Example 2.6: Container element constraint on Optional

```
package org.hibernate.validator.referenceguide.chapter02.containerelement.optional;

public class Car {

    private Optional<@MinTowingCapacity(1000) Integer> towingCapacity = Optional.empty();

    public void setTowingCapacity(Integer alias) {
        towingCapacity = Optional.of( alias );
    }

    //...

}
```

```
Car car = new Car();
car.setTowingCapacity( 100 );

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

assertEquals( 1, constraintViolations.size() );

ConstraintViolation<Car> constraintViolation = constraintViolations.iterator().next();
assertEquals(
        "Not enough towing capacity.",
        constraintViolation.getMessage()
);
assertEquals(
        "towingCapacity",
        constraintViolation.getPropertyPath().toString()
);
```

Here, the property path only contains the name of the property as we are considering `Optional` as a "transparent" container.

2.1.3.5. With custom container types

Container element constraints can also be used with custom containers.

A `ValueExtractor` must be registered for the custom type allowing to retrieve the value(s) to validate (see Chapter 7, *Value extraction* for more information about how to implement your own `ValueExtractor` and how to register it).

Example 2.7, "Container element constraint on custom container type" shows an example of a custom parameterized type with a type argument constraint.

Example 2.7: Container element constraint on custom container type

```java
package org.hibernate.validator.referenceguide.chapter02.containerelement.custom;

public class Car {

    private GearBox<@MinTorque(100) Gear> gearBox;

    public void setGearBox(GearBox<Gear> gearBox) {
        this.gearBox = gearBox;
    }

    //...

}
```

```java
package org.hibernate.validator.referenceguide.chapter02.containerelement.custom;

public class GearBox<T extends Gear> {

    private final T gear;

    public GearBox(T gear) {
        this.gear = gear;
    }

    public Gear getGear() {
        return this.gear;
    }
}
```

```java
package org.hibernate.validator.referenceguide.chapter02.containerelement.custom;

public class Gear {
    private final Integer torque;

    public Gear(Integer torque) {
        this.torque = torque;
    }

    public Integer getTorque() {
        return torque;
    }

    public static class AcmeGear extends Gear {
        public AcmeGear() {
            super( 60 );
        }
    }
}
```

```
package org.hibernate.validator.referenceguide.chapter02.containerelement.custom;

public class GearBoxValueExtractor implements ValueExtractor<GearBox<@ExtractedValue ?>> {

    @Override
    public void extractValues(GearBox<@ExtractedValue ?> originalValue, ValueExtractor
.ValueReceiver receiver) {
        receiver.value( null, originalValue.getGear() );
    }
}
```

```
Car car = new Car();
car.setGearBox( new GearBox<>( new Gear.AcmeGear() ) );

Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );
assertEquals( 1, constraintViolations.size() );

ConstraintViolation<Car> constraintViolation =
        constraintViolations.iterator().next();
assertEquals(
        "Gear is not providing enough torque.",
        constraintViolation.getMessage()
);
assertEquals(
        "gearBox",
        constraintViolation.getPropertyPath().toString()
);
```

2.1.3.6. Nested container elements

Constraints are also supported on nested container elements.

When validating a `Car` object as presented in Example 2.8, "Constraints on nested container elements", both the `@NotNull` constraints on `Part` and `Manufacturer` will be enforced.

Example 2.8: Constraints on nested container elements

```
package org.hibernate.validator.referenceguide.chapter02.containerelement.nested;

public class Car {

    private Map<@NotNull Part, List<@NotNull Manufacturer>> partManufacturers =
            new HashMap<>();

    //...
}
```

2.1.4. 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 2.9, "Class-level constraint" has the two attributes `seatCount` and `passengers` and it should be ensured that the list of passengers does not have more entries than available seats. 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 2.9: Class-level constraint

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

@ValidPassengerCount
public class Car {

    private int seatCount;

    private List<Person> passengers;

    //...
}
```

2.1.5. Constraint inheritance

When a class implements an interface or extends another class, all constraint annotations declared on the super-type 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 2.10: Constraint inheritance

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

public class Car {

    private String manufacturer;

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

    //...
}
```

```java
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.

2.1.6. Object graphs

The Jakarta 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 2.11, "Cascaded validation".

Example 2.11: 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.

As constraints, object graph validation also works for container elements. That means any type argument of a container can be annotated with `@Valid`, which will cause each contained element to be validated when the parent object is validated.

> **ℹ** Cascaded validation is also supported for nested container elements.

Example 2.12: Cascaded validation of containers

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

public class Car {

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

    private Map<@Valid Part, List<@Valid Manufacturer>> partManufacturers = new HashMap<>(
);

    //...
}
```

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

public class Part {

    @NotNull
    private String name;

    //...
}
```

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

public class Manufacturer {

    @NotNull
    private String name;

    //...
}
```

When validating an instance of the `Car` class shown in Example 2.12, "Cascaded validation of containers", a `ConstraintViolation` will be created:

- if any of the `Person` objects contained in the passengers list has a `null` name;

- if any of the `Part` objects contained in the map keys has a `null` name;

- if any of the `Manufacturer` objects contained in the list nested in the map values has a `null` name.

> In versions prior to 6, Hibernate Validator supported cascaded validation for a subset of container elements and it was implemented at the container level (e.g. you would use `@Valid private List<Person>` to enable cascaded validation for `Person`).
>
> This is still supported but is not recommended. Please use container element level `@Valid` annotations instead as it is more expressive.

2.2. Validating bean constraints

The `Validator` interface is the most important object in Jakarta Bean Validation. The next section shows how to obtain a `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 2.13: `Validation#buildDefaultValidatorFactory()`

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

This bootstraps a validator in the default configuration. Refer to Chapter 9, *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 (`jakarta.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 2.14, "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 2.14: Using `Validator#validate()`

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

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

assertEquals( 1, constraintViolations.size() );
assertEquals( "must 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 2.15: Using `Validator#validateProperty()`

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

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

assertEquals( 1, constraintViolations.size() );
assertEquals( "must 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 2.16: Using `Validator#validateValue()`

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

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

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

`Validator#validateProperty()` is for example used in the integration of Jakarta Bean Validation into JSF 2 (see Section 11.2, "JSF & Seam") to perform a validation of the values entered into a form before they are propagated to the model.

2.2.3. `ConstraintViolation`

2.2.3.1. `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. The following gives an overview of these methods. The values under "Example" column refer to Example 2.14, "Using `Validator#validate()`".

`getMessage()`

    The interpolated error message

    Example

        "must not be null"

`getMessageTemplate()`

    The non-interpolated error message

    Example

        "{… NotNull.message}"

`getRootBean()`

    The root bean being validated

    Example

        car

`getRootBeanClass()`

    The class of the root bean being validated

    Example

        `Car.class`

`getLeafBean()`

    If a bean constraint, the bean instance the constraint is applied on; if a property constraint, the bean instance hosting the property the constraint is applied on

    Example

        `car`

`getPropertyPath()`

    The property path to the validated value from root bean

Example

> contains one node with kind `PROPERTY` and name "manufacturer"

`getInvalidValue()`

> The value failing to pass the constraint

Example

> `null`

`getConstraintDescriptor()`

> Constraint metadata reported to fail

Example

> descriptor for `@NotNull`

2.2.3.2. Exploiting the property path

To determine the element that triggered the violation, you need to exploit the result of the `getPropertyPath()` method.

The returned `Path` is composed of `Node`s describing the path to the element.

More information about the structure of the `Path` and the various types of `Node`s can be found in the `ConstraintViolation` section of the Jakarta Bean Validation specification.

2.3. Built-in constraints

Hibernate Validator comprises a basic set of commonly used constraints. These are foremost the constraints defined by the Jakarta Bean Validation specification (see Section 2.3.1, "Jakarta Bean Validation constraints"). Additionally, Hibernate Validator provides useful custom constraints (see Section 2.3.2, "Additional constraints").

2.3.1. Jakarta Bean Validation constraints

Below you can find a list of all constraints specified in the Jakarta Bean Validation API. All these constraints apply to the field/property level, there are no class-level constraints defined in the Jakarta 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 "Hibernate metadata impact").

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

`@AssertFalse`

Checks that the annotated element is false

Supported data types

> `Boolean`, `boolean`

Hibernate metadata impact

> None

`@AssertTrue`

Checks that the annotated element is true

Supported data types

> `Boolean`, `boolean`

Hibernate metadata impact

> None

`@DecimalMax(value=, inclusive=)`

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.

Supported data types

> `BigDecimal`, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types; additionally supported by HV: any sub-type of `Number` and `javax.money.MonetaryAmount` (if the JSR 354 API and an implementation is on the class path)

Hibernate metadata impact

> None

`@DecimalMin(value=, inclusive=)`

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.

Supported data types

> `BigDecimal`, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types; additionally supported by HV: any sub-type of `Number` and `javax.money.MonetaryAmount`

Hibernate metadata impact

> None

`@Digits(integer=, fraction=)`

Checks whether the annotated value is a number having up to `integer` digits and `fraction` fractional digits

Supported data types

> BigDecimal, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types; additionally supported by HV: any sub-type of `Number` and `javax.money.MonetaryAmount`

Hibernate metadata impact

> Defines column precision and scale

`@Email`

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.

Supported data types

> `CharSequence`

Hibernate metadata impact

> None

`@Future`

Checks whether the annotated date is in the future

Supported data types

> `java.util.Date`, `java.util.Calendar`, `java.time.Instant`, `java.time.LocalDate`, `java.time.LocalDateTime`, `java.time.LocalTime`, `java.time.MonthDay`, `java.time.OffsetDateTime`, `java.time.OffsetTime`, `java.time.Year`, `java.time.YearMonth`, `java.time.ZonedDateTime`, `java.time.chrono.HijrahDate`, `java.time.chrono.JapaneseDate`, `java.time.chrono.MinguoDate`, `java.time.chrono.ThaiBuddhistDate`; additionally supported by HV, if the Joda Time date/time API is on the classpath: any implementations of `ReadablePartial` and `ReadableInstant`

Hibernate metadata impact

None

@FutureOrPresent

Checks whether the annotated date is in the present or in the future

Supported data types

`java.util.Date`, `java.util.Calendar`, `java.time.Instant`, `java.time.LocalDate`, `java.time.LocalDateTime`, `java.time.LocalTime`, `java.time.MonthDay`, `java.time.OffsetDateTime`, `java.time.OffsetTime`, `java.time.Year`, `java.time.YearMonth`, `java.time.ZonedDateTime`, `java.time.chrono.HijrahDate`, `java.time.chrono.JapaneseDate`, `java.time.chrono.MinguoDate`, `java.time.chrono.ThaiBuddhistDate`; additionally supported by HV, if the Joda Time date/time API is on the classpath: any implementations of `ReadablePartial` and `ReadableInstant`

Hibernate metadata impact

None

@Max(value=)

Checks whether the annotated value is less than or equal to the specified maximum

Supported data types

`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` and `javax.money.MonetaryAmount`

Hibernate metadata impact

Adds a check constraint on the column

@Min(value=)

Checks whether the annotated value is higher than or equal to the specified minimum

Supported data types

`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` and `javax.money.MonetaryAmount`

Hibernate metadata impact

Adds a check constraint on the column

@NotBlank

Checks that the annotated character sequence is not null and the trimmed length is greater than 0. The difference to `@NotEmpty` is that this constraint can only be applied on character sequences and that trailing white-spaces are ignored.

Supported data types

> `CharSequence`

Hibernate metadata impact

> None

@NotEmpty

Checks whether the annotated element is not null nor empty

Supported data types

> `CharSequence`, `Collection`, `Map` and arrays

Hibernate metadata impact

> None

@NotNull

Checks that the annotated value is not `null`

Supported data types

> Any type

Hibernate metadata impact

> Column(s) are not nullable

@Negative

Checks if the element is strictly negative. Zero values are considered invalid.

Supported data types

> `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` and `javax.money.MonetaryAmount`

Hibernate metadata impact

> None

@NegativeOrZero

Checks if the element is negative or zero.

Supported data types

> `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` and `javax.money.MonetaryAmount`

Hibernate metadata impact

> None

@Null

Checks that the annotated value is `null`

Supported data types

> Any type

Hibernate metadata impact

> None

@Past

Checks whether the annotated date is in the past

Supported data types

> `java.util.Date`,`java.util.Calendar`, `java.time.Instant`, `java.time.LocalDate`, `java.time.LocalDateTime`, `java.time.LocalTime`, `java.time.MonthDay`, `java.time.OffsetDateTime`, `java.time.OffsetTime`, `java.time.Year`, `java.time.YearMonth`, `java.time.ZonedDateTime`, `java.time.chrono.HijrahDate`, `java.time.chrono.JapaneseDate`, `java.time.chrono.MinguoDate`, `java.time.chrono.ThaiBuddhistDate`; Additionally supported by HV, if the Joda Time date/time API is on the classpath: any implementations of `ReadablePartial` and `ReadableInstant`

Hibernate metadata impact

> None

@PastOrPresent

Checks whether the annotated date is in the past or in the present

Supported data types

> `java.util.Date`,`java.util.Calendar`, `java.time.Instant`, `java.time.LocalDate`, `java.time.LocalDateTime`, `java.time.LocalTime`, `java.time.MonthDay`, `java.time.OffsetDateTime`, `java.time.OffsetTime`, `java.time.Year`, `java.time.YearMonth`, `java.time.ZonedDateTime`, `java.time.chrono.HijrahDate`, `java.time.chrono.JapaneseDate`, `java.time.chrono.MinguoDate`,

`java.time.chrono.ThaiBuddhistDate`; Additionally supported by HV, if the Joda Time date/time API is on the classpath: any implementations of `ReadablePartial` and `ReadableInstant`

Hibernate metadata impact

None

@Pattern(regex=, flags=)

Checks if the annotated string matches the regular expression `regex` considering the given flag `match`

Supported data types

`CharSequence`

Hibernate metadata impact

None

@Positive

Checks if the element is strictly positive. Zero values are considered invalid.

Supported data types

`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` and `javax.money.MonetaryAmount`

Hibernate metadata impact

None

@PositiveOrZero

Checks if the element is positive or zero.

Supported data types

`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` and `javax.money.MonetaryAmount`

Hibernate metadata impact

None

@Size(min=, max=)

Checks if the annotated element's size is between `min` and `max` (inclusive)

Supported data types

 CharSequence, Collection, Map and arrays

Hibernate metadata impact

 Column length will be set to max

> ℹ️ On top of the parameters listed above each constraint has the parameters message, groups and payload. This is a requirement of the Jakarta Bean Validation specification.

2.3.2. Additional constraints

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

@CreditCardNumber(ignoreNonDigitCharacters=)

 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 a credit card number. ignoreNonDigitCharacters allows to ignore non digit characters. The default is false.

Supported data types

 CharSequence

Hibernate metadata impact

 None

@Currency(value=)

 Checks that the currency unit of the annotated javax.money.MonetaryAmount is part of the specified currency units.

Supported data types

 any sub-type of javax.money.MonetaryAmount (if the JSR 354 API and an implementation is on the class path)

Hibernate metadata impact

 None

@DurationMax(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=)

 Checks that annotated java.time.Duration element is not greater than the one constructed from annotation parameters. Equality is allowed if inclusive flag is set to true.

Supported data types

    `java.time.Duration`

Hibernate metadata impact

    None

`@DurationMin(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=)`

Checks that annotated `java.time.Duration` element is not less than the one constructed from annotation parameters. Equality is allowed if `inclusive` flag is set to `true`.

Supported data types

    `java.time.Duration`

Hibernate metadata impact

    None

`@EAN`

Checks that the annotated character sequence is a valid EAN barcode. type determines the type of barcode. The default is EAN-13.

Supported data types

    `CharSequence`

Hibernate metadata impact

    None

`@ISBN`

Checks that the annotated character sequence is a valid ISBN. `type` determines the type of ISBN. The default is ISBN-13.

Supported data types

    `CharSequence`

Hibernate metadata impact

    None

`@Length(min=, max=)`

Validates that the annotated character sequence is between `min` and `max` included

Supported data types

    `CharSequence`

Hibernate metadata impact

Column length will be set to max

@CodePointLength(min=, max=, normalizationStrategy=)

Validates that code point length of the annotated character sequence is between `min` and `max` included. Validates normalized value if `normalizationStrategy` is set.

Supported data types

`CharSequence`

Hibernate metadata impact

None

@LuhnCheck(startIndex= , endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=)

Checks that the digits within the annotated character sequence pass the Luhn checksum algorithm (see also Luhn algorithm). `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.

Supported data types

`CharSequence`

Hibernate metadata impact

None

@Mod10Check(multiplier=, weight=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=)

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 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.

Supported data types

`CharSequence`

Hibernate metadata impact

None

`@Mod11Check(threshold=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=, treatCheck10As=, treatCheck11As=)`

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 checkDigitIndex` and `ignoreNonDigitCharacters` carry the same semantics as in `@Mod10Check`.

Supported data types

> `CharSequence`

Hibernate metadata impact

> None

`@Normalized(form=)`

Validates that the annotated character sequence is normalized according to the given `form`.

Supported data types

> `CharSequence`

Hibernate metadata impact

> None

`@Range(min=, max=)`

Checks whether the annotated value lies between (inclusive) the specified minimum and maximum

Supported data types

> `BigDecimal`, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types

Hibernate metadata impact

> None

`@ScriptAssert(lang=, script=, alias=, reportOn=)`

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™ Platform") must be a 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. Even though this is a class-level constraint, one can use the `reportOn` attribute to report a constraint violation on a specific property rather than the whole object.

