Hibernate Validator

JSR 303 Reference Implementation

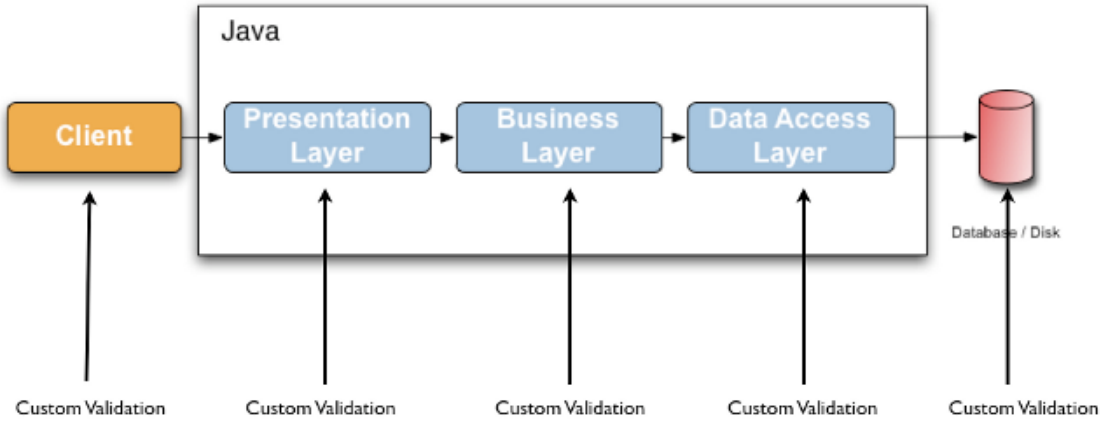
Reference Guide

5.0.0.CR2

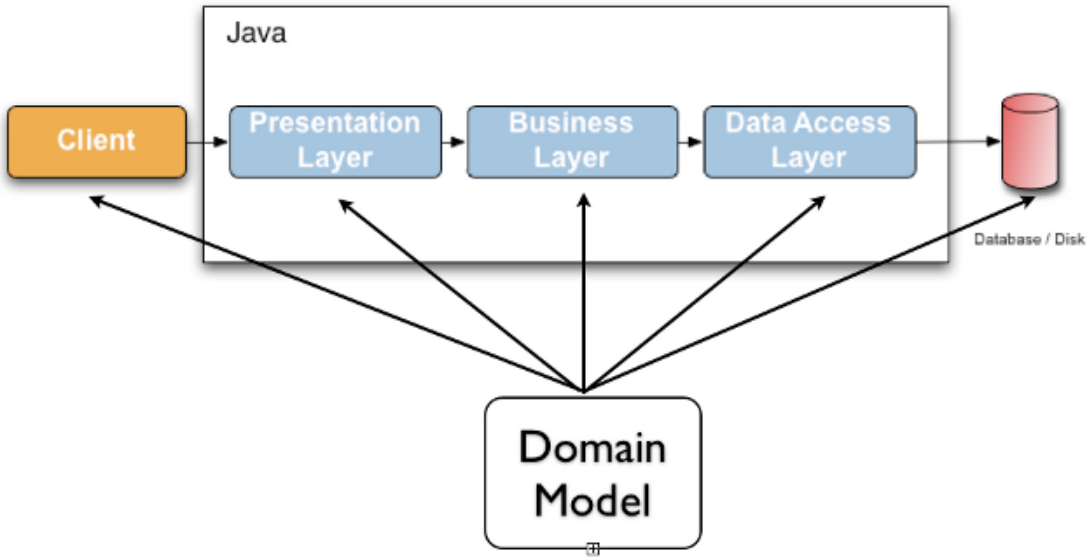
by Hardy Ferentschik and Gunnar Morling

Preface

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



JSR 303 - Bean Validation - defines a metadata model and API for entity validation. The default metadata source is 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 or programming model. It is specifically not tied to either the web tier or the persistence tier, and is available for both server-side application programming, as well as rich client Swing application developers.



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

Getting started

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

- A JDK >= 6
- *Apache Maven* [http://maven.apache.org/]
- An Internet connection (Maven has to download all required libraries)
- A properly configured remote repository. Add the following to your `settings.xml`:

Example 1.1. Configuring the JBoss Maven repository

```
<repositories>
    <repository>
        <id>jboss-public-repository-group</id>
        <url>https://repository.jboss.org/nexus/content/groups/public-jboss</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
     </repository>
</repositories>
```

More information about `settings.xml` can be found in the *Maven Local Settings Model* [http://maven.apache.org/ref/2.0.8/maven-settings/settings.html].

1.1. Project set up

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

Example 1.2. Hibernate Validator Maven dependency

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.0.0.CR2</version>
</dependency>
```

Tip

For the purposes of logging, Hibernate Validator uses the JBoss Logging API. This is an abstraction layer which supports several known logging solutions (e.g. log4j or the logging framework provided by the JDK) as implementation. Just add your preferred logging library to the classpath and all log requests from Hibernate Validator will automatically be delegated to that logging provider.

Alternatively, you can explicitely specify a provider using the system property `org.jboss.logging.provider`. Supported values currently are `jboss`, `jdk`, `log4j` and `slf4j`.

1.2. Applying constraints

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

Example 1.3. Class Car annotated with constraints

```
package com.mycompany;

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

public class Car {

    @NotNull
    private String manufacturer;

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

    @Min(2)
    private int seatCount;

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

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

`@NotNull`, `@Size` and `@Min` are so-called constraint annotations, that we use to declare constraints, which shall be applied to the fields of a `Car` instance:

- manufacturer shall never be null

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

1.3. Validating constraints

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

Example 1.4. Class CarTest showing validation examples

```
package com.mycompany;

import static org.junit.Assert.*;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

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

public class CarTest {

    private static Validator validator;

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

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

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

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

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

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

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

```
    }

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

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

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

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

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

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

In the `setUp()` method we get a `Validator` instance from the `ValidatorFactory`. A `Validator` instance is thread-safe and may be reused multiple times. For this reason we store it as field of our test class. We can use the `Validator` now to validate the different car instances in the test methods.

The `validate()` method returns a set of `ConstraintViolation` instances, which we can iterate 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.

Note that we only use classes from the package javax.validation from the Bean Validation API. As we don't reference any classes of the RI directly, it would be no problem to switch to another implementation of the API, should that need arise.

1.4. Where to go next?

That concludes our 5 minute tour through the world of Hibernate Validator. Continue exploring the code examples or look at further examples referenced in *Chapter 10, Further reading*. To deepen your understanding of Hibernate Validator just continue reading *Chapter 2, Validation step*

by step. In case your application has specific validation requirements have a look at *Chapter 3, Creating custom constraints*.

Validation step by step

In this chapter we will see in more detail how to use Hibernate Validator to validate constraints for a given entity model. We will also learn which default constraints the Bean Validation specification provides and which additional constraints are only provided by Hibernate Validator. Let's start with how to add constraints to an entity.

2.1. Defining constraints

Constraints in Bean Validation are expressed via Java annotations. In this section we show how to annotate an object model with these annotations. We have to differentiate between three different type of constraint annotations - field-, property-, and class-level annotations.

Note

Not all constraints can be placed on all of these levels. In fact, none of the default constraints defined by Bean Validation can be placed at class level. The `java.lang.annotation.Target` annotation in the constraint annotation itself determines on which elements a constraint can be placed. See *Chapter 3, 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 constraint"* shows a field level configuration example:

Example 2.1. Field level constraint

```
package com.mycompany;

import javax.validation.constraints.NotNull;

public class Car {

    @NotNull
    private String manufacturer;

    @AssertTrue
    private boolean isRegistered;

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

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

Note

The access type (private, protected or public) does not matter.

Note

Static fields and properties cannot be validated.

Tip

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

2.1.2. Property-level constraints

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

Note

The property's getter method has to be annotated, not its setter.

Example 2.2. Property level constraint

```
package com.mycompany;

import javax.validation.constraints.AssertTrue;
import javax.validation.constraints.NotNull;

public class Car {

    private String manufacturer;

    private boolean isRegistered;
```

```
    public Car(String manufacturer, boolean isRegistered) {
        super();
        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;
    }
}
```

When using property level constraints property access strategy is used to access the value to be validated. This means the bean validation provider accesses the state via the property accessor method. One advantage of annotating properties instead of fields is that the constraints become part of the constrained type's API that way and users are aware of the existing constraints without having to examine the type's implementation.

Tip

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

2.1.3. Class-level constraints

Last but not least, a constraint can also be placed on class level. When a constraint annotation is placed on this level the class instance itself passed to the `ConstraintValidator`. Class level constraints are useful if it is necessary to inspect more than a single property of the class to validate it or if a correlation between different state variables has to be evaluated. In *Example 2.3, "Class level constraint"* we add the property passengers to the class `Car`. We also add the constraint `PassengerCount` on the class level. We will later see how we can actually create this custom constraint (see *Chapter 3, Creating custom constraints*). For now it is enough to know that `PassengerCount` will ensure that there cannot be more passengers in a car than there are seats.

Example 2.3. Class level constraint

```
package com.mycompany;

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

@PassengerCount
public class Car {

    @NotNull
    private String manufacturer;

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

    @Min(2)
    private int seatCount;

    private List<Person> passengers;

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

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

2.1.4. Constraint inheritance

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

Example 2.4. Constraint inheritance using RentalCar

```
package com.mycompany;

import javax.validation.constraints.NotNull;

public class RentalCar extends Car {

    private String rentalStation;

    public RentalCar(String manufacturer, String rentalStation) {
        super(manufacturer);
        this.rentalStation = rentalStation;
    }
```

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

    public void setRentalStation(String rentalStation) {
        this.rentalStation = rentalStation;
    }
}
```

Our well-known class `Car` is now extended by `RentalCar` with the additional property rentalStation. If an instance of `RentalCar` is validated, not only the `@NotNull` constraint on rentalStation is validated, but also the constraint on manufacturer from the parent class.

The same would hold true, if `Car` were an interface implemented by `RentalCar`.

Constraint annotations are aggregated if methods are overridden. If `RentalCar` would override the `getManufacturer()` method from `Car` any constraints annotated at the overriding method would be evaluated in addition to the `@NotNull` constraint from the super-class.

2.1.5. Object graphs

The Bean Validation API does not only allow to validate single class instances but also complete object graphs. To do so, just annotate a field or property representing a reference to another object with `@Valid`. If the parent object is validated, all referenced objects annotated with `@Valid` will be validated as well (as will be their children etc.). See *Example 2.6, "Adding a driver to the car"*.

Example 2.5. Class Person

```
package com.mycompany;

import javax.validation.constraints.NotNull;

public class Person {

    @NotNull
    private String name;

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

    public String getName() {
        return name;
    }

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

Example 2.6. Adding a driver to the car

```
package com.mycompany;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Car {

    @NotNull
    @Valid
    private Person driver;

    public Car(Person driver) {
        this.driver = driver;
    }

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

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`.

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

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

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

Example 2.7. Car with a list of passengers

```
package com.mycompany;

import java.util.ArrayList;
import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Car {

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

    public Car(List<Person> passengers) {
```

```
        this.passengers = passengers;
    }

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

If a `Car` instance is validated, a `ConstraintValidation` will be created, if any of the `Person` objects contained in the passengers list has a `null` name.

Note

`null` values are getting ignored when validating object graphs.

2.2. Validating constraints

The `Validator` interface is the main entry point to Bean Validation. In *Section 5.1, "Configuration and ValidatorFactory"* we will first show how to obtain an `Validator` instance. Afterwards we will 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 `Validation.buildDefaultValidatorFactory()` method:

Example 2.8. Validation.buildDefaultValidatorFactory()

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

For other ways of obtaining a Validator instance see *Chapter 5, Bootstrapping*. For now we just want to see how we can use the `Validator` instance to validate entity instances.

2.2.2. Validator methods

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

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

All the validation methods have a var-args parameter which can be used to specify, which validation groups shall be considered when performing the validation. If the parameter is not specified the default validation group (`javax.validation.groups.Default`) will be used. We will go into more detail on the topic of validation groups in *Section 2.3, "Validating groups"*

2.2.2.1. `validate`

Use the `validate()` method to perform validation of all constraints of a given entity instance (see *Example 2.9, "Usage of Validator.validate()"*).