Supported data types

> Any type

Hibernate metadata impact

> None

@UniqueElements

Checks that the annotated collection only contains unique elements. The equality is determined using the `equals()` method. The default message does not include the list of duplicate elements but you can include it by overriding the message and using the `{duplicates}` message parameter. The list of duplicate elements is also included in the dynamic payload of the constraint violation.

Supported data types

> `Collection`

Hibernate metadata impact

> None

@URL(protocol=, host=, port=, regexp=, flags=)

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 parameters `regexp` and `flags` allow to specify an additional regular expression (including regular expression flags) which the URL must match. Per default this constraint used the `java.net.URL` constructor to verify whether a given string represents a valid URL. A regular expression based version is also available - `RegexpURLValidator` - which can be configured via XML (see Section 8.2, "Mapping constraints via `constraint-mappings`") or the programmatic API (see Section 12.15.2, "Adding constraint definitions programmatically").

Supported data types

> `CharSequence`

Hibernate metadata impact

> None

2.3.2.1. Country specific constraints

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

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

@CNPJ

Checks that the annotated character sequence represents a Brazilian corporate tax payer registry number (Cadastro de Pessoa Jurídica)

Supported data types

> CharSequence

Hibernate metadata impact

> None

Country

> Brazil

@CPF

Checks that the annotated character sequence represents a Brazilian individual taxpayer registry number (Cadastro de Pessoa Física)

Supported data types

> CharSequence

Hibernate metadata impact

> None

Country

> Brazil

@TituloEleitoral

Checks that the annotated character sequence represents a Brazilian voter ID card number (Título Eleitoral)

Supported data types

> CharSequence

Hibernate metadata impact

> None

Country

> Brazil

@NIP

Checks that the annotated character sequence represents a Polish VAT identification number (NIP)

Supported data types

    `CharSequence`

Hibernate metadata impact

    None

Country

    Poland

@PESEL

Checks that the annotated character sequence represents a Polish national identification number (PESEL)

Supported data types

    `CharSequence`

Hibernate metadata impact

    None

Country

    Poland

@REGON

Checks that the annotated character sequence represents a Polish taxpayer identification number (REGON). Can be applied to both 9 and 14 digits versions of REGON

Supported data types

    `CharSequence`

Hibernate metadata impact

    None

Country

    Poland

@INN

Checks that the annotated character sequence represents a Russian taxpayer identification number (INN). Can be applied to both individual and juridical versions of INN

Supported data types

    `CharSequence`

Hibernate metadata impact

None

Country

Russia

> In some cases neither the Jakarta 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*.

Chapter 3. 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 Jakarta 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)

> For the purpose of this reference guide, the term *method constraint* refers to both, method and constructor constraints, if not stated otherwise. Occasionally, the term *executable* is used when referring 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 `IllegalArgumentException` 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

> In order to make annotations show up in the JavaDoc of annotated 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` as well as be in the future and finally 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) {
        //...
        return null;
    }

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

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

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

public class RentalStation {

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

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

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

- The customer list returned by `getCustomers()` must no contain `null` objects

> As you can see in the above example, container element constraints are supported on method return value. They are also supported on method parameters.

3.1.3. Cascaded validation

Similar to the cascaded validation of JavaBeans properties (see Section 2.1.6, "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

```java
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) {
        //...
        return false;
    }
}
```

```java
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.

Same as for fields and properties, cascaded validation can also be declared on container elements (e.g. elements of collections, maps or custom containers) of return values and parameters.

In this case, each element contained by the container gets validated. So when validating the arguments of the `checkCars()` method in Example 3.6, "Container elements of 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` instances is invalid.

Example 3.6: Container elements of method parameter marked for cascaded validation

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

public class Garage {

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

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 satisfied 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.

The Jakarta 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 {

    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

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

public interface Vehicle {

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

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

public class Car implements Vehicle {

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

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

> 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.

> Enforcement of these rules may be relaxed by setting the configuration parameters contained in the `MethodValidationConfiguration` property of the `HibernateValidatorConfiguration` before creating the `Validator` instance. See also Section 12.3, "Relaxation of requirements for method validation in class hierarchies".

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` instance".

Example 3.10: Obtaining an `ExecutableValidator` instance

```
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 9, *Bootstrapping*, for instance in order to use a specific parameter name provider (see Section 9.2.4, "`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() {
        //...
        return Collections.emptyList();
    }
}
```

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 be validated. The validation in Example 3.13, "Using `ExecutableValidator#validateReturnValue()`" yields one constraint violation since the `getPassengers()` method is expected to return at least one `Passenger` instance.

Example 3.13: Using `ExecutableValidator#validateReturnValue()`

```java
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()`

```java
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 validate a constructor's return value. In Example 3.15, "Using `ExecutableValidator#validateConstructorReturnValue()`", `validateConstructorReturnValue()` returns one constraint violation, since the `Car` instance 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`", `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 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

```java
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( "speedInMph", parameterNode.getName() );
assertEquals( 0, parameterNode.getParameterIndex() );
```

The parameter name is determined using the current `ParameterNameProvider` (see Section 9.2.4, "ParameterNameProvider").

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 9.2.4, "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.parameterscriptassert;

public class Car {

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

Chapter 4. Interpolating constraint error messages

Message interpolation is the process of creating error messages for violated Jakarta 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 Jakarta 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.

> You can find the formal definition of the interpolation algorithm in section 6.3.1.1 of the Jakarta 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 Jakarta Expression Language 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)`.

Expression Language is very flexible and Hibernate Validator offers several feature levels that you can

use to enable Expression Language features through the `ExpressionLanguageFeatureLevel` enum:

- `NONE`: Expression Language interpolation is fully disabled.

- `VARIABLES`: Allow interpolation of the variables injected via `addExpressionVariable()`, resources bundles and usage of the `formatter` object.

- `BEAN_PROPERTIES`: Allow everything `VARIABLES` allows plus the interpolation of bean properties.

- `BEAN_METHODS`: Also allow execution of bean methods. Can be considered safe for hardcoded constraint messages but not for custom violations where extra care is required.

The default feature level for constraint messages is `BEAN_PROPERTIES`.

You can define the Expression Language feature level when bootstrapping the `ValidatorFactory`.

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

```java
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 :

- the `@NotNull` constraint on the `manufacturer` field causes the error message "must not be null", as this is the default message defined by the Jakarta 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 to use an EL expression with a ternary expression to dynamically choose 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 instance

- 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

> 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 12.13.1, "HibernateConstraintValidatorContext"), an EL expression in the form `${attributeName}` must be used.

Example 4.3: Expected error messages

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

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

message = validator.validateProperty( car, "licensePlate" )
        .iterator()
        .next()
        .getMessage();
assertEquals(
        "The license plate 'A' 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 `jakarta.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 Jakarta Bean Validation XML descriptor *META-INF/validation.xml* (see Section 8.1, "Configuring the validator factory in *validation.xml*") or by passing it when bootstrapping a `ValidatorFactory` or `Validator` (see Section 9.2.1, "`MessageInterpolator`" and Section 9.3, "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 fall-back 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.

Chapter 5. 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, "Example class `Person`" has a `@NotNull` constraint on `name`. Since no group is specified for this annotation the default group `jakarta.validation.groups.Default` is assumed.

> ℹ️ 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 `jakarta.validation.groups.Default` is assumed.

Example 5.1: Example class `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.

> 💡 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. See Section 5.2, "Group inheritance".

Example 5.2: Driver

```java
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;
    }
}
```

```java
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

```java
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;
    }

    public boolean isPassedVehicleInspection() {
        return passedVehicleInspection;
    }

    public void setPassedVehicleInspection(boolean passedVehicleInspection) {
        this.passedVehicleInspection = passedVehicleInspection;
    }

    public Driver getDriver() {
        return driver;
    }

    public void setDriver(Driver driver) {
        this.driver = driver;
    }

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

```java
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

```java
// 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, CarChecks.class ).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` as 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. Group inheritance

In Example 5.4, "Using validation groups", we need to call `validate()` for each validation group, or specify all of them one by one.

In some situations, you may want to define a group of constraints which includes another group. You can do that using group inheritance.

In Example 5.5, "SuperCar", we define a `SuperCar` and a group `RaceCarChecks` that extends the `Default` group. A `SuperCar` must have safety belts to be allowed to run in races.

Example 5.5: SuperCar

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

public class SuperCar extends Car {

    @AssertTrue(
            message = "Race car must have a safety belt",
            groups = RaceCarChecks.class
    )
    private boolean safetyBelt;

    // getters and setters ...

}
```

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

import jakarta.validation.groups.Default;

public interface RaceCarChecks extends Default {
}
```

In the example below, we will check if a `SuperCar` with one seat and no security belts is a valid car and if it is a valid race-car.

Example 5.6: Using group inheritance

```
// create a supercar and check that it's valid as a generic Car
SuperCar superCar = new SuperCar( "Morris", "DD-AB-123", 1  );
assertEquals( "must be greater than or equal to 2", validator.validate( superCar ).
iterator().next().getMessage() );

// check that this supercar is valid as generic car and also as race car
Set<ConstraintViolation<SuperCar>> constraintViolations = validator.validate( superCar,
RaceCarChecks.class );

assertThat( constraintViolations ).extracting( "message" ).containsOnly(
        "Race car must have a safety belt",
        "must be greater than or equal to 2"
);
```

On the first call to `validate()`, we do not specify a group. There is one validation error because a car must have at least one seat. It is the constraint from the `Default` group.

On the second call, we specify only the group `RaceCarChecks`. There are two validation errors: one about the missing seat from the `Default` group, another one about the fact that there is no safety belts coming from the `RaceCarChecks` group.

5.3. 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 in which 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 5.7, "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 5.7: Defining a group sequence

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

import jakarta.validation.GroupSequence;
import jakarta.validation.groups.Default;

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

> 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 5.8, "Using a group sequence".

Example 5.8: Using a group sequence

```java
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() );
```

5.4. Redefining the default group sequence

5.4.1. @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 5.9, "Class `RentalCar` with redefined default group" introduces a new class `RentalCar` with a redefined default group.

Example 5.9: Class `RentalCar` *with redefined default group*

```java
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;
    }
}
```

```java
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 5.10, "Validating an object with redefined default group".

Example 5.10: Validating an object with redefined default group

```java
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() );
```

Since there must be 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 5.5, "Group conversion").

5.4.2. `@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 5.11, "Implementing and using a default group sequence provider".

Example 5.11: Implementing and using a default group sequence provider

```java
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;
    }
}
```

```java
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;
    }
}
```

5.5. 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 to use a different group than the originally requested one during cascaded validation.

Let's have a look at Example 5.12, "`@ConvertGroup` usage". Here `@GroupSequence({` `CarChecks.class, Car.class })` is used to combine the car related constraints under the `Default` group (see Section 5.4, "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 5.12: `@ConvertGroup` *usage*

```java
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;
    }

    public int getAge() {
        return age;
    }

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

    // 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;
    }

    public boolean isPassedVehicleInspection() {
        return passedVehicleInspection;
    }

    public void setPassedVehicleInspection(boolean passedVehicleInspection) {
        this.passedVehicleInspection = passedVehicleInspection;
    }

    public Driver getDriver() {
        return driver;
    }

    public void setDriver(Driver driver) {
        this.driver = driver;
    }

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

As a result the validation in Example 5.13, "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 5.13: 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.

> 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.

Chapter 6. Creating custom constraints

The Jakarta Bean Validation API defines a whole set of standard constraint annotations such as `@NotNull`, `@Size` etc. In cases where these built-in constraints are not sufficient, you can 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 an 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

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

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE_USE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE, TYPE_USE })
@Retention(RUNTIME)
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
@Repeatable(List.class)
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 Jakarta 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 Jakarta 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 behavior 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, TYPE_USE})`: 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`), method/constructor parameters (`PARAMETER`) and type argument of parameterized types (`TYPE_USE`). 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 2.1.4, "Class-level constraints"), the element type `TYPE` would have to be used. Constraints targeting 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

- `@Repeatable(List.class)`: Indicates that the annotation can be repeated several times at the same place, usually with a different configuration. `List` is the containing annotation type.

This containing annotation type named `List` is also shown in the example. It allows to specify several `@CheckCase` annotations on the same element, e.g. with different validation groups and messages. While another name could be used, the Jakarta 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 Jakarta Bean Validation 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 Jakarta 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*

```java
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.chapter06." +
                    "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.

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

By default, Expression Language is not enabled for custom violations created in the `ConstraintValidatorContext`.

However, for some advanced requirements, using Expression Language might be necessary.

In this case, you need to unwrap the `HibernateConstraintValidatorContext` and enable Expression Language explicitly. See Section 12.13.1, "`HibernateConstraintValidatorContext`"

for more information.

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.

6.1.2.2. The `HibernateConstraintValidator` extension

Hibernate Validator provides an extension to the `ConstraintValidator` contract: `HibernateConstraintValidator`.

The purpose of this extension is to provide more contextual information to the `initialize()` method as, in the current `ConstraintValidator` contract, only the annotation is passed as parameter.

The `initialize()` method of `HibernateConstraintValidator` takes two parameters:

- The `ConstraintDescriptor` of the constraint at hand. You can get access to the annotation using `ConstraintDescriptor#getAnnotation()`.
- The `HibernateConstraintValidatorInitializationContext` which provides useful helpers and contextual information, such as the clock provider or the temporal validation tolerance.

This extension is marked as incubating so it might be subject to change. The plan is to standardize it and to include it in Jakarta Bean Validation in the future.

The example below shows how to base your validators on `HibernateConstraintValidator`:

Example 6.5: Using the `HibernateConstraintValidator` *contract*

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

public class MyFutureValidator implements HibernateConstraintValidator<MyFuture, Instant> {

    private Clock clock;

    private boolean orPresent;

    @Override
    public void initialize(ConstraintDescriptor<MyFuture> constraintDescriptor,
            HibernateConstraintValidatorInitializationContext initializationContext) {
        this.orPresent = constraintDescriptor.getAnnotation().orPresent();
        this.clock = initializationContext.getClockProvider().getClock();
    }

    @Override
    public boolean isValid(Instant instant, ConstraintValidatorContext constraintContext) {
        //...

        return false;
    }
}
```

> ⚠️ You should only implement one of the `initialize()` methods. Be aware that both are called when initializing the validator.

6.1.2.3. Passing a payload to the constraint validator

From time to time, you might want to condition the constraint validator behavior on some external parameters.

For instance, your zip code validator could vary depending on the locale of your application instance if you have one instance per country. Another requirement could be to have different behaviors on specific environments: the staging environment may not have access to some external production resources necessary for the correct functioning of a validator.

The notion of constraint validator payload was introduced for all these use cases. It is an object passed from the `Validator` instance to each constraint validator via the `HibernateConstraintValidatorContext`.

The example below shows how to set a constraint validator payload during the `ValidatorFactory` initialization. Unless you override this default value, all the `Validator`s created by this `ValidatorFactory` will have this constraint validator payload value set.

Example 6.6: Defining a constraint validator payload during the `ValidatorFactory` *initialization*

```
ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
        .configure()
        .constraintValidatorPayload( "US" )
        .buildValidatorFactory();

Validator validator = validatorFactory.getValidator();
```

Another option is to set the constraint validator payload per `Validator` using a context:

Example 6.7: Defining a constraint validator payload using a `Validator` *context*

```
HibernateValidatorFactory hibernateValidatorFactory = Validation.byDefaultProvider()
        .configure()
        .buildValidatorFactory()
        .unwrap( HibernateValidatorFactory.class );

Validator validator = hibernateValidatorFactory.usingContext()
        .constraintValidatorPayload( "US" )
        .getValidator();

// [...] US specific validation checks

validator = hibernateValidatorFactory.usingContext()
        .constraintValidatorPayload( "FR" )
        .getValidator();

// [...] France specific validation checks
```

Once you have set the constraint validator payload, it can be used in your constraint validators as shown in the example below:

Example 6.8: Using the constraint validator payload in a constraint validator

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

public class ZipCodeValidator implements ConstraintValidator<ZipCode, String> {

    public String countryCode;

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

        boolean isValid = false;

        String countryCode = constraintContext
                .unwrap( HibernateConstraintValidatorContext.class )
                .getConstraintValidatorPayload( String.class );

        if ( "US".equals( countryCode ) ) {
            // checks specific to the United States
        }
        else if ( "FR".equals( countryCode ) ) {
            // checks specific to France
        }
        else {
            // ...
        }

        return isValid;
    }
}
```

HibernateConstraintValidatorContext#getConstraintValidatorPayload() has a type parameter and returns the payload only if the payload is of the given type.

> It is important to note that the constraint validator payload is different from the dynamic payload you can include in the constraint violation raised.
>
> The whole purpose of this constraint validator payload is to be used to condition the behavior of your constraint validators. It is not included in the constraint violations, unless a specific ConstraintValidator implementation passes on the payload to emitted constraint violations by using the constraint violation dynamic payload mechanism.

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 6.9: 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 6.10: Applying the `@CheckCase` *constraint*

```java
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 6.11, "Validating objects with the `@CheckCase` constraint" demonstrates how validating a `Car` instance with an invalid license plate causes the `@CheckCase` constraint to be violated.

Example 6.11: 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 way as are property constraints. Example 6.12, "Implementing a class-level constraint" shows constraint annotation and validator of the @ValidPassengerCount constraint you already saw in use in Example 2.9, "Class-level constraint".

Example 6.12: 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 6.13, "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 6.13: Adding a new `ConstraintViolation` *with custom property path*

```java
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

Jakarta 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, container element, 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 6.14: Cross-parameter constraint

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

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

    String message() default "{org.hibernate.validator.referenceguide.chapter04." +
            "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 6.15, "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").

ℹ️ 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 choose constraint names - such as `@ConsistentDateParameters` - which make the constraint target apparent.

Example 6.15: Generic and cross-parameter constraint

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

@SupportedValidationTarget(ValidationTarget.PARAMETERS)
public class ConsistentDateParametersValidator 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`.

> 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 6.16, "Generic and cross-parameter constraint".

Example 6.16: Generic and cross-parameter constraint

```java
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.chapter04." +
            "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 6.17, "Specifying the target for a generic and cross-parameter constraint".

Example 6.17: Specifying the target for a generic and cross-parameter constraint

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

6.4. Constraint composition

Looking at the `licensePlate` field of the `Car` class in Example 6.10, "Applying the `@CheckCase` constraint", you see three constraint annotations already. In more complex scenarios, where even more constraints could be applied to one element, this might easily become a bit confusing. 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 6.18, "Creating a composing constraint `@ValidLicensePlate`" shows a composed constraint annotation which comprises the constraints `@NotNull`, `@Size` and `@CheckCase`:

Example 6.18: Creating a composing constraint `@ValidLicensePlate`

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

@NotNull
@Size(min = 2, max = 14)
@CheckCase(CaseMode.UPPER)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, TYPE_USE })
@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 6.19: Application of composing constraint `ValidLicensePlate`

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

public class Car {

    @ValidLicensePlate
    private String licensePlate;

    //...
}
```

The set of `ConstraintViolation`s 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 6.20: Using @ReportAsSingleViolation

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

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

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

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

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

Chapter 7. Value extraction

Value extraction is the process of extracting values from a container so that they can be validated.

It is used when dealing with container element constraints and cascaded validation inside containers.

7.1. Built-in value extractors

Hibernate Validator comes with built-in value extractors for the usual Java container types so, except if you are using your own custom container types (or the ones of external libraries such as Guava's `Multimap`), you should not have to add your own value extractors.

Built-in value extractors are present for all the following container types:

- `java.util.Iterable`;

- `java.util.List`;

- `java.util.Map`: for keys and values;

- `java.util.Optional`, `java.util.OptionalInt`, `java.util.OptionalLong` and `java.util.OptionalDouble`;

- JavaFX's `ObservableValue` (see Section 7.4, "JavaFX value extractors" for more details).

The complete list of built-in value extractors with all the details on how they behave can be found in the Jakarta Bean Validation specification.

7.2. Implementing a `ValueExtractor`

To extract values from a custom container, one needs to implement a `ValueExtractor`.

> Implementing a `ValueExtractor` is not enough, you also need to register it. See Section 7.5, "Registering a `ValueExtractor`" for more details.

`ValueExtractor` is a very simple API as the only purpose of a value extractor is to provide the extracted values to a `ValueReceiver`.

For instance, let's consider the case of Guava's `Optional`. It is an easy example as we can shape its value extractor after the `java.util.Optional` one:

Example 7.1: A `ValueExtractor` for Guava's `Optional`

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

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

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

Some explanations are in order:

- The `@ExtractedValue` annotation marks the type argument under consideration: it is going to be used to resolve the type of the validated value;

- We use the `value()` method of the receiver as `Optional` is a pure wrapper type;

- We don't want to add a node to the property path of the constraint violation as we want the violation to be reported as if it were directly on the property so we pass a `null` node name to `value()`.

A more interesting example is the case of Guava's `Multimap`: we would like to be able to validate both the keys and the values of this container type.

Let's first consider the case of the values. A value extractor extracting them is required:

Example 7.2: A `ValueExtractor` for `Multimap` values

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

public class MultimapValueValueExtractor
        implements ValueExtractor<Multimap<?, @ExtractedValue ?>> {

    @Override
    public void extractValues(Multimap<?, ?> originalValue, ValueReceiver receiver) {
        for ( Entry<?, ?> entry : originalValue.entries() ) {
            receiver.keyedValue( "<multimap value>", entry.getKey(), entry.getValue() );
        }
    }
}
```

It allows to validate constraints for the values of the `Multimap`:

Example 7.3: Constraints on the values of a `Multimap`

```
private Multimap<String, @NotBlank String> map1;
```

Another value extractor is required to be able to put constraints on the keys of a `Multimap`:

Example 7.4: A `ValueExtractor` for `Multimap` keys

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

public class MultimapKeyValueExtractor
        implements ValueExtractor<Multimap<@ExtractedValue ?, ?>> {

    @Override
    public void extractValues(Multimap<?, ?> originalValue, ValueReceiver receiver) {
        for ( Object key : originalValue.keySet() ) {
            receiver.keyedValue( "<multimap key>", key, key );
        }
    }
}
```

Once these two value extractors are registered, you can declare constraints on the keys and values of a `Multimap`:

Example 7.5: Constraints on the keys and values of a `Multimap`

```
private Multimap<@NotBlank String, @NotBlank String> map2;
```

The differences between the two value extractors may be a bit subtle at a first glance so let's shed some light on them:

- The `@ExtractedValue` annotation marks the targeted type argument (either `K` or `V` in this case).

- We use different node names (`<multimap key>` vs. `<multimap value>`).

- In one case, we pass the values to the receiver (third argument of the `keyedValue()` call), in the other, we pass the keys.

Depending on your container type, you should choose the `ValueReceiver` method fitting the best:

`value()`

 for a simple wrapping container - it is used for `Optional`s

`iterableValue()`

 for an iterable container - it is used for `Set`s

`indexedValue()`

 for a container containing indexed values - it is used for `List`s

`keyedValue()`

 for a container containing keyed values - it is used for `Map`s. It is used for both the keys and the values. In the case of keys, the key is also passed as the validated value.

For all these methods, you need to pass a node name: it is the name included in the node added to the property path of the constraint violation. As mentioned earlier, if the node name is `null`, no node is added to the property path: it is be useful for pure wrapper types similar to `Optional`.

The choice of the method used is important as it adds contextual information to the property path of the constraint violation e.g. the index or the key of the validated value.

7.3. Non generic containers

You might have noticed that, until now, we only implemented value extractors for generic containers.

Hibernate Validator also supports value extraction for non generic containers.

Let's take the case of `java.util.OptionalInt` which wraps a primitive `int` into an `Optional`-like container.

A first attempt at a value extractor for `OptionalInt` would look like:

Example 7.6: A `ValueExtractor` *for* `OptionalInt`

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

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

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

There is an obvious thing missing for a non generic container: we don't have a type parameter. It has two consequences:

- we cannot determine the type of the validated value using the type argument;
- we cannot add constraints on the type argument (e.g. `Container<@NotNull String>`).

First things first, we need a way to tell Hibernate Validator that the value extracted from an `OptionalInt` is of type `Integer`. As you can see in the above example, the `type` attribute of the `@ExtractedValue` annotation allows to provide this information to the validation engine.

Then you have to tell the validation engine that the `Min` constraint you want to add to the `OptionalInt` property relates to the wrapped value and not the wrapper.

Jakarta Bean Validation provides the `Unwrapping.Unwrap` payload for this situation:

Example 7.7: Using `Unwrapping.Unwrap` *payload*

```
@Min(value = 5, payload = Unwrapping.Unwrap.class)
private OptionalInt optionalInt1;
```

If we take a step back, most - if not all - the constraints we would like to add to an `OptionalInt` property would be applied to the wrapped value so having a way to make it the default would be nice.

This is exactly what the `@UnwrapByDefault` annotation is for:

Example 7.8: A `ValueExtractor` *for* `OptionalInt` *marked with* `@UnwrapByDefault`

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

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

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

When declaring this value extractor for `OptionalInt`, constraint annotations will by default be applied to the wrapped value:

Example 7.9: Implicit unwrapping thanks to `@UnwrapByDefault`

```
@Min(5)
private OptionalInt optionalInt2;
```

Note that you can still declare an annotation for the wrapper itself by using the `Unwrapping.Skip` payload:

Example 7.10: Avoid implicit unwrapping with `Unwrapping.Skip`

```
@NotNull(payload = Unwrapping.Skip.class)
@Min(5)
private OptionalInt optionalInt3;
```

> ℹ The `@UnwrapByDefault` value extractor for `OptionalInt` is part of the built-in value extractors: there is no need to add one.

7.4. JavaFX value extractors

Bean properties in JavaFX 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.

Thus, value extraction is required to be able to apply constraints on the wrapped values.

The JavaFX `ObservableValue` value extractor is marked with `@UnwrapByDefault`. As such, the constraints hosted on the container target the wrapped value by default.

Thus, you can constrain a `StringProperty` as below:

Example 7.11: Constraining a `StringProperty`

```
@NotBlank
private StringProperty stringProperty;
```

Or a `LongProperty`:

Example 7.12: Constraining a `LongProperty`

```
@Min(5)
private LongProperty longProperty;
```

The iterable property types, namely `ReadOnlyListProperty`, `ListProperty` and their `Set` and `Map` counterparts are generic and, as such, container element constraints can be used. Thus, they have specific value extractors that are not marked with `@UnwrapByDefault`.

A `ReadOnlyListProperty` would naturally be constrained as a `List`:

Example 7.13: Constraining a `ReadOnlyListProperty`

```
@Size(min = 1)
private ReadOnlyListProperty<@NotBlank String> listProperty;
```

7.5. Registering a `ValueExtractor`

Hibernate Validator does not detect automatically the value extractors in the classpath so they have to be registered.

There are several ways to register value extractors (in increasing order of priority):

Provided by the validation engine itself

See Section 7.1, "Built-in value extractors".

Via the Java service loader mechanism

The file `META-INF/services/jakarta.validation.valueextraction.ValueExtractor` must be provided, with the fully-qualified names of one or more value extractor implementations as its contents, each on a separate line.

In the `META-INF/validation.xml` *file*

See Section 8.1, "Configuring the validator factory in *validation.xml*" for more information about how to register value extractors in the XML configuration.

By calling `Configuration#addValueExtractor(ValueExtractor<?>)`

See Section 9.2.6, "Registering `ValueExtractors`" for more information.

By invoking `ValidatorContext#addValueExtractor(ValueExtractor<?>)`

It only declares the value extractor for this `Validator` instance.

A value extractor for a given type and type parameter specified at a higher priority overrides any other extractors for the same type and type parameter given at lower priorities.

7.6. Resolution algorithms

In most cases, you should not have to worry about this but, if you are overriding existing value extractors, you can find a detailed description of the value extractors resolution algorithms in the Jakarta Bean Validation specification:

- for container element constraints,
- for cascaded validation,
- and for implicit unwrapping.

One important thing to have in mind is that:

- for container element constraints, the declared type is used to resolve the value extractors;
- for cascaded validation, it is the runtime type.

Chapter 8. Configuring via XML

So far we have used the default configuration source for Jakarta Bean Validation, namely annotations. However, there also exist two kinds of XML descriptors allowing configuration via XML. The first descriptor describes general Jakarta 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.

> **ℹ** The XSD files are available on the https://jakarta.ee/xml/ns/validation/ page.

8.1. 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. Figure 1, "Validation configuration schema" shows a model view of the XML schema to which *validation.xml* has to adhere.
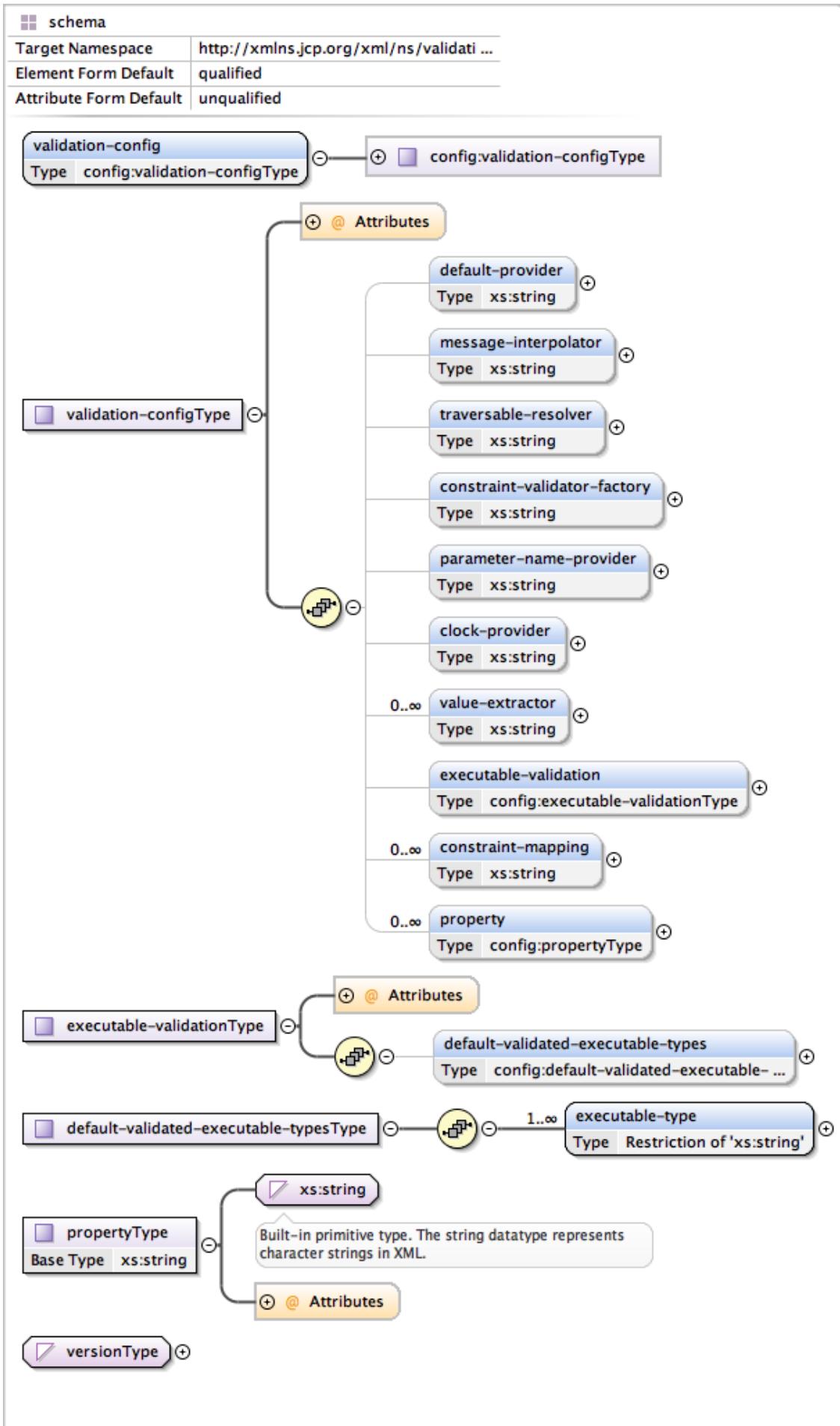
Figure 1. Validation configuration schema

Example 8.1, "`validation.xml`" shows the several configuration options of *validation.xml*. All settings are optional and the same configuration options are also available programmatically through `jakarta.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 9.2, "Configuring a `ValidatorFactory`".

Example 8.1: `validation.xml`

```xml
<validation-config
        xmlns="https://jakarta.ee/xml/ns/validation/configuration"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/configuration
            https://jakarta.ee/xml/ns/validation/validation-configuration-3.0.xsd"
        version="3.0">

    <default-provider>com.acme.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>
    <clock-provider>com.acme.ClockProvider</clock-provider>

    <value-extractor>com.acme.ContainerValueExtractor</value-extractor>

    <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>
```

> ⚠️ 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 Jakarta Bean Validation provider. This is useful if there is more than one provider on the classpath. `message-interpolator`, `traversable-resolver`, `constraint-validator-factory`, `parameter-name-provider` and `clock-provider` allow to customize the used implementations for the interfaces `MessageInterpolator`, `TraversableResolver`, `ConstraintValidatorFactory`, `ParameterNameProvider` and `ClockProvider` defined in the `jakarta.validation` package. See the sub-sections of Section 9.2, "Configuring a `ValidatorFactory`" for more information about these interfaces.

`value-extractor` allows to declare additional value extractors either to extract values from custom container types or to override the built-in value extractors. See Chapter 7, *Value extraction* for more

information about how to implement `jakarta.validation.valueextraction.ValueExtractor`.

`executable-validation` and its subnodes define defaults for method validation. The Jakarta 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 12.2, "Fail fast mode").

8.2. Mapping constraints via `constraint-mappings`

Expressing constraints in XML is possible via files adhering to the schema seen in Figure 2, "Validation mapping schema". Note that these mapping files are only processed if listed via constraint-mapping in *validation.xml*.
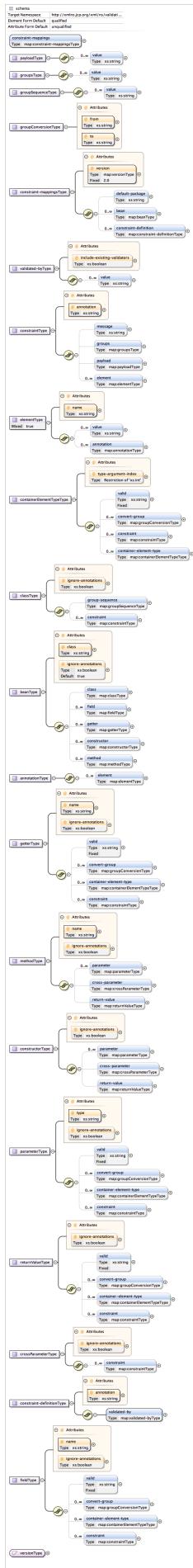
Figure 2. Validation mapping schema

Example 8.2, "Bean constraints configured via XML" shows how the classes Car and RentalCar from Example 5.3, "Car" resp. Example 5.9, "Class `RentalCar` with redefined default group" could be mapped in XML.

Example 8.2: Bean constraints configured via XML