Example 2.9. Usage of `Validator.validate()`

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

Car car = new Car(null);

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

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

2.2.2.2. `validateProperty`

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

Example 2.10. Usage of `Validator.validateProperty()`

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

Car car = new Car(null);

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

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

`Validator.validateProperty` is for example used in the integration of Bean Validation into JSF 2 (see *Section 7.4, "Presentation layer validation"*).

2.2.2.3. `validateValue`

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 2.11. Usage of `Validator.validateValue()`

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

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

assertEquals(1, constraintViolations.size());
```

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

Note

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

2.2.3. `ConstraintViolation` methods

Now it is time to have a closer look at what a `ConstraintViolation`. Using the different methods of `ConstraintViolation` a lot of useful information about the cause of the validation failure can be determined. *Table 2.1, "The various ConstraintViolation methods"* gives an overview of these methods:

Table 2.1. The various `ConstraintViolation` methods

Method	Usage	Example (referring to *Example 2.9, "Usage of Validator.validate()"*)
`getMessage()`	The interpolated error message.	may not be null
`getMessageTemplate()`	The non-interpolated error message.	{javax.validation.constraints.NotNull.message}
`getRootBean()`	The root bean being validated.	car
`getRootBeanClass()`	The class of the root bean being validated.	Car.class
`getLeafBean()`	If a bean constraint, the bean instance the constraint is applied on. If a property constraint, the bean instance hosting the property the constraint is applied on.	car
`getPropertyPath()`	The property path to the value from root bean.	
`getInvalidValue()`	The value failing to pass the constraint.	passengers
`getConstraintDescriptor()`	Constraint metadata reported to fail.	

2.2.4. Message interpolation

As we will see in *Chapter 3, Creating custom constraints* each constraint definition must define a default message descriptor. This message can be overridden at declaration time

using the `message` attribute of the constraint. You can see this in *Example 2.13, "Driver"*. This message descriptors get interpolated when a constraint validation fails using the configured `MessageInterpolator`. The interpolator will try to resolve any message parameters, meaning string literals enclosed in braces. In order to resolve these parameters Hibernate Validator's default `MessageInterpolator` first recursively resolves parameters against a custom `ResourceBundle` called `ValidationMessages.properties` at the root of the classpath (It is up to you to create this file). If no further replacements are possible against the custom bundle the default `ResourceBundle` under `/org/hibernate/validator/ValidationMessages.properties` gets evaluated. If a replacement occurs against the default bundle the algorithm looks again at the custom bundle (and so on). Once no further replacements against these two resource bundles are possible remaining parameters are getting resolved against the attributes of the constraint to be validated.

Since the braces { and } have special meaning in the messages they need to be escaped if they are used literally. The following The following rules apply:

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

If the default message interpolator does not fit your requirements it is possible to plug a custom `MessageInterpolator` when the `ValidatorFactory` gets created. This can be seen in *Chapter 5, Bootstrapping*.

2.3. Validating groups

Groups allow you to restrict the set of constraints applied during validation. This makes for example wizard like validation possible where in each step only a specified subset of constraints get validated. The groups targeted are passed as var-args parameters to `validate`, `validateProperty` and `validateValue`. Let's have a look at an extended `Car` with `Driver` example. First we have the class `Person` (*Example 2.12, "Person"*) which has a `@NotNull` constraint on name. Since no group is specified for this annotation its default group is `javax.validation.groups.Default`.

Note

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

Example 2.12. Person

```
public class Person {
```

```
    @NotNull
    private String name;

    public Person(String name) {
        this.name = name;
    }
    // getters and setters ...
}
```

Next we have the class `Driver` (*Example 2.13, "Driver"*) extending `Person`. Here we are adding the properties age and hasDrivingLicense. In order to drive you must be at least 18 (`@Min(18)`) and you must have a driving license (`@AssertTrue`). Both constraints defined on these properties belong to the group `DriverChecks`. As you can see in *Example 2.14, "Group interfaces"* the group `DriverChecks` 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.

Example 2.13. Driver

```
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;
    }
}
```

Example 2.14. Group interfaces

```
public interface DriverChecks {
}

public interface CarChecks {
}
```

Last but not least we add the property passedVehicleInspection to the `Car` class (*Example 2.15, "Car"*) indicating whether a car passed the road worthy tests.

Example 2.15. Car

```
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;
    }
}
```

Overall three different groups are used in our example. Person.name, Car.manufacturer, Car.licensePlate and Car.seatCount all belong to the `Default` group. Driver.age and Driver.hasDrivingLicense belong to `DriverChecks` and last but not least Car.passedVehicleInspection belongs to the group `CarChecks`. *Example 2.16, "Drive away"* shows how passing different group combinations to the `Validator.validate` method result in different validation results.

Example 2.16. Drive away

```
public class GroupTest {

    private static Validator validator;

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

    @Test
    public void driveAway() {
        // create a car and check that everything is ok with it.
```

```
        Car car = new Car( "Morris", "DD-AB-123", 2 );
        Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );
        assertEquals( 0, constraintViolations.size() );

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

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

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

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

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

First we create a car and validate it using no explicit group. There are no validation errors, even though the property passedVehicleInspection is per default `false`. However, the constraint defined on this property does not belong to the default group. Next we just validate the `CarChecks` group which will fail until we make sure that the car passes the vehicle inspection. When we then add a driver to the car and validate against `DriverChecks` we get again a 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` will pass.

Last but not least, we show that all constraints are passing by validating against all defined groups.

2.3.1. Group sequences

By default, constraints are evaluated in no particular order, regardless of which groups they belong to. In some situations, however, it is useful to control the order constraints are evaluated. In our example from *Section 2.3, "Validating groups"* we could for example require that first all default car constraints are passing before we check the road worthiness of the car. Finally before we drive away we check the actual driver constraints. In order to implement such an order one would define a new interface and annotate it with `@GroupSequence` defining the order in which the groups have to be validated.

Note

If at least one constraint fails in a sequenced group none of the constraints of the following groups in the sequence get validated.

Example 2.17. Interface with @GroupSequence

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

Warning

Groups defining a sequence and groups composing a sequence must not be involved in a cyclic dependency either directly or indirectly, either through cascaded sequence definition or group inheritance. If a group containing such a circularity is evaluated, a `GroupDefinitionException` is raised.

The usage of the new sequence could then look like in *Example 2.18, "Usage of a group sequence"*.

Example 2.18. Usage of a group sequence

```
@Test
public void testOrderedChecks() {
    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() );
}
```

2.3.2. Redefining the default group sequence of a class

2.3.2.1. @GroupSequence

The `@GroupSequence` annotation also fulfills a second purpose. It allows you to redefine what the `Default` group means for a given class. To redefine `Default` for a given class, add a `@GroupSequence` annotation to the class. The defined groups in the annotation express the

sequence of groups that substitute `Default` for this class. *Example 2.19, "RentalCar with @GroupSequence"* introduces a new class `RentalCar` with a redefined default group. With this definition you can evaluate the constraints belonging to `RentalChecks`, `CarChecks` and `RentalCar` by just requesting the `Default` group as seen in *Example 2.20, "RentalCar with redefined default group"*.

Example 2.19. RentalCar with @GroupSequence

```
@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;
    }
}
```

Example 2.20. RentalCar with redefined default group

```
/**
 * Validating the default group leads to validation on the default group sequence of {@code RentalCar}.
 */
@Test
public void carIsRented() {
  RentalCar rentalCar = new RentalCar( "Morris", "DD-AB-123", 2 );
  rentalCar.setPassedVehicleInspection( true );
  rentalCar.setRented( true );

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

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

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

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

Note

Due to the fact that there cannot be a 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!

Note

The `Default` group sequence overriding is local to the class it is defined on and is not propagated to the associated objects. This means in particular 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 when validation a rental car instance.

2.3.2.2. @GroupSequenceProvider

The `@javax.validation.GroupSequence` annotation is a standardized Bean Validation annotation. As seen in the previous section it allows you to statically redefine the default group sequence for a class. Hibernate Validator also offers a custom, non standardized annotation - `org.hibernate.validator.group.GroupSequenceProvider` - which allows for dynamic redefinition of the default group sequence. Using the rental car scenario again, one could dynamically add the `CarChecks` as seen in *Example 2.21, "RentalCar with @GroupSequenceProvider"* and *Example , "DefaultGroupSequenceProvider implementation"*.

Example 2.21. RentalCar with @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;
    }
}
```

Example . DefaultGroupSequenceProvider implementation

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

2.4. Built-in constraints

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

2.4.1. Bean Validation constraints

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

Note

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

Table 2.2. Bean Validation constraints

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

Annotation	Supported data types	Use	Hibernate metadata impact
@AssertTrue	`Boolean`, `boolean`	Checks that the annotated element is `true`.	none
@DecimalMax	`BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of `Number` and `CharSequence`.	The annotated element must be a number whose value must be lower or equal to the specified maximum. The parameter value is the string representation of the max value according to the `BigDecimal` string representation.	none
@DecimalMin	`BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of `Number` and `CharSequence`.	The annotated element must be a number whose value must be higher or equal to the specified minimum. The parameter value is the string representation of the min value according to the `BigDecimal` string representation.	none
@Digits(integer=, fraction=)	`BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of `Number` and `CharSequence`.	Checks whether the annoted value is a number having up to `integer` digits and `fraction` fractional digits.	Define column precision and scale.
@Future	`java.util.Date`, `java.util.Calendar`; Additionally supported by	Checks whether the annotated date is in the future.	none

Annotation	Supported data types	Use	Hibernate metadata impact
	HV, if the *Joda Time* [http://joda-time.sourceforge.net/] date/time API is on the class path: any implementations of `ReadablePartial` and `ReadableInstant`.		
@Max	`BigDecimal`, `BigInteger`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of`CharSequence` (the numeric value represented by the character sequence is evaluated), any sub-type of `Number`.	Checks whether the annotated value is less than or equal to the specified maximum.	Add a check constraint on the column.
@Min	`BigDecimal`, `BigInteger`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of `CharSequence` (the numeric value represented by the char sequence is evaluated), any sub-type of `Number`.	Checks whether the annotated value is higher than or equal to the specified minimum.	Add a check constraint on the column.

Annotation	Supported data types	Use	Hibernate metadata impact
@NotNull	Any type	Checks that the annotated value is not `null`.	Column(s) are not null.
@Null	Any type	Checks that the annotated value is `null`.	none
@Past	`java.util.Date`, `java.util.Calendar`; Additionally supported by HV, if the *Joda Time* [http://joda-time.sourceforge.net/] date/time API is on the class path: any implementations of `ReadablePartial` and `ReadableInstant`.	Checks whether the annotated date is in the past.	none
@Pattern(regex=, flag=)	`String`. Additionally supported by HV: any sub-type of `CharSequence`.	Checks if the annotated string matches the regular expression *`regex`* considering the given flag *`match`*.	none
@Size(min=, max=)	`String`, `Collection`, `Map` and `arrays`. Additionally supported by HV: any sub-type of `CharSequence`.	Checks if the annotated element's size is between min and max (inclusive).	Column length will be set to max.
@Valid	Any non-primitive type	Performs validation recursively on the associated object. If the object is a collection or an array, the elements are validated recursively. If the object is a map, the value elements	none

Annotation	Supported data types	Use	Hibernate metadata impact
		are validated recursively.	

Note

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

2.4.2. Additional constraints

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

Table 2.3. Custom constraints

Annotation	Supported data types	Use	Hibernate metadata impact
@CreditCardNumber	`CharSequence`	Checks that the annotated character sequence passes the Luhn checksum test. Note, this validation aims to check for user mistakes, not credit card validity! See also *Anatomy of Credit Card Numbers* [http://www.merriampark.com/anatomycc.htm].	none
@Email	`CharSequence`	Checks whether the specified character sequence is a valid email address. The optional parameters `regexp` and `flags` allow to specify an additional regular expression (including	none

Annotation	Supported data types	Use	Hibernate metadata impact
		regular expression flags) which the email must match.	
@Length(min=, max=)	`CharSequence`	Validates that the annotated character sequence is between *`min`* and *`max`* included.	Column length will be set to max.
@ModCheck(modType=, multiplier=, startIndex=, endIndex=, checkDigitPosition=, ignoreNonDigitCharacters=)	`CharSequence`	Checks that the digits within the annotated character sequence pass the mod 10 or mod 11 checksum algorithm. `modType` is used to select the modulo type and the `multiplier` determines the algorithm specific multiplier (see also *Luhn algorithm* [http://en.wikipedia.org/wiki/Luhn_algorithm]). `startIndex` and `endIndex` allow to only run the modulo algorithm on the specified sub-string. `checkDigitPosition` allows to use an arbitrary digit within the character sequence to be 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.	none

Annotation	Supported data types	Use	Hibernate metadata impact
@NotBlank	`CharSequence`	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 strings and that trailing whitespaces are ignored.	none
@NotEmpty	`CharSequence`, `Collection`, `Map` and arrays	Checks whether the annotated element is not `null` nor empty.	none
@Range(min=, max=)	`BigDecimal`, `BigInteger`, `CharSequence`, `byte`, `short`, `int`, `long` and the respective wrappers of the primitive types	Checks whether the annotated value lies between (inclusive) the specified minimum and maximum.	none
@SafeHtml(whitelistType=, additionalTags=)	`CharSequence`	Checks whether the annotated value contains potentially malicious fragments such as `<script/>`. In order to use this constraint, the *jsoup* [http://jsoup.org/] library must be part of the class path. With the *`whitelistType`* attribute predefined whitelist types can be chosen. You can also specify additional html tags for the whitelist with the *`additionalTags`* attribute.	none

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

Annotation	Supported data types	Use	Hibernate metadata impact
		which the URL must match.	

2.4.2.1. Country specific constraints

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

Note

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

Table 2.4. Custom country specific constraints

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

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

Tip

In some cases neither the Bean Validation constraints nor the custom constraints provided by Hibernate Validator will fulfill your requirements. In this case you can easily write your own constraint. We will discuss this in *Chapter 3, Creating custom constraints*.

Creating custom constraints

Though the Bean Validation API defines a whole set of standard constraint annotations one can easily think of situations in which these standard annotations won't suffice. For these cases you are able to create custom constraints tailored to your specific validation requirements in a simple manner.

3.1. 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

3.1.1. The constraint annotation

Let's write a constraint annotation, that can be used to express that a given string shall either be upper case or lower case. We'll apply it later on to the licensePlate field of the `Car` class from *Chapter 1, Getting started* to ensure, that the field is always an upper-case string.

First we need a way to express the two case modes. We might use `String` constants, but a better way to go is to use a Java 5 enum for that purpose:

Example 3.1. Enum `CaseMode` to express upper vs. lower case

```
package com.mycompany;

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

Now we can 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 3.2. Defining CheckCase constraint annotation

```
package com.mycompany;

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

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
```

```
import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
public @interface CheckCase {

    String message() default "{com.mycompany.constraints.checkcase}";

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

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

    CaseMode value();

}
```

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

- an attribute message that returns the default key for creating error messages in case the constraint is violated
- an attribute groups that allows the specification of validation groups, to which this constraint belongs (see *Section 2.3, "Validating groups"*). This must default to an empty array of type `Class<?>`.
- an attribute `payload` that can be used by clients of the Bean Validation API to assign custom payload objects to a constraint. This attribute is not used by the API itself.

Tip

An example for a custom payload could be the definition of a severity.

```
public class Severity {
    public static class Info extends Payload {};
    public static class Error extends Payload {};
}

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

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

    // ...
```

```
}
```

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

Besides those three mandatory attributes (message, groups and payload) we add another one allowing for the required case mode to be specified. The name value is a special one, which can be omitted upon using the annotation, if it is the only attribute specified, as e.g. in `@CheckCase(CaseMode.UPPER)`.

In addition we annotate the annotation type with a couple of so-called meta annotations:

- `@Target({ METHOD, FIELD, ANNOTATION_TYPE })`: Says, that methods, fields and annotation declarations may be annotated with @CheckCase (but not type declarations e.g.)
- `@Retention(RUNTIME)`: Specifies, that annotations of this type will be available at runtime by the means of reflection
- `@Constraint(validatedBy = CheckCaseValidator.class)`: Specifies the validator to be used to validate elements annotated with @CheckCase
- `@Documented`: Says, that the use of `@CheckCase` will be contained in the JavaDoc of elements annotated with it

Tip

Hibernate Validator provides support for the validation of method parameters using constraint annotations (see *Section 8.3, "Method validation"*).

In order to use a custom constraint for parameter validation the `ElementType.PARAMETER` must be specified within the `@Target` annotation. This is already the case for all constraints defined by the Bean Validation API and also the custom constraints provided by Hibernate Validator.

3.1.2. The constraint validator

Next, we need to implement a constraint validator, that's able to validate elements with a `@CheckCase` annotation. To do so, we implement the interface `ConstraintValidator` as shown below:

Example 3.3. Implementing a constraint validator for the constraint `CheckCase`

```
package com.mycompany;
```

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

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

    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 we set in our implementation. The first one specifies the annotation type to be validated (in our example `CheckCase`), the second one the type of elements, which the validator can handle (here `String`).

In case a constraint annotation is allowed at elements of different 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 us access to the attribute values of the annotation to be validated. In the example we store the `CaseMode` in a field of the validator for further usage.

In the `isValid()` method we implement the logic, that determines, whether a `String` is valid according to a given `@CheckCase` annotation or not. This decision depends on the case mode retrieved in `initialize()`. As the Bean Validation specification recommends, we consider `null` values as being valid. If `null` is not a valid value for an element, it should be annotated with `@NotNull` explicitly.

3.1.2.1. The ConstraintValidatorContext

Example 3.3, "Implementing a constraint validator for the constraint CheckCase" relies on the default error message generation by just returning `true` or `false` from the `isValid` call. 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 3.4. Use of ConstraintValidatorContext to define custom error messages

```
package com.mycompany;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

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

    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( "{com.mycompany.constraints.CheckCase.message}
        }
        return result;
    }

}
```

Example 3.4, "Use of ConstraintValidatorContext to define custom error messages" shows how you can disable the default error message generation and add a custom error message using a specified message template. In this example the use of the `ConstraintValidatorContext` results in the same error message as the default error message generation.

Tip

It is important to end each new constraint violation with `addConstraintViolation`. Only after that the new constraint violation will be created.

In case you are implementing a `ConstraintValidator` a class level constraint it is also possible to adjust set the property path for the created constraint violations. This is important for the case

where you validate multiple properties of the class or even traverse the object graph. A custom property path creation could look like *Example 3.5, "Adding new ConstraintViolation with custom property path"*.

Example 3.5. Adding new `ConstraintViolation` with custom property path

```
public boolean isValid(Group group, ConstraintValidatorContext constraintValidatorContext) {
    boolean isValid = false;
    ...

    if(!isValid) {
        constraintValidatorContext
            .buildConstraintViolationWithTemplate( "{my.custom.template}" )
            .addNode( "myProperty" ).addConstraintViolation();
    }
    return isValid;
}
```

3.1.3. The error message

Finally we need to specify the error message, that shall be used, in case a `@CheckCase` constraint is violated. To do so, we add the following to our custom `ValidationMessages.properties` (see also *Section 2.2.4, "Message interpolation"*)

Example 3.6. Defining a custom error message for the `CheckCase` constraint

```
com.mycompany.constraints.CheckCase.message=Case mode must be {value}.
```

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

3.1.4. Using the constraint

Now that our first custom constraint is completed, we can use it in the `Car` class from the *Chapter 1, Getting started* chapter to specify that the licensePlate field shall only contain upper-case strings:

Example 3.7. Applying the `CheckCase` constraint

```
package com.mycompany;

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

public class Car {

    @NotNull
```

```
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    @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 let's demonstrate in a little test that the `@CheckCase` constraint is properly validated:

Example 3.8. Testcase demonstrating the `checkcase` validation

```
package com.mycompany;

import static org.junit.Assert.*;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

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

public class CarTest {

    private static Validator validator;

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

    @Test
    public void testLicensePlateNotUpperCase() {

        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());
    }

    @Test
    public void carIsValid() {

        Car car = new Car("Morris", "DD-AB-123", 4);

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

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

3.2. Constraint composition

Looking at the licensePlate field of the `Car` class in *Example 3.7, "Applying the CheckCase constraint"*, we see three constraint annotations already. In complexer scenarios, where even more constraints could be applied to one element, this might become a bit confusing easily. Furthermore, if we had a licensePlate field in another class, we would have to copy all constraint declarations to the other class as well, violating the DRY principle.

This problem can be tackled using compound constraints. In the following we create a new constraint annotation `@ValidLicensePlate`, that comprises the constraints `@NotNull`, `@Size` and `@CheckCase`:

Example 3.9. Creating a composing constraint `ValidLicensePlate`

```
package com.mycompany;

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

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

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

```
public @interface ValidLicensePlate {

    String message() default "{com.mycompany.constraints.validlicenseplate}";

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

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

}
```

To do so, we just have to annotate the constraint declaration with its comprising constraints (btw. that's exactly why we allowed annotation types as target for the `@CheckCase` annotation). As no additional validation is required for the `@ValidLicensePlate` annotation itself, we don't declare a validator within the `@Constraint` meta annotation.

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

Example 3.10. Application of composing constraint `ValidLicensePlate`

```
package com.mycompany;

public class Car {

    @ValidLicensePlate
    private String licensePlate;

    //...

}
```

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

Example 3.11. Usage of `@ReportAsSingleViolation`

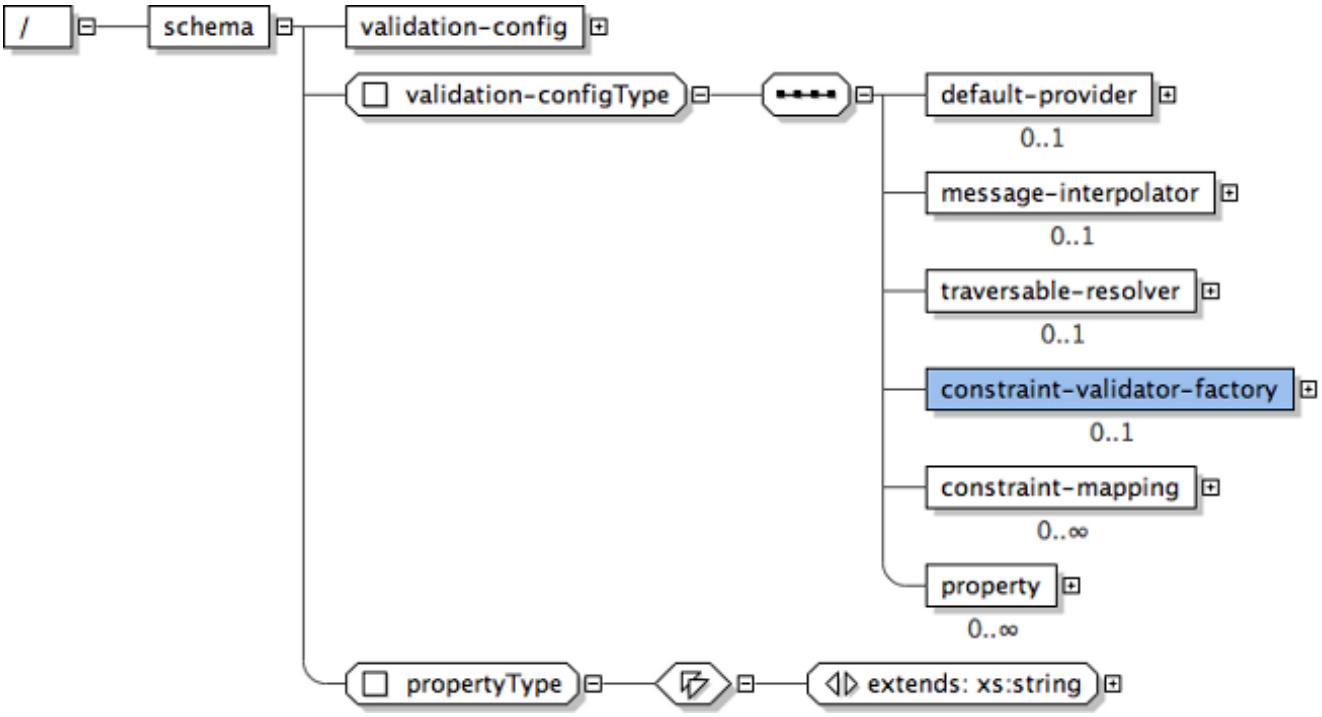
```
//...
@ReportAsSingleViolation
public @interface ValidLicensePlate {

    String message() default "{com.mycompany.constraints.validlicenseplate}";

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

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

}
```

XML configuration

4.1. `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 `ValidationFactory` gets created. *Example 4.1, "validation-configuration-1.0.xsd"* shows a model view of the XML schema to which `validation.xml` has to adhere.