```xml
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/validation-mapping-3.0.xsd"
        version="3.0">

    <default-package>org.hibernate.validator.referenceguide.chapter05</default-package>
    <bean class="Car" ignore-annotations="true">
        <field name="manufacturer">
            <constraint annotation="jakarta.validation.constraints.NotNull"/>
        </field>
        <field name="licensePlate">
            <constraint annotation="jakarta.validation.constraints.NotNull"/>
        </field>
        <field name="seatCount">
            <constraint annotation="jakarta.validation.constraints.Min">
                <element name="value">2</element>
            </constraint>
        </field>
        <field name="driver">
            <valid/>
        </field>
        <field name="partManufacturers">
            <container-element-type type-argument-index="0">
                <valid/>
            </container-element-type>
            <container-element-type type-argument-index="1">
                <container-element-type>
                    <valid/>
                    <constraint annotation="jakarta.validation.constraints.NotNull"/>
                </container-element-type>
            </container-element-type>
        </field>
        <getter name="passedVehicleInspection" ignore-annotations="true">
            <constraint annotation="jakarta.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 8.3, "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 8.3: Method constraints configured via XML

```xml
<constraint-mappings
        xmlns="https://jakarta.ee/xml/ns/validation/mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/mapping
            https://jakarta.ee/xml/ns/validation/validation-mapping-3.0.xsd"
        version="3.0">

    <default-package>org.hibernate.validator.referenceguide.chapter08</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="jakarta.validation.constraints.NotNull"/>
            </parameter>
        </constructor>

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

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

        <method name="addCars">
            <parameter type="java.util.List">
                <container-element-type>
                    <valid/>
                    <constraint annotation="jakarta.validation.constraints.NotNull"/>
                </container-element-type>
            </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.

> A given class 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`, `container-element-type`, `constructor` and `method` (and its sub node parameter) determine on which level the constraint gets placed. The `valid` node is used to enable cascaded validation and the `constraint` node 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 Jakarta Bean Validation specification (`message`, `groups` and `payload`) have dedicated nodes. All other constraint specific attributes are configured using the `element` node.

> `container-element-type` allows to define the cascaded validation behavior and the constraints for container elements. In the above examples, you can see an example of nested container element constraints on a `List` nested in the values of a `Map`. `type-argument-index` is used to precise which type argument of the `Map` is concerned by the configuration. It can be ommitted if the type only has one type argument (e.g. the `List`s in our examples).

The `class` node also allows to reconfigure the default group sequence (see Section 5.4, "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 5.5, "Group conversion"). This node is available on `field`, `getter`, `container-element-type`, `parameter` and `return-value` and specifies a `from` and a `to` attributes to specify the groups.

Last but not least, the list of `ConstraintValidator` instances 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` element 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.

> One use case for constraint-definition is to change the default constraint definition for @URL. Historically, Hibernate Validator's default constraint validator for this constraint uses the `java.net.URL` constructor to verify that an URL is valid. However, there is also a purely regular expression based version available which can be configured using XML:
>
> *Using XML to register a regular expression based constraint definition for @URL*
>
> ```xml
> <constraint-definition annotation="org.hibernate.validator.constraints.URL">
>   <validated-by include-existing-validators="false">
>     <value>
> org.hibernate.validator.constraintvalidators.RegexpURLValidator</value>
>   </validated-by>
> </constraint-definition>
> ```

Chapter 9. Bootstrapping

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

9.1. Retrieving `ValidatorFactory` and `Validator`

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

Example 9.1, "Bootstrapping default `ValidatorFactory` and `Validator`" shows how to obtain a validator from the default validator factory:

Example 9.1: Bootstrapping default `ValidatorFactory` and `Validator`

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

> 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.

Jakarta 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 9.2, "Bootstrapping `ValidatorFactory` and `Validator` using a specific provider".

Example 9.2: 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 9.3, "Retrieving the default `ValidatorFactory` for configuration".

Example 9.3: Retrieving the default `ValidatorFactory` for configuration

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

> ℹ️ 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.

9.1.1. `ValidationProviderResolver`

By default, available Jakarta Bean Validation providers are discovered using the Java Service Provider mechanism.

For that purpose, each provider includes the file *META-INF/services/jakarta.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 in Example 9.4, "Using a custom `ValidationProviderResolver`".

Example 9.4: Using a custom `ValidationProviderResolver`

```java
package org.hibernate.validator.referenceguide.chapter09;

public class OsgiServiceDiscoverer implements ValidationProviderResolver {

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

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

9.2. 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 8, *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 Jakarta 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.

9.2.1. `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 9.5, "Using a custom `MessageInterpolator`".

Example 9.5: Using a custom `MessageInterpolator`

```java
package org.hibernate.validator.referenceguide.chapter09;

public class MyMessageInterpolator implements MessageInterpolator {

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

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

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

9.2.2. `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 9.6, "Using a custom `TraversableResolver`" shows how to use a custom traversable resolver implementation.

Example 9.6: Using a custom `TraversableResolver`

```
package org.hibernate.validator.referenceguide.chapter09;

public class MyTraversableResolver implements TraversableResolver {

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

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

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

If no specific traversable resolver has been configured, the default behavior is to consider all properties as reachable and cascadable. When using Hibernate Validator together with a JPA 2 provider such as Hibernate ORM, only those properties will be considered reachable which already have been loaded by the persistence provider and all properties will be considered cascadable.

By default, the traversable resolver calls are cached per validation call. This is especially important in a JPA environment where calling `isReachable()` has a significant cost.

This caching adds some overhead. In the case your custom traversable resolver is very fast, it might be better to consider turning off the cache.

You can disable the cache either via the XML configuration:

Example 9.7: Disabling the `TraversableResolver` *result cache via the XML configuration*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
        xmlns="https://jakarta.ee/xml/ns/validation/configuration"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/configuration
https://jakarta.ee/xml/ns/validation/validation-configuration-3.0.xsd"
        version="3.0">
    <default-provider>org.hibernate.validator.HibernateValidator</default-provider>

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

or via the programmatic API:

Example 9.8: Disabling the `TraversableResolver` *result cache via the programmatic API*

```java
ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
        .configure()
        .traversableResolver( new MyFastTraversableResolver() )
        .enableTraversableResolverResultCache( false )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

9.2.3. `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 9.9, "Using a custom `ConstraintValidatorFactory`".

Example 9.9: Using a custom `ConstraintValidatorFactory`

```java
package org.hibernate.validator.referenceguide.chapter09;

public class MyConstraintValidatorFactory implements ConstraintValidatorFactory {

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

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

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

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

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

9.2.4. `ParameterNameProvider`

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

The default implementation returns parameter names as obtained through the Java reflection API. If you compile your sources using the `–parameters` compiler flag, the actual parameter names as in the source code will be returned. Otherwise synthetic names in the form of `arg0`, `arg1` etc. will be used.

To use a custom parameter name provider either pass an instance of the provider during bootstrapping as shown in Example 9.10, "Using a custom `ParameterNameProvider`", or specify the fully qualified class name of the provider as value for the `<parameter-name-provider>` element in the *META-INF/validation.xml* file (see Section 8.1, "Configuring the validator factory in *validation.xml*"). This is demonstrated in Example 9.10, "Using a custom `ParameterNameProvider`".

Example 9.10: Using a custom `ParameterNameProvider`

```java
package org.hibernate.validator.referenceguide.chapter09;

public class MyParameterNameProvider implements ParameterNameProvider {

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

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

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

Hibernate Validator comes with a custom `ParameterNameProvider` implementation based on the ParaNamer library which provides several ways for obtaining parameter names at runtime. Refer to Section 12.14, "Paranamer based `ParameterNameProvider`" to learn more about this specific implementation.

9.2.5. `ClockProvider` **and temporal validation tolerance**

For time related validation (`@Past` and `@Future` constraints for instance), it might be useful to define what is considered now.

This is especially important when you want to test your constraints in a reliable manner.

The reference time is defined by the `ClockProvider` contract. The responsibility of the `ClockProvider` is to provide a `java.time.Clock` defining now for time related validators.

Example 9.11: Using a custom `ClockProvider`

```java
package org.hibernate.validator.referenceguide.chapter09;

public class FixedClockProvider implements ClockProvider {

    private Clock clock;

    public FixedClockProvider(ZonedDateTime dateTime) {
        clock = Clock.fixed( dateTime.toInstant(), dateTime.getZone() );
    }

    @Override
    public Clock getClock() {
        return clock;
    }

}
```

```java
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .configure()
        .clockProvider( new FixedClockProvider( ZonedDateTime.of( 2016, 6, 15, 0, 0, 0, 0,
ZoneId.of( "Europe/Paris" ) ) ) )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

Alternatively, you can specify the fully-qualified classname of a `ClockProvider` implementation using the `<clock-provider>` element when configuring the default validator factory via *META-INF/validation.xml* (see Chapter 8, *Configuring via XML*).

> When validating `@Future` and `@Past` constraints, you might want to obtain the current time.
>
> You can obtain the `ClockProvider` in your validators by calling the `ConstraintValidatorContext#getClockProvider()` method.
>
> For instance, this might be useful if you want to replace the default message of the `@Future` constraint with a more explicit one.

When dealing with distributed architectures, you might need some tolerance when applying temporal constraints such as `@Past` or `@Future`.

You can set a temporal validation tolerance by bootstrapping your `ValidatorFactory` as below:

Example 9.12: Using temporal validation tolerance

```
ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
        .configure()
        .temporalValidationTolerance( Duration.ofMillis( 10 ) )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

Alternatively, you can define it in the XML configuration by setting the `hibernate.validator.temporal_validation_tolerance` property in your *META-INF/validation.xml*.

The value of this property must be a `long` defining the tolerance in milliseconds.

When implementing your own temporal constraints, you might need to have access to the temporal validation tolerance.

It can be obtained by calling the `HibernateConstraintValidatorInitializationContext#getTemporalValidationTolerance()` method.

Note that to get access to this context at initialization, your constraint validator has to implement the `HibernateConstraintValidator` contract (see Section 6.1.2.2, "The `HibernateConstraintValidator` extension"). This contract is currently marked as incubating: it might be subject to change in the future.

9.2.6. Registering `ValueExtractor`s

As mentioned in Chapter 7, *Value extraction*, additional value extractors can be registered during bootstrapping (see Section 7.5, "Registering a `ValueExtractor`" for the other ways to register a value extractor).

Example 9.13, "Registering additional value extractors" shows how we would register the value extractors we previously created to extract the keys and the values of Guava's `Multimap`.

Example 9.13: Registering additional value extractors

```
ValidatorFactory validatorFactory = Validation.byDefaultProvider()
        .configure()
        .addValueExtractor( new MultimapKeyValueExtractor() )
        .addValueExtractor( new MultimapValueValueExtractor() )
        .buildValidatorFactory();
Validator validator = validatorFactory.getValidator();
```

9.2.7. Adding mapping streams

As discussed earlier, you can configure the constraints applied to 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 9.14, "Adding constraint mapping streams"). Note that the passed input stream(s) must adhere to the XML schema for constraint mappings presented in Section 8.2, "Mapping constraints via `constraint-mappings`".

Example 9.14: Adding constraint mapping streams

```
InputStream constraintMapping1 = null;
InputStream constraintMapping2 = null;
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.

9.2.8. Provider-specific settings

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

In the 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 9.15, "Setting Hibernate Validator specific options".

Example 9.15: 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 9.16: 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 12.2, "Fail fast mode" and Section 12.4, "Programmatic constraint definition and declaration" to learn more about the fail fast mode and the constraint declaration API.

9.2.9. Configuring the `ScriptEvaluatorFactory`

For constraints like `@ScriptAssert` and `@ParameterScriptAssert`, it might be useful to configure how the script engines are initialized and how the script evaluators are built. This can be done by setting a custom implementation of `ScriptEvaluatorFactory`.

In particular, this is important for modular environments (e.g. OSGi), where user might face issues with modular class loading and JSR 223. It also allows to use any custom script engine, not necessarily based on the JSR 223 (e.g. Spring Expression Language).

9.2.9.1. XML configuration

To specify the `ScriptEvaluatorFactory` via XML, you need to define the `hibernate.validator.script_evaluator_factory` property.

Example 9.17: Defining the `ScriptEvaluatorFactory` *via XML*

```
<validation-config
        xmlns="https://jakarta.ee/xml/ns/validation/configuration"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/configuration
            https://jakarta.ee/xml/ns/validation/validation-configuration-3.0.xsd"
        version="3.0">

    <property name="hibernate.validator.script_evaluator_factory">
        org.hibernate.validator.referenceguide.chapter09.CustomScriptEvaluatorFactory
    </property>

</validation-config>
```

In this case, the specified `ScriptEvaluatorFactory` must have a no-arg constructor.

9.2.9.2. Programmatic configuration

To configure it programmatically, you need to pass an instance of `ScriptEvaluatorFactory` to the `ValidatorFactory`. This gives more flexibility in the configuration of the `ScriptEvaluatorFactory`. Example 9.18, "Defining the `ScriptEvaluatorFactory`

programmatically" shows how this can be done.

Example 9.18: Defining the `ScriptEvaluatorFactory` *programmatically*

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

9.2.9.3. Custom `ScriptEvaluatorFactory` **implementation examples**

This section shows a couple of custom `ScriptEvaluatorFactory` implementations that can be used in modular environments as well as one using the Spring Expression Language for writing constraint scripts.

Problems with modular environments and JSR 223 come from the class loading. The class loader where the script engine is available might be different from the one of Hibernate Validator. Thus the script engine wouldn't be found using the default strategy.

To solve this issue, the `MultiClassLoaderScriptEvaluatorFactory` class below can be introduced:

```java
/*
 * Hibernate Validator, declare and validate application constraints
 *
 * License: Apache License, Version 2.0
 * See the license.txt file in the root directory or
<http://www.apache.org/licenses/LICENSE-2.0>.
 */
package org.hibernate.validator.osgi.scripting;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

import org.hibernate.validator.spi.scripting.AbstractCachingScriptEvaluatorFactory;
import org.hibernate.validator.spi.scripting.ScriptEngineScriptEvaluator;
import org.hibernate.validator.spi.scripting.ScriptEvaluationException;
import org.hibernate.validator.spi.scripting.ScriptEvaluator;
import org.hibernate.validator.spi.scripting.ScriptEvaluatorFactory;

/**
 * {@link ScriptEvaluatorFactory} that allows you to pass multiple {@link ClassLoader}s
that will be used
 * to search for {@link ScriptEngine}s. Useful in environments similar to OSGi, where
script engines can be
 * found only in {@link ClassLoader}s different from default one.
 *
 * @author Marko Bekhta
 */
public class MultiClassLoaderScriptEvaluatorFactory extends
AbstractCachingScriptEvaluatorFactory {

    private final ClassLoader[] classLoaders;

    public MultiClassLoaderScriptEvaluatorFactory(ClassLoader... classLoaders) {
        if ( classLoaders.length == 0 ) {
            throw new IllegalArgumentException( "No class loaders were passed" );
        }
        this.classLoaders = classLoaders;
    }

    @Override
    protected ScriptEvaluator createNewScriptEvaluator(String languageName) {
        for ( ClassLoader classLoader : classLoaders ) {
            ScriptEngine engine = new ScriptEngineManager( classLoader ).getEngineByName(
languageName );
            if ( engine != null ) {
                return new ScriptEngineScriptEvaluator( engine );
            }
        }
        throw new ScriptEvaluationException( "No JSR 223 script engine found for language "
+ languageName );
    }
}
```

and then declared with:

```
Validator validator = Validation.byProvider( HibernateValidator.class )
        .configure()
        .scriptEvaluatorFactory(
                new MultiClassLoaderScriptEvaluatorFactory( GroovyScriptEngineFactory.
class.getClassLoader() )
        )
        .buildValidatorFactory()
        .getValidator();
```

This way, it is possible to pass multiple `ClassLoader` instances: typically the class loaders of the wanted `ScriptEngine`s.