Example 4.1. validation-configuration-1.0.xsd



Example 4.2, "validation.xml" shows the several configuration options of `validation.xml`.

Example 4.2. validation.xml

```
<validation-config
    xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">

    <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
                                                                    <message-
interpolator>org.hibernate.validator.messageinterpolation.ResourceBundleMessageInterpolator</
message-interpolator>
     <traversable-resolver>org.hibernate.validator.engine.resolver.DefaultTraversableResolver</
traversable-resolver>
```

```
                                                                    <constraint-validator-
factory>org.hibernate.validator.engine.ConstraintValidatorFactoryImpl</constraint-validator-
factory>
    <constraint-mapping>META-INF/validation/constraints-car.xml</constraint-mapping>
    <property name="prop1">value1</property>
    <property name="prop2">value2</property>
</validation-config>
```

Warning

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

The node default-provider allows to choose the Bean Validation provider. This is useful if there is more than one provider on the classpath. message-interpolator, traversable-resolver and constraint-validator-factory allow to customize the used implementations for `javax.validation.MessageInterpolator`, `javax.validation.TraversableResolver` resp. `javax.validation.ConstraintValidatorFactory`.

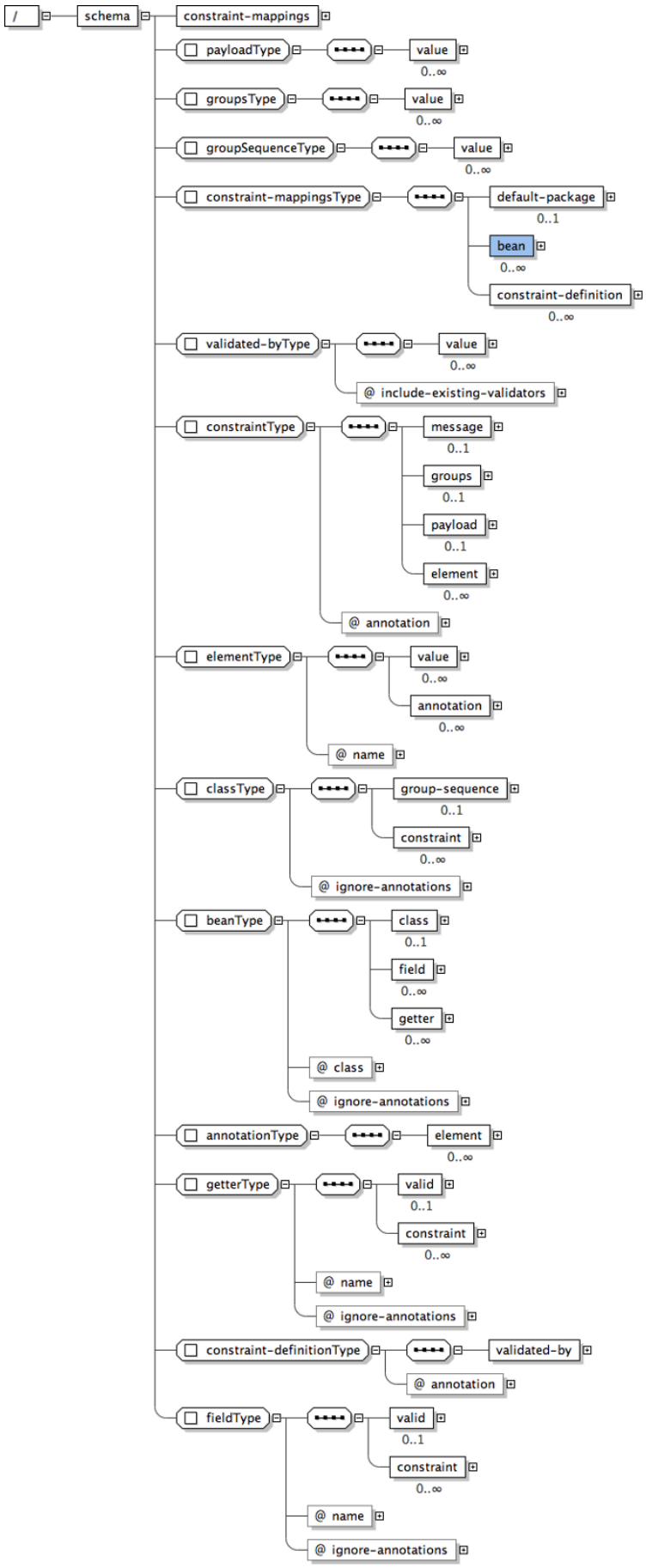
All these settings are optional. *Example 4.2, "validation.xml"* shows the defaults used within Hibernate Validator. The same configuration options are also available programmatically through `javax.validation.Configuration`. In fact XML configuration will be overridden by values explicitly specified via the API. It is even possible to ignore the XML configuration completely via `Configuration.ignoreXmlConfiguration()`. See also *Chapter 5, Bootstrapping*.

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.

4.2. Mapping constraints

Expressing constraints in XML is possible via files adhering to the schema seen in *Example 4.3, "validation-mapping-1.0.xsd"*. Note that these mapping files are only processed if listed via constraint-mapping in your `validation.xml`.


/
schema
constraint-mappings
payloadType
value
0..∞
groupsType
value
0..∞
groupSequenceType
value
0..∞
constraint-mappingsType
default-package
0..1
bean
0..∞
constraint-definition
0..∞
validated-byType
value
0..∞
@ include-existing-validators
constraintType
message
0..1
groups
0..1
payload
0..1
element
0..∞
@ annotation
elementType
value
0..∞
annotation
0..∞
@ name
classType
group-sequence
0..1
constraint
0..∞
@ ignore-annotations
beanType
class
0..1
field
0..∞
getter
0..∞
@ class
@ ignore-annotations
annotationType
element
0..∞
getterType
valid
0..1
constraint
0..∞
@ name
@ ignore-annotations
constraint-definitionType
validated-by
@ annotation
fieldType
valid
0..1
constraint
0..∞
@ name
@ ignore-annotations

Example 4.4, "constraints-car.xml" shows how our classes `Car` and `RentalCar` from *Example 2.15, "Car"* resp. *Example 2.19, "RentalCar with @GroupSequence"* could be mapped in XML.

Example 4.4. constraints-car.xml

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

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

The XML configuration is closely mirroring the programmatic API. For this reason it should suffice to just add some comments. default-package is used for all fields where a class name is expected.

If the specified class is not fully qualified the configured default package will be used. Every mapping file can then have several bean nodes, each describing the constraints on the entity with the specified class name.

Warning

A given entity can only be configured once across all configuration files. If the same class is configured more than once an exception 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 and getter. If not explicitly specified on these levels the configured bean value applies. Otherwise do the nodes class, fields and getter determine on which level the constraints are placed (see *Section 2.1, "Defining constraints"*). The constraint node is then used to add a constraint on the corresponding level. Each constraint definition must define the class via the annotation attribute. The constraint attributes required by the Bean Validation specification (message, groups and payload) have dedicated nodes. All other constraint specific attributes are configured using the element node.

The class node also allows to reconfigure the default group sequence (see *Section 2.3.2, "Redefining the default group sequence of a class"*) via the group-sequence node.

Last but not least, the list of `ConstraintValidators` associated to a given constraint can be altered via the constraint-definition node. The annotation attribute represents the constraint annotation being altered. The validated-by elements represent the (ordered) list of `ConstraintValidator` implementations associated to the constraint. If include-existing-validator is set to `false`, validators defined on the constraint annotation are ignored. If set to `true`, the list of constraint validators described in XML is concatenated to the list of validators specified on the annotation.

Bootstrapping

We already seen in *Section 5.1, "Configuration and ValidatorFactory"* the easiest way to create a `Validator` instance - `Validation.buildDefaultValidatorFactory`. In this chapter we have a look at the other methods in `javax.validation.Validation` and how they allow to configure several aspects of Bean Validation at bootstrapping time.

The different bootstrapping options allow, amongst other things, to bootstrap any Bean Validation implementation on the classpath. Generally, an available provider is discovered by the *Java Service Provider* [http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html#Service%20Provider] mechanism. A Bean Validation implementation includes the file `javax.validation.spi.ValidationProvider` in `META-INF/services`. This file contains the fully qualified classname of the `ValidationProvider` of the implementation. In the case of Hibernate Validator this is `org.hibernate.validator.HibernateValidator`.

Note

If there are more than one Bean Validation implementation providers in the classpath and `Validation.buildDefaultValidatorFactory()` is used, there is no guarantee which provider will be chosen. To enforce the provider `Validation.byProvider()` should be used.

5.1. `Configuration` and `ValidatorFactory`

There are three different methods in the Validation class to create a Validator instance. The easiest in shown in *Example 5.1, "Validation.buildDefaultValidatorFactory()"*.

Example 5.1. Validation.buildDefaultValidatorFactory()

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

You can also use the method `Validation.byDefaultProvider()` which will allow you to configure several aspects of the created Validator instance:

Example 5.2. Validation.byDefaultProvider()

```
Configuration<?> config = Validation.byDefaultProvider().configure();
config.messageInterpolator(new MyMessageInterpolator())
    .traversableResolver( new MyTraversableResolver())
    .constraintValidatorFactory(new MyConstraintValidatorFactory());
```

```
ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

We will learn more about `MessageInterpolator`, `TraversableResolver` and `ConstraintValidatorFactory` in the following sections.

Last but not least you can ask for a Configuration object of a specific Bean Validation provider. This is useful if you have more than one Bean Validation provider in your classpath. In this situation you can make an explicit choice about which implementation to use. In the case of Hibernate Validator the `Validator` creation looks like:

Example 5.3. Validation.byProvider(HibernateValidator.class)

```
HibernateValidatorConfiguration config = Validation.byProvider( HibernateValidator.class ).configure();
config.messageInterpolator(new MyMessageInterpolator())
    .traversableResolver( new MyTraversableResolver())
    .constraintValidatorFactory(new MyConstraintValidatorFactory());

ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

Tip

The generated `Validator` instance is thread safe and can be cached.

5.2. `ValidationProviderResolver`

In the case that the Java Service Provider mechanism does not work in your environment or you have a special classloader setup, you are able to provide a custom `ValidationProviderResolver`. An example in an OSGi environment you could plug your custom provider resolver like seen in *Example 5.4, "Providing a custom ValidationProviderResolver"*.