An alternative approach for OSGi environments can be to use the `OsgiScriptEvaluatorFactory` defined below:

```
/*
 * Hibernate Validator, declare and validate application constraints
 *
 * License: Apache License, Version 2.0
 * See the license.txt file in the root directory or
<http://www.apache.org/licenses/LICENSE-2.0>.
 */
package org.hibernate.validator.osgi.scripting;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.Arrays;
import java.util.Collections;
import java.util.Enumeration;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import javax.script.ScriptEngineFactory;
import javax.script.ScriptEngineManager;
import jakarta.validation.ValidationException;

import org.hibernate.validator.spi.scripting.AbstractCachingScriptEvaluatorFactory;
import org.hibernate.validator.spi.scripting.ScriptEngineScriptEvaluator;
import org.hibernate.validator.spi.scripting.ScriptEvaluator;
import org.hibernate.validator.spi.scripting.ScriptEvaluatorFactory;
import org.hibernate.validator.spi.scripting.ScriptEvaluatorNotFoundException;
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;

/**
 * {@link ScriptEvaluatorFactory} suitable for OSGi environments. It is created
 * based on the {@code BundleContext} which is used to iterate through {@code Bundle}s and
find all {@link ScriptEngineFactory}
 * candidates.
 *
 * @author Marko Bekhta
 */
public class OsgiScriptEvaluatorFactory extends AbstractCachingScriptEvaluatorFactory {

    private final List<ScriptEngineManager> scriptEngineManagers;
```

```java
    public OsgiScriptEvaluatorFactory(BundleContext context) {
        this.scriptEngineManagers = Collections.unmodifiableList( findManagers( context )
);
    }

    @Override
    protected ScriptEvaluator createNewScriptEvaluator(String languageName) throws
ScriptEvaluatorNotFoundException {
        return scriptEngineManagers.stream()
                .map( manager -> manager.getEngineByName( languageName ) )
                .filter( Objects::nonNull )
                .map( engine -> new ScriptEngineScriptEvaluator( engine ) )
                .findFirst()
                .orElseThrow( () -> new ValidationException( String.format( "Unable to find
script evaluator for '%s'.", languageName ) ) );
    }

    private List<ScriptEngineManager> findManagers(BundleContext context) {
        return findFactoryCandidates( context ).stream()
                .map( className -> {
                    try {
                        return new ScriptEngineManager( Class.forName( className )
.getClassLoader() );
                    }
                    catch (ClassNotFoundException e) {
                        throw new ValidationException( "Unable to instantiate '" +
className + "' based engine factory manager.", e );
                    }
                } ).collect( Collectors.toList() );
    }

    /**
     * Iterates through all bundles to get the available {@link ScriptEngineFactory}
classes
     *
     * @return the names of the available ScriptEngineFactory classes
     *
     * @throws IOException
     */
    private List<String> findFactoryCandidates(BundleContext context) {
        return Arrays.stream( context.getBundles() )
                .filter( Objects::nonNull )
                .filter( bundle -> !"system.bundle".equals( bundle.getSymbolicName() ) )
                .flatMap( this::toStreamOfResourcesURL )
                .filter( Objects::nonNull )
                .flatMap( url -> toListOfFactoryCandidates( url ).stream() )
                .collect( Collectors.toList() );
    }

    private Stream<URL> toStreamOfResourcesURL(Bundle bundle) {
        Enumeration<URL> entries = bundle.findEntries(
                "META-INF/services",
                "javax.script.ScriptEngineFactory",
                false
        );
        return entries != null ? Collections.list( entries ).stream() : Stream.empty();
    }

    private List<String> toListOfFactoryCandidates(URL url) {
        try ( BufferedReader reader = new BufferedReader( new InputStreamReader( url
.openStream(), "UTF-8" ) ) ) {
            return reader.lines()
                    .map( String::trim )
                    .filter( line -> !line.isEmpty() )
                    .filter( line -> !line.startsWith( "#" ) )
                    .collect( Collectors.toList() );
        }
```

```
        catch (IOException e) {
            throw new ValidationException( "Unable to read the ScriptEngineFactory resource
file", e );
        }
    }
}
```

and then declared with:

```
Validator validator = Validation.byProvider( HibernateValidator.class )
        .configure()
        .scriptEvaluatorFactory(
                new OsgiScriptEvaluatorFactory( FrameworkUtil.getBundle( this.getClass() )
.getBundleContext() )
        )
        .buildValidatorFactory()
        .getValidator();
```

It is designed specifically for OSGi environments and allows you to pass the `BundleContext` which will be used to search for `ScriptEngineFactory` as a parameter.

As already mentioned, you can also use script engines that are not based on JSR 223.

For instance, to use the Spring Expression Language, you can define a `SpringELScriptEvaluatorFactory` as:

```java
package org.hibernate.validator.referenceguide.chapter09;

public class SpringELScriptEvaluatorFactory extends AbstractCachingScriptEvaluatorFactory {

    @Override
    public ScriptEvaluator createNewScriptEvaluator(String languageName) {
        if ( !"spring".equalsIgnoreCase( languageName ) ) {
            throw new IllegalStateException( "Only Spring EL is supported" );
        }

        return new SpringELScriptEvaluator();
    }

    private static class SpringELScriptEvaluator implements ScriptEvaluator {

        private final ExpressionParser expressionParser = new SpelExpressionParser();

        @Override
        public Object evaluate(String script, Map<String, Object> bindings) throws
ScriptEvaluationException {
            try {
                Expression expression = expressionParser.parseExpression( script );
                EvaluationContext context = new StandardEvaluationContext( bindings.values
().iterator().next() );
                for ( Entry<String, Object> binding : bindings.entrySet() ) {
                    context.setVariable( binding.getKey(), binding.getValue() );
                }
                return expression.getValue( context );
            }
            catch (ParseException | EvaluationException e) {
                throw new ScriptEvaluationException( "Unable to evaluate SpEL script", e );
            }
        }
    }
}
```

This factory allows to use Spring Expression Language in `ScriptAssert` and
`ParameterScriptAssert` constraints:

```java
@ScriptAssert(script = "value > 0", lang = "spring")
public class Foo {

    private final int value;

    private Foo(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

9.3. 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 9.25, "Configuring a `Validator` instance via `usingContext()`" shows how this can be achieved by calling `ValidatorFactory#usingContext()`.

Example 9.25: Configuring a `Validator` *instance via* `usingContext()`

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

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

Chapter 10. Using constraint metadata

The Jakarta 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 `jakarta.validation.metadata`.

The examples presented in this chapter are based on the classes and constraint declarations shown in Example 10.1, "Example classes".

Example 10.1: Example classes

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

public class Person {

    public interface Basic {
    }

    @NotNull
    private String name;

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

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

public interface Vehicle {

    public interface Basic {
    }

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

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

@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...
}
```

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

public class Library {

    @NotNull
    private String name;

    private List<@NotNull @Valid Book> books;

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

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

public class Book {

    @NotEmpty
    private String title;

    @NotEmpty
    private String author;

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

10.1. BeanDescriptor

The entry point into the metadata API is the method `Validator#getConstraintsForClass()`, which returns an instance of the `BeanDescriptor` 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 10.2, "Using `BeanDescriptor`" demonstrates how to retrieve a `BeanDescriptor` for the `Car` class and how to use this descriptor in form of assertions.

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

Example 10.2: Using `BeanDescriptor`

```java
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 10.4, "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 10.7, "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 2.1.5, "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).

10.2. `PropertyDescriptor`

The interface `PropertyDescriptor` 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 10.3, "Using `PropertyDescriptor`" shows how to use the `PropertyDescriptor` interface.

```
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 `getConstraintDescriptors()`, 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 10.6, "`GroupConversionDescriptor`" for more details on `GroupConversionDescriptor`.

10.3. `MethodDescriptor` and `ConstructorDescriptor`

Constrained methods and constructors are represented by the interfaces `MethodDescriptor` `ConstructorDescriptor`, respectively. Example 10.4, "Using `MethodDescriptor` and `ConstructorDescriptor`" demonstrates how to work with these descriptors.

Example 10.4: 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( "speed", 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 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 the 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 "ParameterNameProvider") via `getName()` and `getIndex()`.

> 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").getReturnValueDescriptor()`).

10.4. `ElementDescriptor`

The `ElementDescriptor` 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 10.5, "Using `ElementDescriptor methods`" shows how these methods are used.

Example 10.5: 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 10.6, "Usage of `ConstraintFinder`" shows how to retrieve a `ConstraintFinder` instance via `findConstraints()` and use the API to query for constraint metadata.

Example 10.6: Usage of `ConstraintFinder`

```java
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.

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

10.5. `ContainerDescriptor` and `ContainerElementTypeDescriptor`

The `ContainerDescriptor` interface is the common interface for all the elements that support container element constraints and cascading validation (`PropertyDescriptor`, `ParameterDescriptor`, `ReturnValueDescriptor`).

It has a single method `getConstrainedContainerElementTypes()` that returns a set of `ContainerElementTypeDescriptor`.

`ContainerElementTypeDescriptor` extends `ContainerDescriptor` to support nested container element constraints.

`ContainerElementTypeDescriptor` contains the information about the container, the constraints and the cascading validation.

Example 10.7, "Using `ContainerElementTypeDescriptor`" shows how to use `getConstrainedContainerElementTypes()` to retrieve the set of `ContainerElementTypeDescriptor`.

```java
PropertyDescriptor booksDescriptor = libraryDescriptor.getConstraintsForProperty(
        "books"
);

Set<ContainerElementTypeDescriptor> booksContainerElementTypeDescriptors =
        booksDescriptor.getConstrainedContainerElementTypes();
ContainerElementTypeDescriptor booksContainerElementTypeDescriptor =
        booksContainerElementTypeDescriptors.iterator().next();

assertTrue( booksContainerElementTypeDescriptor.hasConstraints() );
assertTrue( booksContainerElementTypeDescriptor.isCascaded() );
assertEquals(
        0,
        booksContainerElementTypeDescriptor.getTypeArgumentIndex().intValue()
);
assertEquals(
        List.class,
        booksContainerElementTypeDescriptor.getContainerClass()
);

Set<ConstraintDescriptor<?>> constraintDescriptors =
        booksContainerElementTypeDescriptor.getConstraintDescriptors();
ConstraintDescriptor<?> constraintDescriptor =
        constraintDescriptors.iterator().next();

assertEquals(
        NotNull.class,
        constraintDescriptor.getAnnotation().annotationType()
);
```

10.6. `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` for each configured conversion, allowing to retrieve source and target groups of the conversion. Example 10.8, "Using `GroupConversionDescriptor`" shows an example.

Example 10.8: Using `GroupConversionDescriptor`

```java
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() );
```

10.7. `ConstraintDescriptor`

Last but not least, the `ConstraintDescriptor` 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 10.9, "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 10.9: Using `ConstraintDescriptor`

```java
//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()
);
```

Chapter 11. 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.

11.1. ORM integration

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

> When lazy loaded associations are supposed to be validated it is recommended to place the constraint on the getter of the association. Hibernate ORM 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.

11.1.1. Database schema-level validation

Out of the box, Hibernate ORM 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 ORM.

If, for some reason, the feature needs to be disabled, set `hibernate.validator.apply_to_ddl` to `false`. See also Section 2.3.1, "Jakarta Bean Validation constraints" and Section 2.3.2, "Additional 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.

11.1.2. Hibernate ORM event-based validation

Hibernate Validator has a built-in Hibernate event listener - `org.hibernate.cfg.beanvalidation.BeanValidationEventListener` - which is part of Hibernate ORM. 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 ORM. Pre deletion events will per default not trigger a validation. You can configure the groups to be validated per event type using the properties `jakarta.persistence.validation.group.pre-persist`, `jakarta.persistence.validation.group.pre-update` and `jakarta.persistence.validation.group.pre-remove`. The values of these properties are the

comma-separated fully specified class names of the groups to validate. Example 11.1, "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 `ConstraintViolation` instances describing each failure.

If Hibernate Validator is present in the classpath, Hibernate ORM will use it transparently. To avoid validation even though Hibernate Validator is in the classpath, set `jakarta.persistence.validation.mode` to none.

> ℹ️ 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 ORM, use the following configuration in *hibernate.cfg.xml*:

Example 11.1: Manual configuration of `BeanValidationEvenListener`

```xml
<hibernate-configuration>
    <session-factory>
        ...
        <property name="jakarta.persistence.validation.group.pre-persist">
            jakarta.validation.groups.Default
        </property>
        <property name="jakarta.persistence.validation.group.pre-update">
            jakarta.validation.groups.Default
        </property>
        <property name="jakarta.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>
```

11.1.3. JPA

If you are using JPA 2 and Hibernate Validator is in the classpath, the JPA2 specification requires that Jakarta Bean Validation gets enabled. The properties `jakarta.persistence.validation.group.pre-persist`, `jakarta.persistence.validation.group.pre-update` and

`jakarta.persistence.validation.group.pre-remove` as described in Section 11.1.2, "Hibernate ORM 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` or `NONE`. The default is `AUTO`.

11.2. JSF & Seam

When working with JSF2 or JBoss Seam and Hibernate Validator (Jakarta Bean Validation) is present in the runtime environment, validation is triggered for every field in the application. Example 11.2, "Usage of Jakarta 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 separated list of validation groups. The default is `jakarta.validation.groups.Default`. For more information refer to the Seam documentation or the JSF 2 specification.

Example 11.2: Usage of Jakarta Bean Validation within JSF2

```
<h:form>

  <f:validateBean validationGroups="jakarta.validation.groups.Default">

    <h:inputText value=#{model.property}/>
    <h:selectOneRadio value=#{model.radioProperty}> ... </h:selectOneRadio>
    <!-- other input components here -->

  </f:validateBean>

</h:form>
```

The integration between JSF 2 and Jakarta Bean Validation is described in the "Jakarta Bean Validation Integration" chapter of JSR-314. It is interesting to know that JSF 2 implements a custom `MessageInterpolator` to ensure proper localization. To encourage the use of the Jakarta 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 Jakarta Bean Validation message and `{1}` is replaced with the JSF component label):

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

The default is:

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

11.3. CDI

As of version 1.1, Bean Validation (and therefore Jakarta Bean Validation) is integrated with CDI (Contexts and Dependency Injection for Jakarta 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, parameter name providers, clock providers and value extractors.

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 Jakarta 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".

11.3.1. 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 `@jakarta.inject.Inject` as shown in Example 11.3, "Retrieving validator factory and validator via `@Inject`".

Example 11.3: Retrieving validator factory and validator via `@Inject`

```
package org.hibernate.validator.referenceguide.chapter11.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 Jakarta Bean Validation XML descriptors as discussed in Chapter 8, *Configuring via XML*.

If you are working with several Jakarta 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 11.4, "Using the

@HibernateValidator qualifier annotation".

Example 11.4: Using the @HibernateValidator qualifier annotation

```
package org.hibernate.validator.referenceguide.chapter11.cdi.validator.qualifier;

@ApplicationScoped
public class RentalStation {

    @Inject
    @HibernateValidator
    private ValidatorFactory validatorFactory;

    @Inject
    @HibernateValidator
    private Validator validator;

    //...
}
```

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 9.1, "Retrieving ValidatorFactory and Validator").

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

Example 11.5, "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 11.5: Constraint validator with injected bean

```java
package org.hibernate.validator.referenceguide.chapter11.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 );
    }
}
```

11.3.2. Method validation

The method interception facilities of CDI allow for a very tight integration with Jakarta 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.

The interceptor `org.hibernate.validator.cdi.internal.interceptor.ValidationInterceptor` is registered by `org.hibernate.validator.cdi.internal.ValidationExtension`. This happens implicitly within a Jakarta EE runtime environment or explicitly by adding the *hibernate-validator-cdi* artifact - see Section 1.1.2, "CDI"

You can see an example in Example 11.6, "CDI managed beans with method-level constraints".

Example 11.6: CDI managed beans with method-level constraints

```java
package org.hibernate.validator.referenceguide.chapter11.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) {
        //...
        return null;
    }

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

```java
package org.hibernate.validator.referenceguide.chapter11.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`.

11.3.2.1. Validated executable types

Jakarta 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 11.6, "CDI managed beans with method-level constraints" does not get 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 8.1, "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 11.7, "Using `@ValidateOnExecution`" shows how to use the `@ValidateOnExecution` annotation:

Example 11.7: Using `@ValidateOnExecution`

```java
package
org.hibernate.validator.referenceguide.chapter11.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) {
        //...
        return null;
    }

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

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`.

> Executable validation can be turned off globally by specifying `<executable-validation enabled="false"/>` in *META-INF/validation.xml*. In this case, all the `@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 11.8, "Using ExecutableType.IMPLICIT", which makes sure that all required metadata is discovered and the validation interceptor kicks in when the methods on `ExpressRentalStation` are invoked.

Example 11.8: Using `ExecutableType.IMPLICIT`

```
package org.hibernate.validator.referenceguide.chapter11.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);

    @NotNull
    List<Car> getAvailableCars();
}
```

```
package org.hibernate.validator.referenceguide.chapter11.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) {
        //...
        return null;
    }

    @Override
    public List<Car> getAvailableCars() {
        //...
        return null;
    }
}
```

11.4. Jakarta EE

When your application runs on a Jakarta 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 11.9, "Retrieving `Validator` and `ValidatorFactory` via `@Resource` injection".

Example 11.9: Retrieving `Validator` *and* `ValidatorFactory` *via* `@Resource` *injection*

```
package org.hibernate.validator.referenceguide.chapter11.javaee;

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 8, *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.

11.5. JavaFX

Hibernate Validator also provides support for the unwrapping of JavaFX properties. If JavaFX is present on the classpath, `ValueExtractor`s for JavaFX properties are automatically registered. See Section 7.4, "JavaFX value extractors" for examples and further discussion.

Chapter 12. 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 Jakarta Bean Validation specification. This includes the fail fast mode, the API for programmatic constraint configuration and the boolean composition of constraints.

New APIs or SPIs are tagged with the `org.hibernate.validator.Incubating` annotation as long as they are under development. This means that such elements (e.g. packages, types, methods, constants etc.) may be incompatibly altered - or removed - in subsequent releases. Usage of incubating API/SPI members is encouraged (so the development team can get feedback on these new features) but you should be prepared for updating code which is using them as needed when upgrading to a new version of Hibernate Validator.

> Using the features described in the following sections may result in application code which is not portable between Jakarta Bean Validation providers.

12.1. Public API

Let's start, however, with a look at the public API of Hibernate Validator. Below you can find a list of all packages belonging to this API and their purpose. Note that when a package is part of the public API this is not necessarily true for its sub-packages.

`org.hibernate.validator`

Classes used by the Jakarta Bean Validation bootstrap mechanism (eg. validation provider, configuration class); for more details see Chapter 9, *Bootstrapping*.