Example 5.4. Providing a custom ValidationProviderResolver

```
Configuration<?> config = Validation.byDefaultProvider()
    .providerResolver( new OSGiServiceDiscoverer() )
    .configure();

ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

Your `OSGiServiceDiscoverer` must in this case implement the interface `ValidationProviderResolver`:

Example 5.5. ValidationProviderResolver interface

```
public interface ValidationProviderResolver {
    /**
     * Returns a list of ValidationProviders available in the runtime environment.
     *
     * @return list of validation providers.
     */
    List<ValidationProvider<?>> getValidationProviders();
}
```

5.3. `MessageInterpolator`

Section 2.2.4, "Message interpolation" already discussed the default message interpolation algorithm. If you have special requirements for your message interpolation you can provide a custom interpolator using `Configuration.messageInterpolator()`. This message interpolator will be shared by all validators generated by the `ValidatorFactory` created from this `Configuration`. *Example 5.6, "Providing a custom MessageInterpolator"* shows an interpolator (available in Hibernate Validator) which can interpolate the value being validated in the constraint message. To refer to this value in the constraint message you can use:

- ${validatedValue}: this will call `String.valueOf` on the validated value.
- ${validatedValue:<format>}: provide your own format string which will be passed to `String.format` together with the validated value. Refer to the javadoc of `String.format` for more information about the format options.

Example 5.6. Providing a custom MessageInterpolator

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .messageInterpolator(new ValueFormatterMessageInterpolator(configuration.getDefaultMessageInterpolator()))
    .buildValidatorFactory();

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

Tip

It is recommended that `MessageInterpolator` implementations delegate final interpolation to the Bean Validation default `MessageInterpolator` to ensure standard Bean Validation interpolation rules are followed. The default implementation is accessible through `Configuration.getDefaultMessageInterpolator()`.

5.3.1. ResourceBundleLocator

A common use case is the ability to specify your own resource bundles for message interpolation. The default `MessageInterpolator` implementation in Hibernate Validator is called `ResourceBundleMessageInterpolator` and per default loads resource bundles via `ResourceBundle.getBundle`. However, `ResourceBundleMessageInterpolator` also allows you to specify a custom implementation of `ResourceBundleLocator` allowing you to provide your own resource bundles. *Example 5.7, "Providing a custom ResourceBundleLocator"* shows an example. In the example `HibernateValidatorConfiguration.getDefaultResourceBundleLocator` is used to retrieve the default `ResourceBundleLocator` which then can be passed to the custom implementation in order implement delegation.

Example 5.7. Providing a custom ResourceBundleLocator

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

ResourceBundleLocator defaultResourceBundleLocator = configure.getDefaultResourceBundleLocator();
ResourceBundleLocator myResourceBundleLocator = new MyCustomResourceBundleLocator(defaultResourceBundleLocator);

configure.messageInterpolator(new ResourceBundleMessageInterpolator(myResourceBundleLocator));
```

Hibernate Validator provides the following implementation of `ResourceBundleLocator` - `PlatformResourceBundleLocator` (the default) and `AggregateResourceBundleLocator`. The latter can be used to specify a list of resource bundle names which will get loaded and merged into a single resource bundle. Refer to the JavaDoc documentation for more information.

5.4. `TraversableResolver`

The usage of the `TraversableResolver` has so far not been discussed. The idea is that in some cases, the state of a property should not be accessed. The most obvious example for that is a lazy loaded property or association of a Java Persistence provider. Validating this lazy property or association would mean that its state would have to be accessed triggering a load from the database. Bean Validation controls which property can and cannot be accessed via the `TraversableResolver` interface (see *Example 5.8, "TraversableResolver interface"*). In the example HibernateValidatorConfiguration.

Example 5.8. TraversableResolver interface

```
/**
 * Contract determining if a property can be accessed by the Bean Validation provider
 * This contract is called for each property that is being either validated or cascaded.
 *
 * A traversable resolver implementation must be thread-safe.
 *
 */
public interface TraversableResolver {
    /**
```

```
     * Determine if the Bean Validation provider is allowed to reach the property state
     *
     * @param traversableObject object hosting <code>traversableProperty</code> or null
     *                          if validateValue is called
     * @param traversableProperty the traversable property.
     * @param rootBeanType type of the root object passed to the Validator.
     * @param pathToTraversableObject path from the root object to
     *        <code>traversableObject</code>
     *        (using the path specification defined by Bean Validator).
     * @param elementType either <code>FIELD</code> or <code>METHOD</code>.
     *
     * @return <code>true</code> if the Bean Validation provider is allowed to
     *         reach the property state, <code>false</code> otherwise.
     */
    boolean isReachable(Object traversableObject,
                        Path.Node traversableProperty,
                        Class<?> rootBeanType,
                        Path pathToTraversableObject,
                        ElementType elementType);

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

Hibernate Validator provides two `TraversableResolvers` out of the box which will be enabled automatically depending on your environment. The first is the `DefaultTraversableResolver` which will always return true for `isReachable()` and `isTraversable()`. The second is the `JPATraversableResolver` which gets enabled when Hibernate Validator gets used in combination with JPA 2. In case you have to provide your own resolver you can do so again using the `Configuration` object as seen in *Example 5.9, "Providing a custom TraversableResolver"*.

Example 5.9. Providing a custom TraversableResolver

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .traversableResolver(new MyTraversableResolver())
    .buildValidatorFactory();

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

5.5. `ConstraintValidatorFactory`

Last but not least, there is one more configuration option to discuss, the `ConstraintValidatorFactory`. The default `ConstraintValidatorFactory` provided by Hibernate Validator requires a public no-arg constructor to instantiate `ConstraintValidator` instances (see *Section 3.1.2, "The constraint validator"*). Using a custom `ConstraintValidatorFactory` offers for example the possibility to use dependency injection in constraint implementations. The configuration of the custom factory is once more via the `Configuration` (*Example 5.10, "Providing a custom ConstraintValidatorFactory"*).

Example 5.10. Providing a custom ConstraintValidatorFactory

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .constraintValidatorFactory(new IOCConstraintValidatorFactory())
    .buildValidatorFactory();

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

The interface you have to implement is:

Example 5.11. ConstraintValidatorFactory interface

```
public interface ConstraintValidatorFactory {
    /**
     * @param key The class of the constraint validator to instantiate.
     *
     * @return A constraint validator instance of the specified class.
     */
     <T extends ConstraintValidator<?,?>> T getInstance(Class<T> key);
}
```

Warning

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

Note

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

Metadata API

The Bean Validation specification provides not only a validation engine, but also a metadata repository for all defined constraints. The following paragraphs are discussing this API. All the introduced classes can be found in the javax.validation.metadata package.

6.1. `BeanDescriptor`

The entry into the metadata API is via `Validator.getConstraintsForClass` which returns an instance of the *`BeanDescriptor`* [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/BeanDescriptor.html] interface. Using this bean descriptor you can determine whether the specified class hosts any constraints at all via `beanDescriptor.isBeanConstrained`.

Tip

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

You can then call `beanDescriptor.getConstraintDescriptors` to get a set of `ConstraintDescriptors` representing all class level constraints.

If you are interested in property level constraints, you can call `beanDescriptor.getConstraintsForProperty` or `beanDescriptor.getConstrainedProperties` to get a single resp. set of `PropertyDescriptors` (see *Section 6.2, "PropertyDescriptor"*).

6.2. `PropertyDescriptor`

The *`PropertyDescriptor`* [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/PropertyDescriptor.html] interface extends the `ElementDescriptor` interface and represents constraints on properties of a class. The constraint can be declared on the attribute itself or on the getter of the attribute - provided Java Bean naming conventions are respected. A `PropertyDescriptor` adds `isCascaded` (returning `true` if the property is marked with `@Valid`) and `getPropertyName` to the `ElementDescriptor` functionality.

6.3. `ElementDescriptor`

The *`ElementDiscriptor`* [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/ElementDescriptor.html] interface is the common base class for `BeanDescriptor` and `PropertyDescriptor`. Next to the `hasConstraints` and `getConstraintDescriptors` methods it also offers access to the `ConstraintFinder` API which allows you to query the metadata API in a more fine grained way. For example you can restrict

your search to constraints described on fields or on getters or a given set of groups. Given an `ElementDescriptor` instance you just call `findConstraints` to retrieve a `ConstraintFinder` instance.

Example 6.1. Usage of ConstraintFinder