`org.hibernate.validator.cfg`, `org.hibernate.validator.cfg.context`, `org.hibernate.validator.cfg.defs`, `org.hibernate.validator.spi.cfg`

Hibernate Validator's fluent API for constraint declaration; in `org.hibernate.validator.cfg` you will find the `ConstraintMapping` interface, in `org.hibernate.validator.cfg.defs` all constraint definitions and in `org.hibernate.validator.spi.cfg` a callback for using the API for configuring the default validator factory. Refer to Section 12.4, "Programmatic constraint definition and declaration" for the details.

`org.hibernate.validator.constraints`, `org.hibernate.validator.constraints.br`, `org.hibernate.validator.constraints.pl`

Some useful custom constraints provided by Hibernate Validator in addition to the built-in constraints defined by the Jakarta 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 12.13.1, "`HibernateConstraintValidatorContext`" describes how to make use of that feature.

`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 5.4, "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 12.14, "Paranamer based `ParameterNameProvider`".

`org.hibernate.validator.propertypath`

Extensions to the `jakarta.validation.Path` API, see Section 12.7, "Extensions of the Path API".

`org.hibernate.validator.spi.constraintdefinition`

An SPI for registering additional constraint validators programmatically, see Section 12.15, "Providing constraint definitions".

`org.hibernate.validator.spi.messageinterpolation`

An SPI that can be used to tweak the resolution of the locale when interpolating the constraint violation messages. See Section 12.12, "Customizing the locale resolution".

`org.hibernate.validator.spi.nodenameprovider`

An SPI that can be used to alter how the names of properties will be resolved when the property path is constructed. See Section 12.18, "Customizing the property name resolution for constraint violations".

> 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.

12.2. 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 12.1, "Using the fail fast validation mode" shows how to bootstrap and use a fail fast enabled validator.

Example 12.1: Using the fail fast validation mode

```java
package org.hibernate.validator.referenceguide.chapter12.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...
}
```

```java
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.

> ℹ️ 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 5.3, "Defining group sequences".

Refer to Section 9.2.8, "Provider-specific settings" to learn about the different ways of enabling the fail fast mode when bootstrapping a validator.

12.3. Relaxation of requirements for method validation in class hierarchies

The Jakarta Bean Validation specification defines a set of preconditions which apply when defining constraints on methods within class hierarchies. These preconditions are defined in section 5.6.5 of the Jakarta Bean Validation specification. See also Section 3.1.4, "Method constraints in inheritance hierarchies" in this guide.

As per specification, a Jakarta Bean Validation provider is allowed to relax these preconditions. With Hibernate Validator you can do this in one of two ways.

First you can use the configuration properties *hibernate.validator.allow_parameter_constraint_override*, *hibernate.validator.allow_multiple_cascaded_validation_on_result* and *hibernate.validator.allow_parallel_method_parameter_constraint* in *validation.xml*. See example Example 12.2, "Configuring method validation behaviour in class hierarchies via properties".

Example 12.2: Configuring method validation behaviour in class hierarchies via properties

```xml
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
        xmlns="https://jakarta.ee/xml/ns/validation/configuration"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/configuration
https://jakarta.ee/xml/ns/validation/validation-configuration-3.0.xsd"
        version="3.0">
    <default-provider>org.hibernate.validator.HibernateValidator</default-provider>

    <property name="hibernate.validator.allow_parameter_constraint_override">
true</property>
    <property name="hibernate.validator.allow_multiple_cascaded_validation_on_result">
true</property>
    <property name="hibernate.validator.allow_parallel_method_parameter_constraint">
true</property>
</validation-config>
```

Alternatively these settings can be applied during programmatic bootstrapping.

Example 12.3: Configuring method validation behaviour in class hierarchies

```
HibernateValidatorConfiguration configuration = Validation.byProvider( HibernateValidator
.class ).configure();

configuration.allowMultipleCascadedValidationOnReturnValues( true )
        .allowOverridingMethodAlterParameterConstraint( true )
        .allowParallelMethodsDefineParameterConstraints( true );
```

By default, all of these properties are false, implementing the default behavior as defined in the Jakarta Bean Validation specification.

> ⚠️ Changing the default behaviour for method validation will result in non specification-conforming and non portable application. Make sure to understand what you are doing and that your use case really requires changes to the default behaviour.

12.4. Programmatic constraint definition and declaration

As per the Jakarta Bean Validation specification, you can define and 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 12.4, "Programmatic constraint declaration".

Example 12.4: Programmatic constraint declaration

```
HibernateValidatorConfiguration configuration = Validation
        .byProvider( HibernateValidator.class )
        .configure();

ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .field( "manufacturer" )
            .constraint( new NotNullDef() )
        .field( "licensePlate" )
            .ignoreAnnotations( true )
            .constraint( new NotNullDef() )
            .constraint( new SizeDef().min( 2 ).max( 14 ) )
    .type( RentalCar.class )
        .getter( "rentalStation" )
            .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.

> ℹ️ 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.

> ℹ️ 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 12.5, "Programmatic declaration of a custom constraint".

Example 12.5: Programmatic declaration of a custom constraint

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .field( "licensePlate" )
            .constraint( new GenericConstraintDef<>( CheckCase.class )
                .param( "value", CaseMode.UPPER )
            );
```

Container element constraints are supported by the programmatic API, using `containerElementType()`.

Example 12.6, "Programmatic declaration of a nested container element constraint" show an example where constraints are declared on nested container elements.

Example 12.6: Programmatic declaration of a nested container element constraint

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .field( "manufacturer" )
            .constraint( new NotNullDef() )
        .field( "licensePlate" )
            .ignoreAnnotations( true )
            .constraint( new NotNullDef() )
            .constraint( new SizeDef().min( 2 ).max( 14 ) )
        .field( "partManufacturers" )
            .containerElementType( 0 )
                .constraint( new NotNullDef() )
            .containerElementType( 1, 0 )
                .constraint( new NotNullDef() )
    .type( RentalCar.class )
        .getter( "rentalStation" )
            .constraint( new NotNullDef() );
```

As demonstrated, the parameters passed to `containerElementType()` are the path of type argument indexes used to obtain the desired nested container element type.

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 12.7, "Marking a property for cascaded validation".

Example 12.7: Marking a property for cascaded validation

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
    .type( Car.class )
        .field( "driver" )
            .constraint( new NotNullDef() )
            .valid()
            .convertGroup( Default.class ).to( PersonDefault.class )
        .field( "partManufacturers" )
            .containerElementType( 0 )
                .valid()
            .containerElementType( 1, 0 )
                .valid()
    .type( Person.class )
        .field( "name" )
            .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 12.8, "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.

As shown in the example, `valid()` can be also invoked on a container element type.

Example 12.8: 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.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 12.9: 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 );
```

12.5. Applying programmatic constraint declarations to the default validator factory

If you are not bootstrapping a validator factory manually but work with the default factory as configured via *META-INF/validation.xml* (see Chapter 8, *Configuring via XML*), you can add one or more constraint mappings by creating one or several constraint mapping contributors. To do so, implement the `ConstraintMappingContributor` contract:

Example 12.10: Custom `ConstraintMappingContributor` implementation

```
package org.hibernate.validator.referenceguide.chapter12.constraintapi;

public class MyConstraintMappingContributor implements ConstraintMappingContributor {

    @Override
    public void createConstraintMappings(ConstraintMappingBuilder builder) {
        builder.addConstraintMapping()
            .type( Marathon.class )
                .getter( "name" )
                    .constraint( new NotNullDef() )
                .field( "numberOfHelpers" )
                    .constraint( new MinDef().value( 1 ) );

        builder.addConstraintMapping()
            .type( Runner.class )
                .field( "paidEntryFee" )
                    .constraint( new AssertTrueDef() );
    }
}
```

You then need to specify the fully-qualified class name of the contributor implementation in *META-INF/validation.xml*, using the property key `hibernate.validator.constraint_mapping_contributors`. You can specify several contributors by separating them with a comma.

12.6. Advanced constraint composition features

12.6.1. Validation target specification for purely composed constraints

In case you specify a purely composed constraint - i.e. a constraint which has no validator itself but is solely made up from other, composing constraints - on a method declaration, the validation engine cannot determine whether that constraint is to be applied as a return value constraint or as a cross-parameter constraint.

Hibernate Validator allows to resolve such ambiguities by specifying the @SupportedValidationTarget annotation on the declaration of the composed constraint type as shown in Example 12.11, "Specifying the validation target of a purely composed constraint". The @ValidInvoiceAmount does not declare any validator, but it is solely composed by the @Min and @NotNull constraints. The @SupportedValidationTarget ensures that the constraint is applied to the method return value when given on a method declaration.

Example 12.11: Specifying the validation target of a purely composed constraint

```java
package org.hibernate.validator.referenceguide.chapter12.purelycomposed;

@Min(value = 0)
@NotNull
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
@SupportedValidationTarget(ValidationTarget.ANNOTATED_ELEMENT)
@ReportAsSingleViolation
public @interface ValidInvoiceAmount {

    String message() default
"{org.hibernate.validator.referenceguide.chapter11.purelycomposed."
            + "ValidInvoiceAmount.message}";

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

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

    @OverridesAttribute(constraint = Min.class, name = "value")
    long value();
}
```

12.6.2. Boolean composition of constraints

Jakarta 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 to obtain 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 12.12, "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 12.12: OR composition of constraints

```
package org.hibernate.validator.referenceguide.chapter12.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 { };
}
```

> 💡 Using *ALL_FALSE* as composition type implicitly enforces that only a single violation will get reported in case validation of the constraint composition fails.

12.7. Extensions of the Path API

Hibernate Validator provides an extension to the `jakarta.validation.Path` API. For nodes of `ElementKind.PROPERTY` and `ElementKind.CONTAINER_ELEMENT` it allows to obtain the value of the represented property. To do so, narrow down a given node to the type `org.hibernate.validator.path.PropertyNode` or `org.hibernate.validator.path.ContainerElementNode` respectively using `Node#as()`, as shown in the following example:

Example 12.13: Getting the value from property nodes

```java
Building building = new Building();

// Assume the name of the person violates a @Size constraint
Person bob = new Person( "Bob" );
Apartment bobsApartment = new Apartment( bob );
building.getApartments().add( bobsApartment );

Set<ConstraintViolation<Building>> constraintViolations = validator.validate( building );

Path path = constraintViolations.iterator().next().getPropertyPath();
Iterator<Path.Node> nodeIterator = path.iterator();

Path.Node node = nodeIterator.next();
assertEquals( node.getName(), "apartments" );
assertSame( node.as( PropertyNode.class ).getValue(), bobsApartment );

node = nodeIterator.next();
assertEquals( node.getName(), "resident" );
assertSame( node.as( PropertyNode.class ).getValue(), bob );

node = nodeIterator.next();
assertEquals( node.getName(), "name" );
assertEquals( node.as( PropertyNode.class ).getValue(), "Bob" );
```

This is also very useful to obtain the element of `Set` properties on the property path (e.g. `apartments` in the example) which otherwise could not be identified (unlike for `Map` and `List`, there is no key nor index in this case).

12.8. Dynamic payload as part of `ConstraintViolation`

In some cases automatic processing of violations can be aided, if the constraint violation provides additional data - a so called dynamic payload. This dynamic payload could for example contain hints to the user on how to resolve the violation.

Dynamic payloads can be set in custom constraints using `HibernateConstraintValidatorContext`. This is shown in example Example 12.14, "ConstraintValidator implementation setting a dynamic payload" where the `jakarta.validation.ConstraintValidatorContext` is unwrapped to `HibernateConstraintValidatorContext` in order to call `withDynamicPayload`.

Example 12.14: `ConstraintValidator` *implementation setting a dynamic payload*

```
package org.hibernate.validator.referenceguide.chapter12.dynamicpayload;

import static org.hibernate.validator.internal.util.CollectionHelper.newHashMap;

public class ValidPassengerCountValidator implements ConstraintValidator
<ValidPassengerCount, Car> {

    private static final Map<Integer, String> suggestedCars = newHashMap();

    static {
        suggestedCars.put( 2, "Chevrolet Corvette" );
        suggestedCars.put( 3, "Toyota Volta" );
        suggestedCars.put( 4, "Maserati GranCabrio" );
        suggestedCars.put( 5, " Mercedes-Benz E-Class" );
    }

    @Override
    public void initialize(ValidPassengerCount constraintAnnotation) {
    }

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

        int passengerCount = car.getPassengers().size();
        if ( car.getSeatCount() >= passengerCount ) {
            return true;
        }
        else {

            if ( suggestedCars.containsKey( passengerCount ) ) {
                HibernateConstraintValidatorContext hibernateContext = context.unwrap(
                        HibernateConstraintValidatorContext.class
                );
                hibernateContext.withDynamicPayload( suggestedCars.get( passengerCount ) );
            }
            return false;
        }
    }
}
```

On the constraint violation processing side, a `jakarta.validation.ConstraintViolation` can then in turn be unwrapped to `HibernateConstraintViolation` in order to retrieve the dynamic payload for further processing.

Example 12.15: Retrieval of a `ConstraintViolation`*'s dynamic payload*

```
Car car = new Car( 2 );
car.addPassenger( new Person() );
car.addPassenger( new Person() );
car.addPassenger( new Person() );
Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );

assertEquals( 1, constraintViolations.size() );

ConstraintViolation<Car> constraintViolation = constraintViolations.iterator().next();
@SuppressWarnings("unchecked")
HibernateConstraintViolation<Car> hibernateConstraintViolation = constraintViolation.
unwrap(
        HibernateConstraintViolation.class
);
String suggestedCar = hibernateConstraintViolation.getDynamicPayload( String.class );
assertEquals( "Toyota Volta", suggestedCar );
```

12.9. Enabling Expression Language features

Hibernate Validator restricts the Expression Language features exposed by default.

For this purpose, we define several feature levels in `ExpressionLanguageFeatureLevel`:

- `NONE`: Expression Language interpolation is fully disabled.

- `VARIABLES`: Allow interpolation of the variables injected via `addExpressionVariable()`, resources bundles and usage of the `formatter` object.

- `BEAN_PROPERTIES`: Allow everything `VARIABLES` allows plus the interpolation of bean properties.

- `BEAN_METHODS`: Also allow execution of bean methods. This can lead to serious security issues, including arbitrary code execution if not carefully handled.

Depending on the context, the features we expose are different:

- For constraints, the default level is `BEAN_PROPERTIES`. For all the built-in constraint messages to be correctly interpolated, you need at least the `VARIABLES` level.

- For custom violations, created via the `ConstraintValidatorContext`, Expression Language is disabled by default. You can enable it for specific custom violations and, when enabled, it will default to `VARIABLES`.

Hibernate Validator provides ways to override these defaults when boostrapping the `ValidatorFactory`.

To change the Expression Language feature level for constraints, use the following:

```
ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
        .configure()
        .constraintExpressionLanguageFeatureLevel( ExpressionLanguageFeatureLevel.VARIABLES )
        .buildValidatorFactory();
```

To change the Expression Language feature level for custom violations, use the following:

```
ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
        .configure()
        .customViolationExpressionLanguageFeatureLevel( ExpressionLanguageFeatureLevel
.VARIABLES )
        .buildValidatorFactory();
```

> Doing this will automatically enable Expression Language for all the custom violations in your application.
>
> It should only be used for compatibility and to ease the migration from older Hibernate Validator versions.

These levels can also be defined using the following properties:

- `hibernate.validator.constraint_expression_language_feature_level`

- `hibernate.validator.custom_violation_expression_language_feature_level`

Accepted values for these properties are: `none`, `variables`, `bean-properties` and `bean-methods`.

12.10. `ParameterMessageInterpolator`

Hibernate Validator requires per default an implementation of the Unified EL (see Section 1.1.1, "Unified EL") to be available. This is needed to allow the interpolation of constraint error messages using EL expressions as defined by the Jakarta Bean Validation specification.

For environments where you cannot or do not want to provide an EL implementation, Hibernate Validator offers a non EL based message interpolator - `org.hibernate.validator.messageinterpolation.ParameterMessageInterpolator`.

Refer to Section 4.2, "Custom message interpolation" to see how to plug in custom message interpolator implementations.

> Constraint messages containing EL expressions will be returned un-interpolated by `org.hibernate.validator.messageinterpolation.ParameterMessageInterpolator`. This also affects built-in default constraint messages which use EL expressions. At the moment, `DecimalMin` and `DecimalMax` are affected.

12.11. `ResourceBundleLocator`

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.

12.12. Customizing the locale resolution

> ⚠️ These contracts are marked as `@Incubating` so they might be subject to change in the future.

Hibernate Validator provides several extension points to build a custom locale resolution strategy. The resolved locale is used when interpolating the constraint violation messages.

The default behavior of Hibernate Validator is to always use the system default locale (as obtained via `Locale.getDefault()`). This might not be the desired behavior if, for example, you usually set your system locale to `en-US` but want your application to provide messages in French.

The following example shows how to set the Hibernate Validator default locale to `fr-FR`:

Example 12.16: Configure the default locale

```
Validator validator = Validation.byProvider( HibernateValidator.class )
        .configure()
        .defaultLocale( Locale.FRANCE )
        .buildValidatorFactory()
        .getValidator();

Set<ConstraintViolation<Bean>> violations = validator.validate( new Bean() );
assertEquals( "doit avoir la valeur vrai", violations.iterator().next().getMessage() );
```

While this is already a nice improvement, in a fully internationalized application, this is not sufficient: you need Hibernate Validator to select the locale depending on the user context.

Hibernate Validator provides the `org.hibernate.validator.spi.messageinterpolation.LocaleResolver` SPI which allows to fine-tune the resolution of the locale. Typically, in a JAX-RS environment, you can resolve the locale to use from the `Accept-Language` HTTP header.

In the following example, we use a hardcoded value but, for instance, in the case of a RESTEasy application, you could extract the header from the `ResteasyContext`.

Example 12.17: Fine tune the locale used to interpolate the messages via a `LocaleResolver`