```
Validator validator = Validation.buildDefaultValidatorFactory().getValidator();
BeanDescriptor beanDescriptor = validator.getConstraintsForClass(Person.class);
PropertyDescriptor propertyDescriptor = beanDescriptor.getConstraintsForProperty("name");
Set<ConstraintDescriptor<?>> constraints = propertyDescriptor.findConstraints()
                                           .declaredOn(ElementType.METHOD)
                                           .unorderedAndMatchingGroups(Default.class)
                                           .lookingAt(Scope.LOCAL_ELEMENT)
                                           .getConstraintDescriptors();
```

Example 6.1, "Usage of ConstraintFinder" shows an example on how to use the `ConstraintFinder` API. Interesting are especially the restrictions `unorderedAndMatchingGroups` and `lookingAt(Scope.LOCAL_ELEMENT` [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/Scope.html]). The former allows to only return `ConstraintDescriptors` matching a specified set of groups wheras the latter allows to distinguish between constraint directly specified on the element (`Scope.LOCAL_ELEMENT`) or constraints belonging to the element but hosted anywhere in the class hierarchy (`Scope.HIERARCHY`).

Warning

Order is not respected by unorderedAndMatchingGroups, but group inheritance and inheritance via sequence are.

6.4. `ConstraintDescriptor`

Last but not least, the *`ConstraintDescriptor`* [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/ConstraintDescriptor.html] interface describes a single constraint together with its composing constraints. Via an instance of this interface you get access to the constraint annotation and its parameters, as well as the groups the constraint is supposed to be applied on. It also also you to access the pass-through constraint payload (see *Example 3.2, "Defining CheckCase constraint annotation"*).

Integration 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.

7.1. OSGi

The Hibernate Validator jar file is conform to the OSGi specification and can be used within any OSGi container. The following lists represent the packages imported and exported by Hibernate Validator. The classes within the exported packages are considered part of Hibernate Validator public API.

Tip

The Java Service Provider mechanism used by Bean Validation to automatically discover validation providers doesn't work in an OSGi environment. To solve this, you have to provide a custom `ValidationProviderResolver`. See *Section 5.2, "ValidationProviderResolver"*

Exported packages

- org.hibernate.validator
- org.hibernate.validator.cfg.*
- org.hibernate.validator.constraints.*
- org.hibernate.validator.group
- org.hibernate.validator.messageinterpolation
- org.hibernate.validator.resourceloading
- org.hibernate.validator.spi.*

Imported packages

- javax.persistence.*, [2.0.0,3.0.0), optional
- javax.validation.*, [1.0.0,2.0.0)
- javax.xml.*

- javax.el.*, [2.0.0,4.0.0)
- org.xml.sax.*
- org.jboss.logging.*, [3.1.0,4.0.0)
- com.fasterxml.classmate.*, 0.8.0
- org.joda.time.*, [1.6.0,2.0.0), optional
- org.jsoup.*, [1.5.2,2.0.0), optional

7.2. Database schema-level validation

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

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

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

7.3. ORM integration

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

Tip

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

7.3.1. Hibernate event-based validation

Hibernate Validator has a built-in Hibernate event listener - *`org.hibernate.cfg.beanvalidation.BeanValidationEventListener`* [http://fisheye.jboss.org/browse/Hibernate/core/trunk/annotations/src/main/java/org/hibernate/cfg/beanvalidation/BeanValidationEventListener.java] - which is part of Hibernate Annotations

(as of Hibernate 3.5.x). Whenever a `PreInsertEvent`, `PreUpdateEvent` or `PreDeleteEvent` occurs, the listener will verify all constraints of the entity instance and throw an exception if any constraint is violated. Per default objects will be checked before any inserts or updates are made by Hibernate. Pre deletion events will per default not trigger a validation. You can configure the groups to be validated per event type using the properties javax.persistence.validation.group.pre-persist, javax.persistence.validation.group.pre-update and javax.persistence.validation.group.pre-remove. The values of these properties are the comma-separated, fully specified class names of the groups to validate. *Example 7.1, "Manual configuration of BeanValidationEvenListener"* shows the default values for these properties. In this case they could also be omitted.

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

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

Note

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

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

Example 7.1. Manual configuration of `BeanValidationEvenListener`

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

7.3.2. JPA

If you are using JPA 2 and Hibernate Validator is in the classpath the JPA2 specification requires that Bean Validation gets enabled. The properties javax.persistence.validation.group.pre-persist, javax.persistence.validation.group.pre-update and javax.persistence.validation.group.pre-remove as described in *Section 7.3.1, "Hibernate event-based validation"* can in this case be configured in `persistence.xml`. `persistence.xml` also defines a node validation-mode which can be set to `AUTO`, `CALLBACK`, `NONE`. The default is `AUTO`.

In a JPA 1 you will have to create and register Hibernate Validator yourself. In case you are using Hibernate EntityManager you can add a customized version of the `BeanValidationEventListener` described in *Section 7.3.1, "Hibernate event-based validation"* to your project and register it manually.

7.4. Presentation layer validation

When working with JSF2 or JBoss Seam™ and Hibernate Validator (Bean Validation) is present in the runtime environment validation is triggered for every field in the application. *Example 7.2, "Usage of Bean Validation within JSF2"* shows an example of the f:validateBean tag in a JSF page. For more information refer to the Seam documentation or the JSF 2 specification.

Example 7.2. Usage of Bean Validation within JSF2

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

Tip

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

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

The default is:

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

Hibernate Validator Specifics

In the following sections we are having a closer look at some of the Hibernate Validator specific features (features which are not part of the Bean Validation specification). This includes the fail fast mode, the programmatic constraint configuration API and boolean composition of composing constraints.

Note

The features described in the following sections are not portable between Bean Validation providers/implementations.

8.1. Public API

Let's start, however, with a look at the public API of Hibernate Validator. *Table 8.1, "Hibernate Validator public API"* lists all packages belonging to this API and describes their purpose.

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.

Note

In the following table, when a package is public its not necessarily true for its nested packages.

Table 8.1. Hibernate Validator public API

Packages	Description
org.hibernate.validator	This package contains the classes used by the Bean Validation bootstrap mechanism (eg. validation provider, configuration class). For more details see *Chapter 5, Bootstrapping*.
org.hibernate.validator.cfg, org.hibernate.validator.cfg.context, org.hibernate.validator.cfg.defs	With Hibernate Validator you can define constraints via a fluent API. These packages contain all classes needed to use this feature. In the package org.hibernate.validator.cfg you will find the `ConstraintMapping` class and in package org.hibernate.validator.cfg.defs all constraint definitions. For more details see *Section 8.4, "Programmatic constraint definition"*.

Packages	Description
org.hibernate.validator.constraints, org.hibernate.validator.constraints.br	In addition to Bean Validation constraints, Hibernate Validator provides some useful custom constraints. These packages contain all custom annotation classes. For more details see *Section 2.4.2, "Additional constraints"*.
org.hibernate.validator.group, org.hibernate.validator.spi.group	With Hibernate Validator you can define dynamic default group sequences in function of the validated object state. These packages contain all classes needed to use this feature (`GroupSequenceProvider` annotation and `DefaultGroupSequenceProvider` contract). For more details see *Section 2.3.2, "Redefining the default group sequence of a class"*.
org.hibernate.validator.messageinterpolation, org.hibernate.validator.resourceloading, org.hibernate.validator.spi.resourceloading	These packages contain the classes related to constraint message interpolation. The first package contains two implementations of `MessageInterpolator`. The first one, `ValueFormatterMessageInterpolator` allows to interpolate the validated value into the constraint message, see *Section 5.3, "MessageInterpolator"*. The second implementation named `ResourceBundleMessageInterpolator` is the implementation used by default by Hibernate Validator. This implementation relies on a `ResourceBundleLocator`, see *Section 5.3.1, "ResourceBundleLocator"*. Hibernate Validator provides different `ResourceBundleLocator` implementations located in the package org.hibernate.validator.resourceloading.
org.hibernate.validator.method, org.hibernate.validator.method.metadata	Hibernate Validator provides support for method-level constraints based on appendix C of the Bean Validation specification. The first package contains the `MethodValidator` interface allowing you to validate method return values and parameters. The second package contains meta data for constraints hosted on parameters and methods which can be retrieved via the `MethodValidator`.

Note

The public packages of Hibernate Validator fall into two categories: while the actual API parts are intended to be *invoked* or *used* by clients (e.g. the API for programmatic constraint declaration or the custom constraints), the SPI (service provider interface) packages contain interfaces which are intended to be *implemented* by clients (e.g. `ResourceBundleLocator`).

8.2. Fail fast mode

First off, the fail fast mode. Hibernate Validator allows to return from the current validation as soon as the first constraint violation occurs. This is called the *fail fast mode* and can be useful for validation of large object graphs where one is only interested whether there is a constraint violation or not. *Example 8.1, "Enabling failFast via a property"*, *Example 8.2, "Enabling failFast at the Configuration level"* and *Example 8.3, "Enabling failFast at the ValidatorFactory level"* show multiple ways to enable the fail fast mode.

Example 8.1. Enabling `failFast` via a property

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

// do some actual fail fast validation
...
```

Example 8.2. Enabling `failFast` at the `Configuration` level

```
HibernateValidatorConfiguration                          configuration                          =
 Validation.byProvider( HibernateValidator.class ).configure();
ValidatorFactory factory = configuration.failFast( true ).buildValidatorFactory();
Validator validator = factory.getValidator();

// do some actual fail fast validation
...
```

Example 8.3. Enabling `failFast` at the `ValidatorFactory` level

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

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

```
// do some non fail fast validation
...

validator = factory.unwrap( HibernateValidatorFactory.class )
            .usingContext()
            .failFast( true )
            .getValidator();

// do fail fast validation
...
```

8.3. Method validation

The Bean Validation API allows to specify constraints for fields, properties and types. Hibernate Validator goes one step further and allows to place contraint annotations also on method parameters and method return values, thus enabling a programming style known as "Programming by Contract".

More specifically this means that Bean Validation constraints can be used to specify

- the preconditions that must be met before a method invocation (by annotating method parameters with constraints) and
- the postconditions that are guaranteed after a method invocation (by annotating methods)

This approach has several advantages over traditional ways of parameter and return value checking:

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

Note

Method validation was also considered to be included in the Bean Validation API as defined by JSR 303, but it didn't become part of the 1.0 version. A basic draft is outlined in appendix C of the specification, and the implementation in Hibernate Validator is largely influenced by this draft. The feature is considered again for inclusion in BV 1.1.

8.3.1. Defining method-level constraints

Example 8.4, "Using method-level constraints" demonstrates the definition of method-level constraints.

Example 8.4. Using method-level constraints

```
public class RentalStation {

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

Here the following pre- and postconditions for the `rentCar()` method are declared:

- The renting customer may not be null
- The rental's start date must not be null and must be in the future
- The rental duration must be at least one day
- The returned `Car` instance may not be null

Using the `@Valid` annotation it's also possible to define that a cascaded validation of parameter or return value objects shall be performed. An example can be found in *Example 8.5, "Cascaded validation of method-level constraints"*.

Example 8.5. Cascaded validation of method-level constraints

```
public class RentalStation {

    @Valid
    public Set<Rental> getRentalsByCustomer(@Valid Customer customer) {
        //...
    }
}
```

Here all the constraints declared at the `Customer` type will be evaluated when validating the method parameter and all constraints declared at the returned `Rental` objects will be evaluated when validating the method's return value.

8.3.1.1. Using method constraints in type hierarchies

Special care must be taken when defining parameter constraints in inheritance hierarchies.

When a method is overridden in sub-types method parameter constraints can only be declared at the base type. The reason for this restriction is that the preconditions to be fulfilled by a type's client must not be strengthened in sub-types (which may not even be known to the base type's client). Note that also if the base method doesn't declare any parameter constraints at all, no parameter constraints may be added in overriding methods.

The same restriction applies to interface methods: no parameter constraints may be defined at the implementing method (or the same method declared in sub-interfaces).

If a violation of this rule is detected by the validation engine, a `javax.validation.ConstraintDeclarationException` will be thrown. In *Example 8.6, "Illegal parameter constraint declarations"* some examples for illegal parameter constraints declarations are shown.

Example 8.6. Illegal parameter constraint declarations

```
public class Car {

    public void drive(Person driver) { ... }

}

public class RentalCar extends Car {

    //not allowed, parameter constraint added in overriding method
    public void drive(@NotNull Person driver) { ... }

}

public interface ICar {

    void drive(Person driver);

}

public class CarImpl implements ICar {

    //not allowed, parameter constraint added in implementation of interface method
    public void drive(@NotNull Person driver) { ... }

}
```

This rule only applies to parameter constraints, return value constraints may be added in sub-types without any restrictions as it is alright to strengthen the postconditions guaranteed to a type's client.

8.3.2. Evaluating method-level constraints

To validate method-level constraints Hibernate Validator provides the interface `org.hibernate.validator.method.MethodValidator`.

As shown in *Example 8.7, "The MethodValidator interface"* this interface defines methods for the evaluation of parameter as well as return value constraints and for retrieving an extended type descriptor providing method constraint related meta data.

Example 8.7. The `MethodValidator` interface

```
public interface MethodValidator {

     <T> Set<MethodConstraintViolation<T>> validateParameter(T object, Method method, Object
 parameterValue, int parameterIndex, Class<?>... groups);

   <T> Set<MethodConstraintViolation<T>> validateAllParameters(T object, Method method, Object[]
 parameterValues, Class<?>... groups);

    <T> Set<MethodConstraintViolation<T>> validateReturnValue(T object, Method method, Object
 returnValue, Class<?>... groups);

    TypeDescriptor getConstraintsForType(Class<?> clazz);
}
```

To retrieve a method validator get hold of an instance of HV's `javax.validation.Validator` implementation and unwrap it to `MethodValidator` as shown in *Example 8.8, "Retrieving a MethodValidator instance"*.

Example 8.8. Retrieving a `MethodValidator` instance

```
MethodValidator methodValidator = Validation.byProvider( HibernateValidator.class )
    .configure()
    .buildValidatorFactory()
    .getValidator()
    .unwrap( MethodValidator.class );
```

The validation methods defined on `MethodValidator` each return a `Set<MethodConstraintViolation>`. The type `MethodConstraintViolation` (see *Example 8.9, "The MethodConstraintViolation type"*) extends `javax.validation.ConstraintViolation` and provides additional method level validation specific information such as the method and index of the parameter which caused the constraint violation.

Example 8.9. The `MethodConstraintViolation` type

```
public interface MethodConstraintViolation<T> extends ConstraintViolation<T> {

    public static enum Kind { PARAMETER, RETURN_VALUE }

    Method getMethod();

    Integer getParameterIndex();
```

```
    String getParameterName();

    Kind getKind();
}
```

Note

The method `getParameterName()` currently returns synthetic parameter identifiers such as "arg0", "arg1" etc. In a future version of Hibernate Validator support for specifying parameter identifiers might be added.

Typically the validation of method-level constraints is not invoked manually but automatically upon method invocation by an integration layer using AOP (aspect-oriented programming) or similar method interception facilities such as the JDK's `java.lang.reflect.Proxy` API or CDI ("JSR 299: Contexts and Dependency Injection for the Java™ EE platform").

If a parameter or return value constraint can't be validated sucessfully such an integration layer typically will throw a `MethodConstraintViolationException` which similar to `javax.validation.ConstraintViolationException` contains a set with the occurred constraint violations.

Tip

If you are using CDI you might be interested in the *Seam Validation* [http://seamframework.org/Seam3/ValidationModule] project. This Seam module provides an interceptor which integrates the method validation functionality with CDI.

8.3.3. Retrieving method-level constraint meta data

As outlined in *Chapter 6, Metadata API* the Bean Validation API provides rich capabilities for retrieving constraint related meta data. Hibernate Validator extends this API and allows to retrieve constraint meta data also for method-level constraints.

Example 8.10, "Retrieving meta data for method-level constraints" shows how to use this extended API to retrieve constraint meta data for the `rentCar()` method from the `RentalStation` type.

Example 8.10. Retrieving meta data for method-level constraints

```
TypeDescriptor typeDescriptor = methodValidator.getConstraintsForType(RentalStation.class)

//retrieve a descriptor for the rentCar() method
MethodDescriptor     rentCarMethod     =     typeDescriptor.getConstraintsForMethod("rentCar",
 Customer.class, Date.class, int.class);
```