```
LocaleResolver localeResolver = new LocaleResolver() {

    @Override
    public Locale resolve(LocaleResolverContext context) {
        // get the locales supported by the client from the Accept-Language header
        String acceptLanguageHeader = "it-IT;q=0.9,en-US;q=0.7";

        List<LanguageRange> acceptedLanguages = LanguageRange.parse( acceptLanguageHeader
);
        List<Locale> resolvedLocales = Locale.filter( acceptedLanguages, context
.getSupportedLocales() );

        if ( resolvedLocales.size() > 0 ) {
            return resolvedLocales.get( 0 );
        }

        return context.getDefaultLocale();
    }
};

Validator validator = Validation.byProvider( HibernateValidator.class )
        .configure()
        .defaultLocale( Locale.FRANCE )
        .locales( Locale.FRANCE, Locale.ITALY, Locale.US )
        .localeResolver( localeResolver )
        .buildValidatorFactory()
        .getValidator();

Set<ConstraintViolation<Bean>> violations = validator.validate( new Bean() );
assertEquals( "deve essere true", violations.iterator().next().getMessage() );
```

> ℹ️ When using the `LocaleResolver`, you have to define the list of supported locales via the `locales()` method.

12.13. Custom contexts

The Jakarta Bean Validation specification offers at several points in its API the possibility to unwrap a given interface to an 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.

12.13.1. `HibernateConstraintValidatorContext`

`HibernateConstraintValidatorContext` is a subtype of `ConstraintValidatorContext` which allows you to:

- enable Expression Language interpolation for a particular custom violation - see below

- set arbitrary parameters for interpolation via the Expression Language message interpolation

facility using
`HibernateConstraintValidatorContext#addExpressionVariable(String, Object)`
or `HibernateConstraintValidatorContext#addMessageParameter(String, Object)`.

Example 152. Custom `@Future` validator injecting an expression variable

```java
package org.hibernate.validator.referenceguide.chapter12.context;

public class MyFutureValidator implements ConstraintValidator<Future, Instant> {

    @Override
    public void initialize(Future constraintAnnotation) {
    }

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

        HibernateConstraintValidatorContext hibernateContext = context.unwrap(
                HibernateConstraintValidatorContext.class
        );

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

        if ( !value.isAfter( now ) ) {
            hibernateContext.disableDefaultConstraintViolation();
            hibernateContext
                    .addExpressionVariable( "now", now )
                    .buildConstraintViolationWithTemplate( "Must be after ${now}" )
                    .addConstraintViolation();

            return false;
        }

        return true;
    }
}
```

Example 153. Custom @Future validator injecting a message parameter

```java
package org.hibernate.validator.referenceguide.chapter12.context;

public class MyFutureValidatorMessageParameter implements ConstraintValidator<Future,
Instant> {

    @Override
    public void initialize(Future constraintAnnotation) {
    }

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

        HibernateConstraintValidatorContext hibernateContext = context.unwrap(
                HibernateConstraintValidatorContext.class
        );

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

        if ( !value.isAfter( now ) ) {
            hibernateContext.disableDefaultConstraintViolation();
            hibernateContext
                    .addMessageParameter( "now", now )
                    .buildConstraintViolationWithTemplate( "Must be after {now}" )
                    .addConstraintViolation();

            return false;
        }

        return true;
    }
}
```

> ℹ️ Apart from the syntax, the main difference between message parameters and expression variables is that message parameters are simply interpolated whereas expression variables are interpreted using the Expression Language engine. In practice, use message parameters if you do not need the advanced features of an Expression Language.

> ℹ️ Note that the parameters specified via `addExpressionVariable(String, Object)` and `addMessageParameter(String, Object)` are global and apply to 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()`.

- set an arbitrary dynamic payload - see Section 12.8, "Dynamic payload as part of `ConstraintViolation`"

By default, Expression Language interpolation is **disabled** for custom violations, this to avoid arbitrary code execution or sensitive data leak if message templates are built from improperly escaped user input.

It is possible to enable Expression Language for a given custom violation by using `enableExpressionLanguage()` as shown in the example below:

```java
public class SafeValidator implements ConstraintValidator<ZipCode, String> {

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

        HibernateConstraintValidatorContext hibernateContext = context.unwrap(
                HibernateConstraintValidatorContext.class );
        hibernateContext.disableDefaultConstraintViolation();

        if ( isInvalid( value ) ) {
            hibernateContext
                    .addExpressionVariable( "validatedValue", value )
                    .buildConstraintViolationWithTemplate( "${validatedValue} is not a valid
 ZIP code" )
                    .enableExpressionLanguage()
                    .addConstraintViolation();

            return false;
        }

        return true;
    }

    private boolean isInvalid(String value) {
        // ...
        return false;
    }
}
```

In this case, the message template will be interpolated by the Expression Language engine.

By default, only variables interpolation is enabled when enabling Expression Language.

You can enable more features by using `HibernateConstraintViolationBuilder#enableExpressionLanguage(ExpressionLanguageFeatureLevel level)`.

We define several levels of features for Expression Language interpolation:

- `NONE`: Expression Language interpolation is fully disabled - this is the default for custom violations.

- `VARIABLES`: Allow interpolation of the variables injected via `addExpressionVariable()`, resources bundles and usage of the `formatter` object.

- `BEAN_PROPERTIES`: Allow everything `VARIABLES` allows plus the interpolation of bean properties.

- BEAN_METHODS: Also allow execution of bean methods. This can lead to serious security issues, including arbitrary code execution if not carefully handled.

Using `addExpressionVariable()` is the only safe way to inject a variable into an expression and it's especially important if you use the `BEAN_PROPERTIES` or `BEAN_METHODS` feature levels.

If you inject user input by simply concatenating the user input in the message, you will allow potential arbitrary code execution and sensitive data leak: if the user input contains valid expressions, they will be executed by the Expression Language engine.

Here is an example of something you should **ABSOLUTELY NOT** do:

```java
public class UnsafeValidator implements ConstraintValidator<ZipCode,
String> {

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

        context.disableDefaultConstraintViolation();

        HibernateConstraintValidatorContext hibernateContext = context
.unwrap(
                HibernateConstraintValidatorContext.class );
        hibernateContext.disableDefaultConstraintViolation();

        if ( isInvalid( value ) ) {
            hibernateContext
                    // THIS IS UNSAFE, DO NOT COPY THIS EXAMPLE
                    .buildConstraintViolationWithTemplate( value + " is not
 a valid ZIP code" )
                    .enableExpressionLanguage()
                    .addConstraintViolation();

            return false;
        }

        return true;
    }

    private boolean isInvalid(String value) {
        // ...
        return false;
    }
}
```

In the example above, if `value`, which might be user input, contains a valid expression, it will be interpolated by the Expression Language engine, potentially leading to unsafe behaviors.

12.13.2. `HibernateMessageInterpolatorContext`

Hibernate Validator also offers a custom extension of `MessageInterpolatorContext`, namely `HibernateMessageInterpolatorContext` (see Example 12.18, "HibernateMessageInterpolatorContext"). This subtype was introduced to allow a better integration of Hibernate Validator into Glassfish. The root bean type was in this case needed to determine the right class loader for the message resource bundle. If you have any other use cases, let us know.

Example 12.18: `HibernateMessageInterpolatorContext`

```
public interface HibernateMessageInterpolatorContext extends MessageInterpolator.Context {

    /**
     * Returns the currently validated root bean type.
     *
     * @return The currently validated root bean type.
     */
    Class<?> getRootBeanType();

    /**
     * @return the message parameters added to this context for interpolation
     *
     * @since 5.4.1
     */
    Map<String, Object> getMessageParameters();

    /**
     * @return the expression variables added to this context for EL interpolation
     *
     * @since 5.4.1
     */
    Map<String, Object> getExpressionVariables();

    /**
     * @return the path to the validated constraint starting from the root bean
     *
     * @since 6.1
     */
    Path getPropertyPath();

    /**
     * @return the level of features enabled for the Expression Language engine
     *
     * @since 6.2
     */
    ExpressionLanguageFeatureLevel getExpressionLanguageFeatureLevel();
}
```

12.14. Paranamer based `ParameterNameProvider`

Hibernate Validator comes with a `ParameterNameProvider` implementation which leverages the Paranamer 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 9.10, "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.

> 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.

12.15. Providing constraint definitions

Jakarta Bean Validation allows to (re-)define constraint definitions via XML in its constraint mapping files. See Section 8.2, "Mapping constraints via `constraint-mappings`" for more information and Example 8.2, "Bean constraints configured via XML" for an example. While this approach is sufficient for many use cases, it has its shortcomings in others. Imagine for example a constraint library wanting to contribute constraint definitions for custom types. This library could provide a mapping file with their library, but this file still would need to be referenced by the user of the library. Luckily there are better ways.

> The following concepts are considered experimental at this time. Let us know whether you find them useful and whether they meet your needs.

12.15.1. Constraint definitions via `ServiceLoader`

Hibernate Validator allows to utilize Java's ServiceLoader mechanism to register additional constraint definitions. All you have to do is to add the file *jakarta.validation.ConstraintValidator* to *META-INF/services*. In this service file you list the fully qualified classnames of your constraint validator classes (one per line). Hibernate Validator will automatically infer the constraint types they apply to. See Constraint definition via service file for an example.

Example 12.19: META-INF/services/jakarta.validation.ConstraintValidator

```
# Assuming a custom constraint annotation @org.mycompany.CheckCase
org.mycompany.CheckCaseValidator
```

To contribute default messages for your custom constraints, place a file *ContributorValidationMessages.properties* and/or its locale-specific specializations at the root of your JAR. Hibernate Validator will consider the entries from all the bundles with this name found on the classpath in addition to those given in *ValidationMessages.properties*.

This mechanism is also helpful when creating large multi-module applications: instead of putting all the constraint messages into one single bundle, you can have one resource bundle per module containing only those messages of that module.

> We highly recommend the reading of this blog post by Marko Bekhta, guiding you step by step through the process of creating an independent JAR that contains your custom constraints and declares them via the `ServiceLoader`.

12.15.2. Adding constraint definitions programmatically

While the service loader approach works in many scenarios, but not in all (think for example OSGi where service files are not visible), there is yet another way of contributing constraint definitions. You can use the programmatic constraint declaration API - see Example 12.20, "Adding constraint definitions through the programmatic API".

Example 12.20: Adding constraint definitions through the programmatic API

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
        .constraintDefinition( ValidPassengerCount.class )
        .validatedBy( ValidPassengerCountValidator.class );
```

If your validator implementation is rather simple (i.e. no initialization from the annotation is needed, and `ConstraintValidatorContext` is not used), you also can use this alternative API to specify the constraint logic using a Lambda expression or method reference:

Example 12.21: Adding constraint definition with a Lambda expression

```
ConstraintMapping constraintMapping = configuration.createConstraintMapping();

constraintMapping
        .constraintDefinition( ValidPassengerCount.class )
            .validateType( Bus.class )
                .with( b -> b.getSeatCount() >= b.getPassengers().size() );
```

Instead of directly adding a constraint mapping to the configuration object, you may use a `ConstraintMappingContributor` as detailed in Section 12.5, "Applying programmatic constraint declarations to the default validator factory". This can be useful when configuring the default validator factory using *META-INF/validation.xml* (see Chapter 8, *Configuring via XML*).

One use case for registering constraint definitions through the programmatic API is the ability to specify an alternative constraint validator for the `@URL` constraint. Historically, Hibernate Validator's default constraint validator for this constraint uses the `java.net.URL` constructor to validate an URL. However, there is also a purely regular expression based version available which can be configured using a `ConstraintDefinitionContributor`:

Using the programmatic constraint declaration API to register a regular expression based constraint definition for `@URL`

```
ConstraintMapping constraintMapping = configuration.
createConstraintMapping();

constraintMapping
        .constraintDefinition( URL.class )
        .includeExistingValidators( false )
        .validatedBy( RegexpURLValidator.class );
```

12.16. Customizing class-loading

There are several cases in which Hibernate Validator needs to load resources or classes given by name:

- XML descriptors (*META-INF/validation.xml* as well as XML constraint mappings)

- classes specified by name in XML descriptors (e.g. custom message interpolators etc.)

- the *ValidationMessages* resource bundle

- the `ExpressionFactory` implementation used for expression based message interpolation

By default, Hibernate Validator tries to load these resources via the current thread context class loader. If that's not successful, Hibernate Validator's own class loader will be tried as a fallback.

For cases where this strategy is not appropriate (e.g. modularized environments such as OSGi), you may provide a specific class loader for loading these resources when bootstrapping the validator factory:

Example 12.22: Providing a class loader for loading external resources and classes

```
Validator validator = Validation.byProvider( HibernateValidator.class )
        .configure()
        .externalClassLoader( classLoader )
        .buildValidatorFactory()
        .getValidator();
```

In the case of OSGi, you could e.g. pass the loader of a class from the bundle bootstrapping Hibernate Validator or a custom class loader implementation which delegates to `Bundle#loadClass()` etc.

> Call `ValidatorFactory#close()` if a given validator factory instance is not needed any longer. Failure to do so may result in a class loader leak in cases where applications/bundles are re-deployed and a non-closed validator factory still is referenced by application code.

12.17. Customizing the getter property selection strategy

When a bean is validated by Hibernate Validator, its properties get validated. A property can either be a field or a getter. By default, Hibernate Validator respects the JavaBeans specification and considers a method as a getter as soon as one of the conditions below is true:

- the method name starts with `get`, it has a non-void return type and has no parameters;

- the method name starts with `is`, has a return type of `boolean` and has no parameters;

- the method name starts with `has`, has a return type of `boolean` and has no parameters (this rule is specific to Hibernate Validator and is not mandated by the JavaBeans specification)

While these rules are usually appropriate when following the classic JavaBeans convention, it might happen, especially with code generators, that the JavaBeans naming convention is not followed and that the getters' names are following a different convention.

In this case, the strategy for detecting getters should be redefined in order to fully validate the object.

A classic example of this requirement is when the classes follow a fluent naming convention, as illustrated in Example 12.23, "A class that uses non-standard getters".

Example 12.23: A class that uses non-standard getters

```
package org.hibernate.validator.referenceguide.chapter12.getterselectionstrategy;

public class User {

    private String firstName;
    private String lastName;
    private String email;

    // [...]

    @NotEmpty
    public String firstName() {
        return firstName;
    }

    @NotEmpty
    public String lastName() {
        return lastName;
    }

    @Email
    public String email() {
        return email;
    }
}
```

If such object gets validated, no validation will be performed on the getters as they are not detected by the standard strategy.

Example 12.24: Validating a class with non-standard getters using the default getter property selection strategy

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

User user = new User( "", "", "not an email" );

Set<ConstraintViolation<User>> constraintViolations = validator.validate( user );

// as User has non-standard getters no violations are triggered
assertEquals( 0, constraintViolations.size() );
```

To make Hibernate Validator treat such methods as properties, a custom `GetterPropertySelectionStrategy` should be configured. In this particular case, a possible implementation of the strategy would be:

Example 12.25: Custom `GetterPropertySelectionStrategy` *implementation*

```java
package org.hibernate.validator.referenceguide.chapter12.getterselectionstrategy;

public class FluentGetterPropertySelectionStrategy implements
GetterPropertySelectionStrategy {

    private final Set<String> methodNamesToIgnore;

    public FluentGetterPropertySelectionStrategy() {
        // we will ignore all the method names coming from Object
        this.methodNamesToIgnore = Arrays.stream( Object.class.getDeclaredMethods() )
                .map( Method::getName )
                .collect( Collectors.toSet() );
    }

    @Override
    public Optional<String> getProperty(ConstrainableExecutable executable) {
        if ( methodNamesToIgnore.contains( executable.getName() )
                || executable.getReturnType() == void.class
                || executable.getParameterTypes().length > 0 ) {
            return Optional.empty();
        }

        return Optional.of( executable.getName() );
    }

    @Override
    public Set<String> getGetterMethodNameCandidates(String propertyName) {
        // As method name == property name, there always is just one possible name for a
method
        return Collections.singleton( propertyName );
    }
}
```

There are multiple ways to configure Hibernate Validator to use this strategy. It can either be done programmatically (see Example 12.26, "Configuring a custom `GetterPropertySelectionStrategy` programmatically") or by using the `hibernate.validator.getter_property_selection_strategy` property in the XML configuration (see Example 12.27, "Configuring a custom `GetterPropertySelectionStrategy` using an XML property").

Example 12.26: Configuring a custom `GetterPropertySelectionStrategy` *programmatically*

```java
Validator validator = Validation.byProvider( HibernateValidator.class )
        .configure()
        // Setting a custom getter property selection strategy
        .getterPropertySelectionStrategy( new FluentGetterPropertySelectionStrategy() )
        .buildValidatorFactory()
        .getValidator();

User user = new User( "", "", "not an email" );

Set<ConstraintViolation<User>> constraintViolations = validator.validate( user );

assertEquals( 3, constraintViolations.size() );
```

Example 12.27: Configuring a custom `GetterPropertySelectionStrategy` *using an XML property*

```xml
<validation-config
        xmlns="https://jakarta.ee/xml/ns/validation/configuration"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/configuration
            https://jakarta.ee/xml/ns/validation/validation-configuration-3.0.xsd"
        version="3.0">

    <property name="hibernate.validator.getter_property_selection_strategy">

org.hibernate.validator.referenceguide.chapter12.getterselectionstrategy.NoPrefixGetterProp
ertySelectionStrategy
    </property>

</validation-config>
```

> ⚠️ It is important to mention that in cases where programmatic constraints are added using `HibernateValidatorConfiguration#addMapping(ConstraintMapping)`, adding mappings should always be done after the required getter property selection strategy is configured. Otherwise, the default strategy will be used for the mappings added before defining the strategy.

12.18. Customizing the property name resolution for constraint violations

Imagine that we have a simple data class that has `@NotNull` constraints on some fields:

Example 12.28: Person data class

```java
public class Person {
    @NotNull
    @JsonProperty("first_name")
    private final String firstName;

    @JsonProperty("last_name")
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

This class can be serialized to JSON by using the Jackson library:

```
public class PersonSerializationTest {
    private final ObjectMapper objectMapper = new ObjectMapper();

    @Test
    public void personIsSerialized() throws JsonProcessingException {
        Person person = new Person( "Clark", "Kent" );

        String serializedPerson = objectMapper.writeValueAsString( person );

        assertEquals( "{\"first_name\":\"Clark\",\"last_name\":\"Kent\"}", serializedPerson
);
    }
}
```

As we can see, the object is serialized to:

Example 12.30: Person as json

```
{
  "first_name": "Clark",
  "last_name": "Kent"
}
```

Notice how the names of the properties differ. In the Java object, we have `firstName` and `lastName`, whereas in the JSON output, we have `first_name` and `last_name`. We customized this behavior through `@JsonProperty` annotations.

Now imagine that we use this class in a REST environment, where a user can send a `Person` instance as JSON in the request body. It would be nice, when indicating on which field the validation failed, to indicate the name they use in their JSON request, `first_name`, and not the name we use internally in our Java code, `firstName`.

The `org.hibernate.validator.spi.nodenameprovider.PropertyNodeNameProvider` contract allows us to do this. By implementing it, we can define how the name of a property will be resolved during validation. In our case, we want to read the value from the Jackson configuration.

One example of how to do this is to leverage the Jackson API:

Example 12.31: JacksonPropertyNodeNameProvider implementation

```java
import org.hibernate.validator.spi.nodenameprovider.JavaBeanProperty;
import org.hibernate.validator.spi.nodenameprovider.Property;
import org.hibernate.validator.spi.nodenameprovider.PropertyNodeNameProvider;

import com.fasterxml.jackson.databind.BeanDescription;
import com.fasterxml.jackson.databind.JavaType;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.introspect.BeanPropertyDefinition;

public class JacksonPropertyNodeNameProvider implements PropertyNodeNameProvider {
    private final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public String getName(Property property) {
        if ( property instanceof JavaBeanProperty ) {
            return getJavaBeanPropertyName( (JavaBeanProperty) property );
        }

        return getDefaultName( property );
    }

    private String getJavaBeanPropertyName(JavaBeanProperty property) {
        JavaType type = objectMapper.constructType( property.getDeclaringClass() );
        BeanDescription desc = objectMapper.getSerializationConfig().introspect( type );

        return desc.findProperties()
                .stream()
                .filter( prop -> prop.getInternalName().equals( property.getName() ) )
                .map( BeanPropertyDefinition::getName )
                .findFirst()
                .orElse( property.getName() );
    }

    private String getDefaultName(Property property) {
        return property.getName();
    }
}
```

And when doing the validation:

Example 12.32: JacksonPropertyNodeNameProvider usage

```java
public class JacksonPropertyNodeNameProviderTest {
    @Test
    public void nameIsReadFromJacksonAnnotationOnField() {
        ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class
)
                .configure()
                .propertyNodeNameProvider( new JacksonPropertyNodeNameProvider() )
                .buildValidatorFactory();

        Validator validator = validatorFactory.getValidator();

        Person clarkKent = new Person( null, "Kent" );

        Set<ConstraintViolation<Person>> violations = validator.validate( clarkKent );
        ConstraintViolation<Person> violation = violations.iterator().next();

        assertEquals( violation.getPropertyPath().toString(), "first_name" );
    }
```

We can see that the property path now returns `first_name`.

Note that this also works when the annotations are on a getter:

Example 12.33: Annotation on a getter

```java
@Test
public void nameIsReadFromJacksonAnnotationOnGetter() {
    ValidatorFactory validatorFactory = Validation.byProvider( HibernateValidator.class )
            .configure()
            .propertyNodeNameProvider( new JacksonPropertyNodeNameProvider() )
            .buildValidatorFactory();

    Validator validator = validatorFactory.getValidator();

    Person clarkKent = new Person( null, "Kent" );

    Set<ConstraintViolation<Person>> violations = validator.validate( clarkKent );
    ConstraintViolation<Person> violation = violations.iterator().next();

    assertEquals( violation.getPropertyPath().toString(), "first_name" );
}

public class Person {
    private final String firstName;

    @JsonProperty("last_name")
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @NotNull
    @JsonProperty("first_name")
    public String getFirstName() {
        return firstName;
    }
}
```

This is just one use case of why we would like to change how the property names are resolved.

`org.hibernate.validator.spi.nodenameprovider.PropertyNodeNameProvider` can be implemented to provide a property name in whatever way you see fit (reading from annotations, for instance).

There are two more interfaces that are worth mentioning:

- `org.hibernate.validator.spi.nodenameprovider.Property` is a base interface that holds metadata about a property. It has a single `String getName()` method that can be used to get the "original" name of a property. This interface should be used as a default way of resolving the name (see how it is used in Example 12.31, "JacksonPropertyNodeNameProvider implementation").

- `org.hibernate.validator.spi.nodenameprovider.JavaBeanProperty` is an interface that holds metadata about a bean property. It extends `org.hibernate.validator.spi.nodenameprovider.Property` and provide some additional methods like `Class<?> getDeclaringClass()` which returns the class that is the owner of the property.

Chapter 13. 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.

> You can find the Hibernate Validator Annotation Processor as part of the distribution bundle on Sourceforge or in the usual Maven repositories such as Maven Central under the GAV `org.hibernate.validator:hibernate-validator-annotation-processor:7.0.5.Final`.

13.1. Prerequisites

The Hibernate Validator Annotation Processor is based on the "Pluggable Annotation Processing API" as defined by JSR 269 which is part of the Java Platform.

13.2. Features

As of Hibernate Validator 7.0.5.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

- annotation parameter values are meaningful and valid

- method parameter constraints in inheritance hierarchies respect the inheritance rules

- method return value constraints in inheritance hierarchies respect the inheritance rules

13.3. Options

The behavior of the Hibernate Validator Annotation Processor can be controlled using the following processor options:

`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 Jakarta 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`.

13.4. Using the Annotation Processor

This section shows in detail how to integrate the Hibernate Validator Annotation Processor into command line builds (Maven, Ant, javac) as well as IDE-based builds (Eclipse, IntelliJ IDEA, NetBeans).

13.4.1. Command line builds

13.4.1.1. Maven

For using the Hibernate Validator annotation processor with Maven, set it up via the `annotationProcessorPaths` option like this:

Example 13.1: Using the HV Annotation Processor with Maven

```xml
<project>
    [...]
    <build>
        [...]
        <plugins>
            [...]
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.hibernate.validator</groupId>
                            <artifactId>hibernate-validator-annotation-
processor</artifactId>
                            <version>7.0.5.Final</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
            [...]
        </plugins>
        [...]
    </build>
    [...]
</project>
```

13.4.1.2. Gradle

When using Gradle it is enough to reference the annotation processor as an `annotationProcessor`
dependency.

Example 13.2: Using the annotation processor with Gradle

```
dependencies {
    annotationProcessor group: 'org.hibernate.validator', name: 'hibernate-validator-
annotation-processor', version: '7.0.5.Final'

    // any other dependencies ...
}
```

13.4.1.3. Apache Ant

Similar to directly working with javac, the annotation processor can be added as as compiler argument
when invoking the javac task for Apache Ant:

Example 13.3: Using the annotation processor with Ant

```
<javac srcdir="src/main"
       destdir="build/classes"
       classpath="/path/to/validation-api-3.0.0.jar">
       <compilerarg value="-processorpath" />
       <compilerarg value="/path/to/hibernate-validator-annotation-processor-
7.0.5.Final.jar"/>
</javac>
```

13.4.1.4. javac

When compiling on the command line using javac, specify the JAR *hibernate-validator-annotation-processor-7.0.5.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 13.4: Using the annotation processor with javac

```
javac src/main/java/org/hibernate/validator/ap/demo/Car.java \
    -cp /path/to/validation-api-3.0.0.jar \
    -processorpath /path/to/hibernate-validator-annotation-processor-7.0.5.Final.jar
```

13.4.2. IDE builds
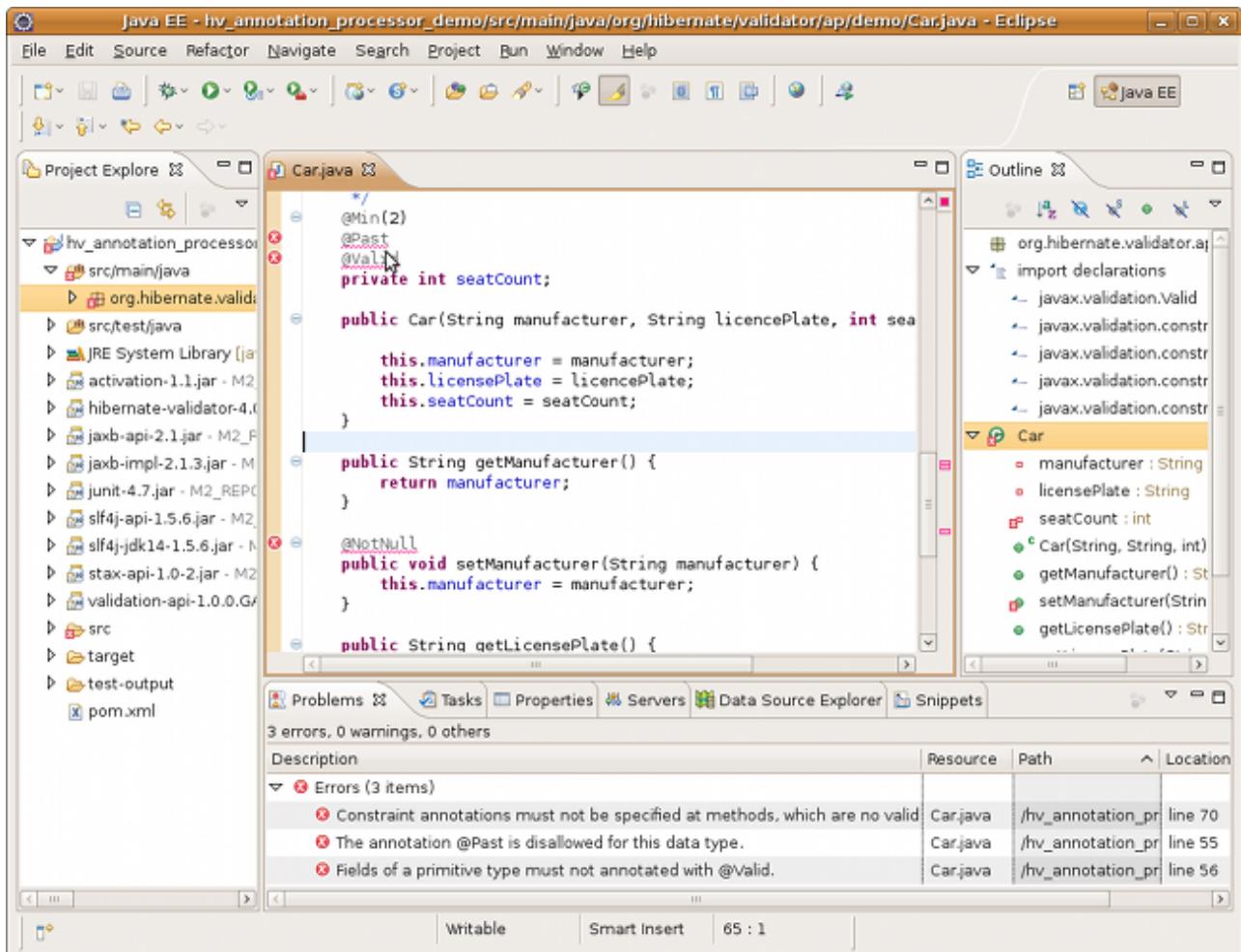
13.4.2.1. Eclipse

The annotation processor will automatically be set up for Maven projects configured as described above, provided you have the M2E Eclipse plug-in installed.

For plain Eclipse projects follow these steps to set up the annotation processor:

- Right-click your project, choose "Properties"

- Go to "Java Compiler" and make sure, that "Compiler compliance level" is set to "1.8". 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-7.0.5.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:
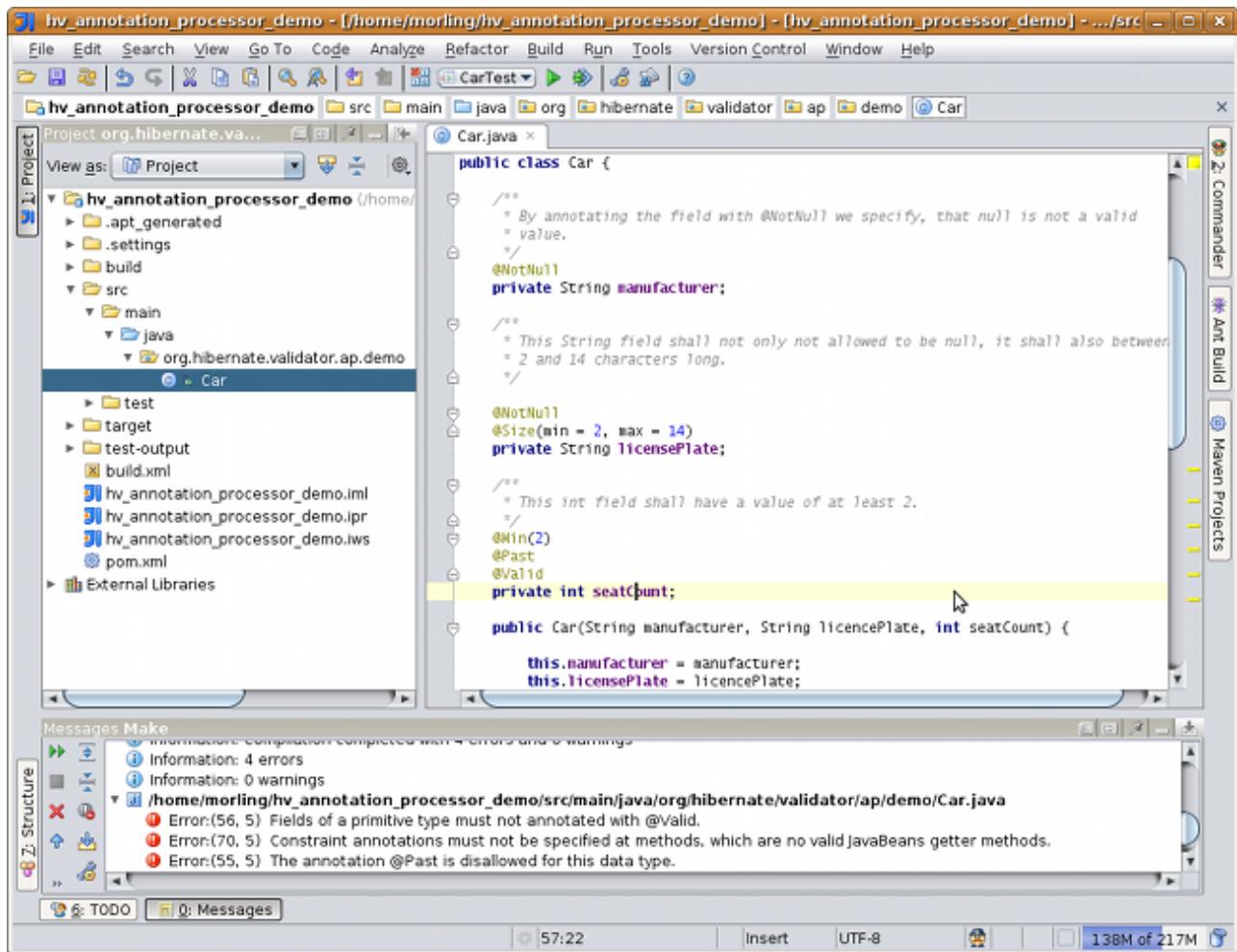
13.4.2.2. IntelliJ IDEA

The following steps must be followed to use the annotation processor within IntelliJ 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-7.0.5.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 erroneous constraint annotations:

13.4.2.3. NetBeans

The NetBeans 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-7.0.5.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:

13.5. Known issues

The following known issues exist as of July 2017:

- Container element constraints are not supported for now.

- Constraints applied to a container but in reality applied to the container elements (be it via the `Unwrapping.Unwrap` payload or via a value extractor marked with `@UnwrapByDefault`) are not supported correctly.

- HV-308: Additional validators registered for a constraint using XML are not evaluated by the annotation processor.

- Sometimes custom constraints can't be properly evaluated 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.

Chapter 14. Further reading

Last but not least, a few pointers to further information.

A great source for examples is the Jakarta Bean Validation TCK which is available for anonymous access on GitHub. In particular the TCK's tests might be of interest. The Jakarta Bean Validation specification itself is also a great way to deepen your understanding of Jakarta Bean Validation and Hibernate Validator.

If you have any further questions about Hibernate Validator or want to share some of your use cases, have a look at the Hibernate Validator Wiki, the Hibernate Validator Forum and the Hibernate Validator tag on Stack Overflow.

In case you would like to report a bug use Hibernate's Jira instance. Feedback is always welcome!