```
assertEquals(rentCarMethod.getMethodName(), "rentCar");
assertTrue(rentCarMethod.hasConstraints());
assertFalse(rentCarMethod.isCascaded());

//retrieve constraints from the return value
Set<ConstraintDescriptor<?>>                    returnValueConstraints                    =
 rentCarMethod.findConstraints().getConstraintDescriptors();
assertEquals(returnValueConstraints.size(), 1);
assertEquals(returnValueConstraints.iterator().next().getAnnotation().annotationType(),
 NotNull.class);

List<ParameterDescriptor> allParameters = rentCarMethod.getParameterDescriptors();
assertEquals(allParameters.size(), 3);

//retrieve a descriptor for the startDate parameter
ParameterDescriptor startDateParameter = allParameters.get(1);
assertEquals(startDateParameter.getIndex(), 1);
assertFalse(startDateParameter.isCascaded());
assertEquals(startDateParameter.findConstraints().getConstraintDescriptors().size(), 2);
```

Refer to the *JavaDoc* [http://docs.jboss.org/hibernate/validator/4.2/api/index.html?org/hibernate/validator/method/metadata/package-summary.html] of the package `org.hibernate.validator.method.metadata` for more details on the extended meta data API.

8.4. Programmatic constraint definition

Another addition to the Bean Validation specification is the ability to configure constraints via a fluent API. This API can be used exclusively or in combination with annotations and xml. If used in combination programmatic constraints are additive to constraints configured via the standard configuration capabilities.

The API is centered around the `ConstraintMapping` class which can be found in the package org.hibernate.validator.cfg. Starting with the instantiation of a new `ConstraintMapping`, constraints can be defined in a fluent manner as shown in *Example 8.11, "Programmatic constraint definition"*.

Example 8.11. Programmatic constraint definition

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .property( "manufacturer", FIELD )
        .constraint( new NotNullDef() )
    .property( "licensePlate", FIELD )
        .constraint( new NotNullDef() )
        .constraint( new SizeDef().min( 2 ).max( 14 ) )
    .property( "seatCount", FIELD )
        .constraint( new MinDef()value ( 2 ) )
.type( RentalCar.class )
    .property( "rentalStation", METHOD )
        .constraint( new NotNullDef() );
```

As you can see constraints can be configured on multiple classes and properties using method chaining. The constraint definition classes `NotNullDef`, `SizeDef` and `MinDef` 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.

For custom constraints you can either create your own definition classes extending `ConstraintDef` or you can use `GenericConstraintDef` as seen in *Example 8.12, "Programmatic constraint definition using createGeneric()"*.

Example 8.12. Programmatic constraint definition using `createGeneric()`

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .property( "licensePlate", FIELD )
      .constraint( new GenericConstraintDef<CheckCase.class>( CheckCase.class ).param( "value",
 CaseMode.UPPER ) );
```

Not only standard class- and property-level constraints but also method constraints can be configured using the API. As shown in *Example 8.13, "Programmatic definition of method constraints"* methods are identified by their name and their parameters (if there are any). Having selected a method, constraints can be placed on the method's parameters and/or return value.

Example 8.13. Programmatic definition of method constraints

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .method( "drive", String.class, Integer.class )
        .parameter( 0 )
            .constraint( new NotNullDef() )
            .constraint( new MinDef().value ( 1 ) )
        .parameter( 1 )
            .constraint( new NotNullDef() )
        .returnValue()
            .constraint( new NotNullDef() )
    .method( "check" )
        .returnValue()
            .constraint( new NotNullDef() );
```

Using the API it's also possible to mark properties, method parameters and method return values as cascading (equivalent to annotating them with `@Valid`). An example can be found in *Example 8.14, "Marking constraints for cascaded validation"*.

Example 8.14. Marking constraints for cascaded validation

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .property( "manufacturer", FIELD )
```

```
        .valid()
    .property( "licensePlate", METHOD )
        .valid()
    .method( "drive", String.class, Integer.class )
        .parameter( 0 )
            .valid()
        .parameter( 1 )
            .valid()
        .returnValue()
            .valid()
.type( RentalCar.class )
    .property( "rentalStation", METHOD )
        .valid();
```

Last but not least you can configure the default group sequence or the default group sequence provider of a type as shown in *Example 8.15, "Configuration of default group sequence and default group sequence provider"*.

Example 8.15. Configuration of default group sequence and default group sequence provider

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .defaultGroupSequence( Car.class, CarChecks.class )
.type( RentalCar.class )
    .defaultGroupSequenceProvider( RentalCarGroupSequenceProvider.class );
```

Once a `ConstraintMapping` is set up it has to be passed to the configuration. Since the programmatic API is not part of the official Bean Validation specification you need to get hold of a `HibernateValidatorConfiguration` instance as shown in *Example 8.16, "Creating a Hibernate Validator specific configuration"*.

Example 8.16. Creating a Hibernate Validator specific configuration

```
ConstraintMapping mapping = new ConstraintMapping();
// configure mapping instance

HibernateValidatorConfiguration                              config                              =
 Validation.byProvider( HibernateValidator.class ).configure();
config.addMapping( mapping );
ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

8.5. Boolean composition for constraint composition

As per Bean Validation specification the constraints of a composed constraint (see *Section 3.2, "Constraint composition"*) are all combined via a logical *AND*. This means all of the composing

constraints need to return `true` in order for an overall successful validation. Hibernate Validator offers an extension to this logical *AND* combination which 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 8.17, "OR composition of constraints"* shows how to build a composing constraint where only one of the constraints has to be successful in order to pass the validation. Either the validated string is all lowercased or it is between two and three characters long.

Example 8.17. OR composition of constraints

```
@ConstraintComposition(OR)
@Pattern(regexp = "[a-z]")
@Size(min = 2, max = 3)
@ReportAsSingleViolation
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = { })
public @interface PatternOrSize {
   public abstract String message() default "{PatternOrSize.message}";
   public abstract Class<?>[] groups() default { };
   public abstract Class<? extends Payload>[] payload() default { };
}
```

Tip

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

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 JavaBean 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.

Note

You can find the Hibernate Validator Annotation Processor as part of the distribution bundle on *Sourceforge* [http://sourceforge.net/projects/hibernate/files/hibernate-validator] or in the JBoss Maven Repository (see *Example 1.1, "Configuring the JBoss Maven repository"*) under the GAV org.hibernate:hibernate-validator-annotation-processor.

9.1. Prerequisites

The Hibernate Validator Annotation Processor is based on the "Pluggable Annotation Processing API" as defined by *JSR 269* [http://jcp.org/en/jsr/detail?id=269] which is part of the Java Platform since Java 6.

9.2. Features

As of Hibernate Validator 5.0.0.CR2 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

9.3. Options

The behavior of the Hibernate Validator Annotation Processor can be controlled using the *processor options* [http://java.sun.com/javase/6/docs/technotes/tools/windows/javac.html#options] listed in table*Table 9.1, "Hibernate Validator Annotation Processor options"*:

Table 9.1. Hibernate Validator Annotation Processor options

Option	Explanation
`diagnosticKind`	Controls how constraint problems are reported. Must be the string representation of one of the values from the enum `javax.tools.Diagnostic.Kind`, e.g. `WARNING`. A value of `ERROR` will cause compilation to halt whenever the AP detects a constraint problem. Defaults to `ERROR`.
`methodConstraintsSupported`	Controls whether constraints are allowed at methods of any kind. Must be set to `true` when working with method level constraints as supported by Hibernate Validator. Can be set to `false` to allow constraints only at JavaBeans getter methods as defined by the Bean Validation API. Defaults to `true`.
`verbose`	Controls whether detailed processing information shall be displayed or not, useful for debugging purposes. Must be either `true` or `false`. Defaults to `false`.

9.4. Using the Annotation Processor

This section shows in detail how to integrate the Hibernate Validator Annotation Processor into command line builds (javac, Ant, Maven) as well as IDE-based builds (Eclipse, IntelliJ IDEA, NetBeans).

9.4.1. Command line builds

9.4.1.1. javac

When compiling on the command line using *javac* [http://java.sun.com/javase/6/docs/technotes/guides/javac/index.html], specify the JAR hibernate-validator-annotation-processor-5.0.0.CR2.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 9.1. Using the annotation processor with javac

```
javac src/main/java/org/hibernate/validator/ap/demo/Car.java \
   -cp /path/to/validation-api-1.0.0.GA.jar \
   -processorpath /path/to/hibernate-validator-annotation-processor-5.0.0.CR2.jar
```

9.4.1.2. Apache Ant

Similar to directly working with javac, the annotation processor can be added as as compiler argument when invoking the *javac task* [http://ant.apache.org/manual/CoreTasks/javac.html] for *Apache Ant* [http://ant.apache.org/]:

Example 9.2. Using the annotation processor with Ant

```
<javac srcdir="src/main"
       destdir="build/classes"
       classpath="/path/to/validation-api-1.0.0.GA.jar">
       <compilerarg value="-processorpath" />
       <compilerarg value="/path/to/hibernate-validator-annotation-processor-5.0.0.CR2.jar"/>
</javac>
```

9.4.1.3. Maven

There are several options for integrating the annotation processor with *Apache Maven* [http://maven.apache.org/]. Generally it is sufficient to add the Hibernate Validator Annotation Processor as dependency to your project:

Example 9.3. Adding the HV Annotation Processor as dependency

```
...
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>5.0.0.CR2</version>
    <scope>compile</scope>
</dependency>
...
```

The processor will then be executed automatically by the compiler. This basically works, but comes with the disadavantage that in some cases messages from the annotation processor are not displayed (see *MCOMPILER-66* [http://jira.codehaus.org/browse/MCOMPILER-66]).

Another option is using the *Maven Annotation Plugin* [http://code.google.com/p/maven-annotation-plugin/]. At the time of this writing the plugin is not yet available in any of the well-known repositories. Therefore you have to add the project's own repository to your settings.xml or pom.xml:

Example 9.4. Adding the Maven Annotation Plugin repository

```
...
<pluginRepositories>
    <pluginRepository>
        <id>maven-annotation-plugin-repo</id>
        <url>http://maven-annotation-plugin.googlecode.com/svn/trunk/mavenrepo</url>
    </pluginRepository>
</pluginRepositories>
...
```

Now disable the standard annotation processing performed by the compiler plugin and configure the annotation plugin by specifying an execution and adding the Hibernate Validator Annotation Processor as plugin dependency (that way the AP is not visible on the project's actual classpath):

Example 9.5. Configuring the Maven Annotation Plugin

```
...
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
        <compilerArgument>-proc:none</compilerArgument>
    </configuration>
</plugin>
<plugin>
    <groupId>org.bsc.maven</groupId>
    <artifactId>maven-processor-plugin</artifactId>
    <version>1.3.4</version>
    <executions>
        <execution>
            <id>process</id>
            <goals>
                <goal>process</goal>
            </goals>
            <phase>process-sources</phase>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator-annotation-processor</artifactId>
            <version>5.0.0.CR2</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
</plugin>
...
```

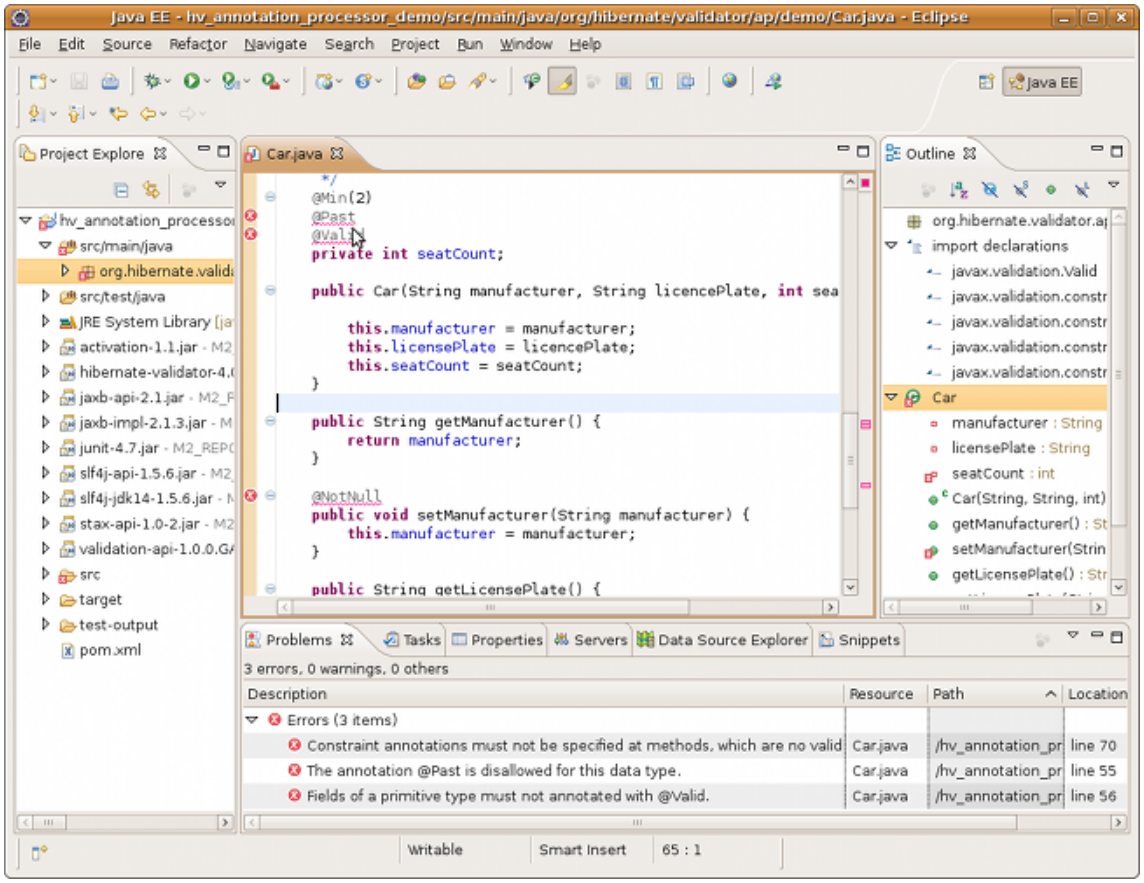
9.4.2. IDE builds

9.4.2.1. Eclipse

Do the following to use the annotation processor within the *Eclipse* [http://www.eclipse.org/] IDE:

- Right-click your project, choose "Properties"
- Go to "Java Compiler" and make sure, that "Compiler compliance level" is set to "1.6". Otherwise the processor won't be activated
- Go to "Java Compiler - Annotation Processing" and choose "Enable annotation processing"
- Go to "Java Compiler - Annotation Processing - Factory Path" and add the JAR hibernate-validator-annotation-processor-5.0.0.CR2.jar
- Confirm the workspace rebuild

You now should see any annotation problems as regular error markers within the editor and in the "Problem" view:

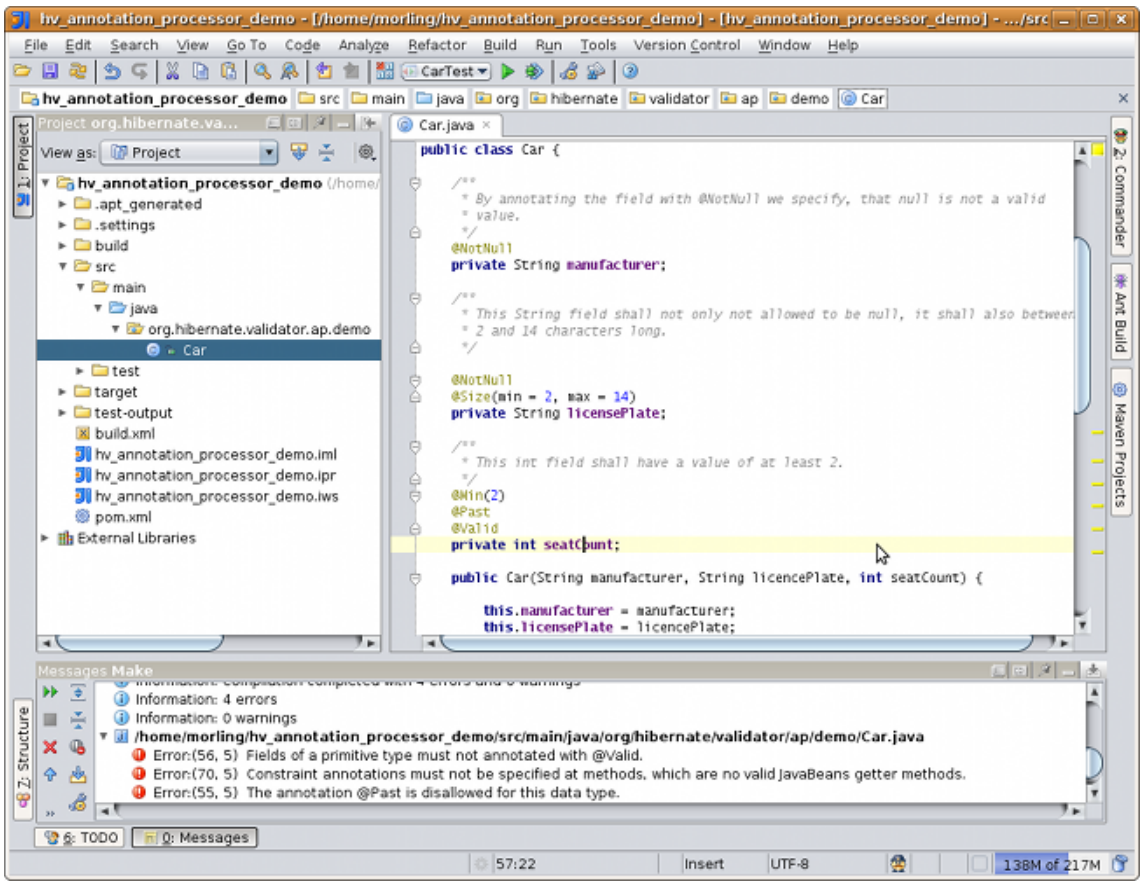


9.4.2.2. IntelliJ IDEA

The following steps must be followed to use the annotation processor within *IntelliJ IDEA* [http://www.jetbrains.com/idea/] (version 9 and above):

- Go to "File", then "Settings",
- Expand the node "Compiler", then "Annotation Processors"
- Choose "Enable annotation processing" and enter the following as "Processor path": /path/to/hibernate-validator-annotation-processor-5.0.0.CR2.jar
- Add the processor's fully qualified name `org.hibernate.validator.ap.ConstraintValidationProcessor` to the "Annotation Processors" list
- If applicable add you module to the "Processed Modules" list

Rebuilding your project then should show any erronous constraint annotations:



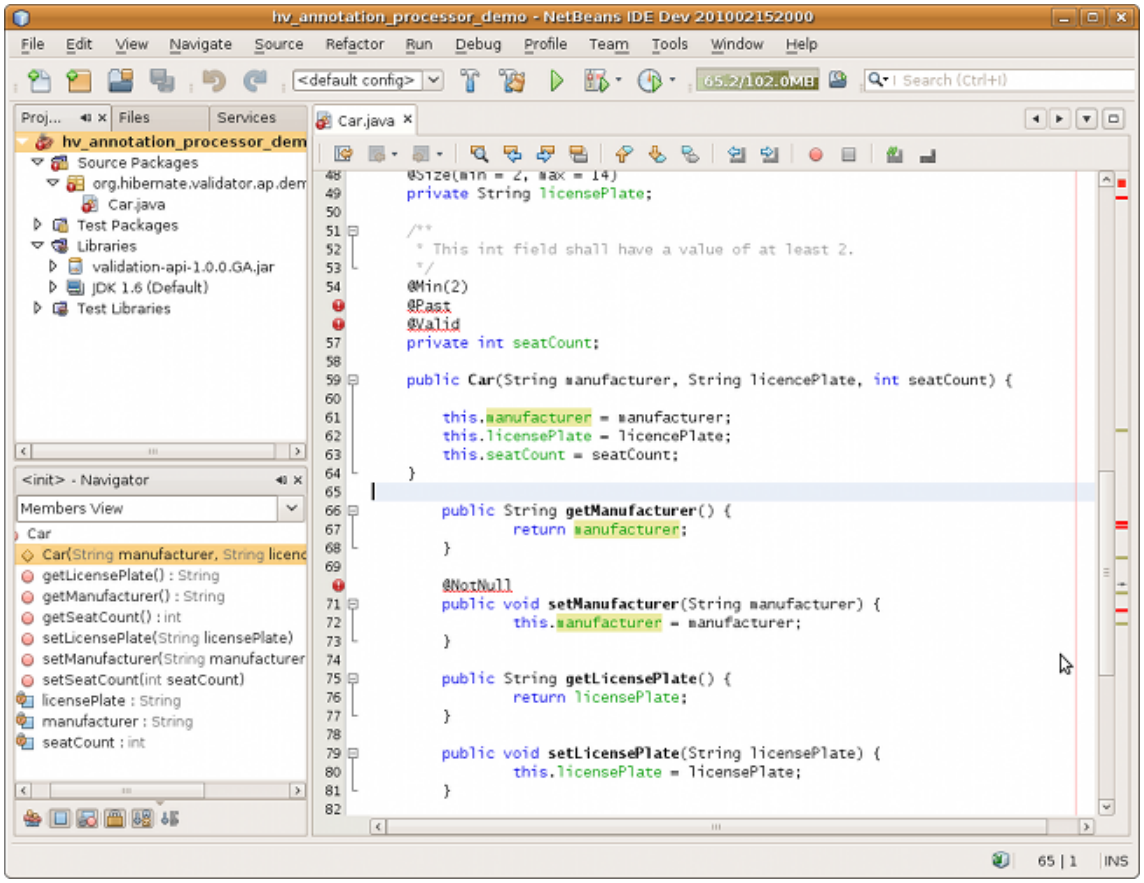
9.4.2.3. NetBeans

Starting with version 6.9, also the *NetBeans* [http://www.netbeans.org/] IDE supports using annotation processors within the IDE build. To do so, do the following:

- Right-click your project, choose "Properties"
- Go to "Libraries", tab "Processor", and add the JAR hibernate-validator-annotation-processor-5.0.0.CR2.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:



9.5. Known issues

The following known issues exist as of May 2010:

- *HV-308* [http://opensource.atlassian.com/projects/hibernate/browse/HV-308]: Additional validators registered for a constraint *using XML* [http://docs.jboss.org/hibernate/stable/validator/reference/en/html_single/#d0e1957] are not evaluated by the annotation processor.
- Sometimes custom constraints can't be *properly evaluated* [http://opensource.atlassian.com/projects/hibernate/browse/HV-293] when using the processor within Eclipse. Cleaning the project can help in these situations. This seems to be an issue with the Eclipse JSR 269 API implementation, but further investigation is required here.
- When using the processor within Eclipse, the check of dynamic default group sequence definitions doesn't work. After further investigation, it seems to be an issue with the Eclipse JSR 269 API implementation.

Further reading

Last but not least, a few pointers to further information.

A great source for examples is the Bean Validation TCK which is available for anonymous access on *GitHub* [https://github.com/beanvalidation/beanvalidation-tck]. In particular the TCK's *tests* [https://github.com/beanvalidation/beanvalidation-tck/tree/master/src/main/java/org/hibernate/beanvalidation/tck/tests] might be of interest. *The JSR 303* [http://jcp.org/en/jsr/detail?id=303] specification itself is also a great way to deepen your understanding of Bean Validation resp. Hibernate Validator.

If you have any further questions to Hibernate Validator or want to share some of your use cases have a look at the *Hibernate Validator Wiki* [http://community.jboss.org/en/hibernate/validator] and the *Hibernate Validator Forum* [https://forum.hibernate.org/viewforum.php?f=9].

In case you would like to report a bug use *Hibernate's Jira* [http://opensource.atlassian.com/projects/hibernate/browse/HV] instance. Feedback is always welcome!