JAVA ANNOTATIONS TUTORIAL

THE ULTIMATE GUIDE

Java™

HUGH HAMILL

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Java Annotations Tutorial

Contents

1 Overview **1**

2 Why annotations? **2**

3 Introduction **3**

4 Consumers **4**

5 Annotations syntax and annotation elements **5**

6 Where can be used **6**

7 Use cases **7**

8 Built in annotations **8**

9 Java 8 and annotations **10**

10 Custom annotations **13**

11 Retrieving Annotations **15**

12 Inheritance in annotations **17**

13 Known libraries using annotations **19**

13.1 Junit 19

13.2 Hibernate ORM 20

13.3 Spring MVC 21

13.4 Findbugs 22

13.5 JAXB 23

14 Summary **25**

15 Download **26**

16 Resources **27**

Preface

Annotations in Java are a major feature and every Java developer should know how to utilize them.

We have provided an abundance of tutorials here at Java Code Geeks, like Creating Your Own Java Annotations [1], Java Annotations Tutorial with Custom Annotation and Java Annotations: Explored & Explained.

We also featured articles on annotations used in various libraries, including Make your Spring Security @Secured annotations more DRY and Java Annotations & A Real World Spring Example.

Now, it is time to gather all the information around Annotations under one reference post for your reading pleasure. Enjoy!

References:

[1] http://www.javacodegeeks.com/2014/07/creating-your-own-java-annotations.html

[2] http://www.javacodegeeks.com/2012/11/java-annotations-tutorial-with-custom-annotation.html

[3] http://www.javacodegeeks.com/2012/08/java-annotations-explored-explained.html

[4] http://www.javacodegeeks.com/2012/06/make-your-spring-security-secured.html

[5] http://www.javacodegeeks.com/2012/01/java-annotations-real-world-spring.html

About the Author

Daniel Gutierrez Diez holds a Master in Computer Science Engineering from the University of Oviedo (Spain) and a Post Grade as Specialist in Foreign Trade from the UNED (Spain). Daniel has been working for different clients and companies in several Java projects as programmer, designer, trainer, consultant and technical lead.

Chapter 1

Overview

In this article we are going to explain what Java annotations are, how they work and what can be done using annotations in Java.

We will show what annotations come with Java out of the box, also called Built in or Meta annotations and what new features are available in Java 8 related to them.

Finally we will implement a custom annotation and a processor application (consumer) that makes use of annotations one using reflection in Java.

We will list some very well known and broadly used libraries based on annotations like Junit, JAXB, Spring and Hibernate.

At the end of this article you can find a compressed file with all the examples shown in this tutorial. In the implementation of these examples following software versions were used:

- Eclipse Luna 4.4
- JRE Update 8.20
- Junit 4
- Hibernate 4.3.6
- FindBugs 3.0.0

Chapter 2

Why annotations?

Annotations were introduced in Java in the J2SE update 5 already and the main reason was the need to provide a mechanism that allows programmers to write metadata about their code directly in the code itself.

Before annotations, the way programmers were describing their code was not standardized and each developer did it in his own original way: using transient keywords, via comments, with interfaces, etc. This was not a good approach and a decision was taken: a new form of metadata is going to be available in Java, annotations were introduced.

Other circumstances help to that decision: At that moment, XML was used as standard code configuration mechanism for different type of applications. This was not the best way to do it because of the decoupling between code and XML (XML is not code!) and the future maintenance of this decoupled applications. There were other reasons like for example the usage of the reserved word `@deprecated` (with small d) in the Javadocs since Java update 4, I am very sure that this was one of the reasons for the current annotations syntax using the `@`.

The main Java Specification Requests involved in the annotations design and development are:

- JSR 175 A metadata facility for the Java programming Language
- JSR 250 Common Annotations for the Java Platform

Chapter 3

Introduction

The best way to explain what an annotation is is the word metadata: data that contains information about itself. Annotations are code metadata; they contain information about the code itself.

Annotations can be used within packages, classes, methods, variables and parameters. Since Java 8 annotations can be placed almost in every place of the code, this is called type annotations; we will see this in this tutorial more in detail.

The annotated code is not directly affected by their annotations. These only provide information about it to third systems that may use (or not) the annotations for different purposes.

Annotations are compiled in the class files and can be retrieved in run time and used with some logical purpose by a consumer or processor. It is also possible to create annotations that are not available at runtime, even is possible to create annotations that are only available in the source and not at compilation time.

Chapter 4

Consumers

It can be difficult to understand what is actually the annotations purpose and what they can be used for, they do not contain any kind of functional logic and they do not affect the code that are annotating, what are they for?

The explanation to this is what I call the annotation consumers. These are systems or applications that are making use of the annotated code and executing different actions depending on the annotations information.

For example, in the case of the built in annotations (Meta annotations) that come out of the box with the standard Java, the consumer is the Java Virtual Machine (JVM) executing the annotated code. They are other examples that we are going to see later in this tutorial like Junit, where the consumer is the Junit processor reading and analyzing the annotated test classes, and deciding for example, depending on the annotation, in which order the unit tests are going to be executed or what methods are going to be executed before and after every test.

We will see this in more detail in the Junit related chapter.

Consumers use reflection in Java in order to read and analyze the annotated source code. The main packages used for this purpose are `java.lang` and `java.lang.reflect`. We will explain in this tutorial how to create a custom consumer from scratch using reflection.

Chapter 5

Annotations syntax and annotation elements

An annotation is declared using the character `@` as prefix of the annotation name. This indicates the compiler that this element is an annotation. An example:

```
@Annotation
public void annotatedMehod() {
...
 }
```

The annotation above is called "Annotation" and is annotating the method `annotatedMethod()`. The compiler will take care of it.

An annotation has elements in form of key-values. These "elements" are the properties of the annotation.

```
@Annotation(
   info = "I am an annotation",
   counter = "55"
)
public void annotatedMehod() {
...
 }
```

If the annotation contains only one element (or if only one element needs to be specified because the rest have default values), we can do something like:

```
@Annotation("I am an annotation")
public void annotatedMehod() {
...
 }
```

As we saw in the first example, if no elements need to be specified, then the parentheses are not needed.

Multiple annotations are possible for an element, in this case for a class:

```
@ Annotation (info = "U a u O")
@ Annotation2
class AnnotatedClass { ... }
```

Some annotations come out of the box with Java; these are called built in annotations. It is also possible to define your own annotations, these are called custom annotations. We will see these in the next chapters.

Chapter 6

Where can be used

Annotations can be used basically in almost every element of a Java program: classes, fields, methods, packages, variables, etc.

Since Java 8 the concept of annotations by type is available. Before Java 8 annotations should be used only in the declarations of the elements listed before. After Java 8 also in declaration of types an annotation can be used. Something like the following is now available:

```
@MyAnnotation String str = "danibuiza";
```

We will see this mechanism in more detail in the chapter related to Java 8 annotations.

Chapter 7

Use cases

Annotations can be used for many different purposes, the most common ones are:

- **Information for the compiler:** Annotations can be used by the compiler to produce warnings or even errors based on different rules. One example of this kind of usage is the Java 8 `@FunctionalInterface` annotation. This one makes the compiler to validate the annotated class and check if it is a correct functional interface or not.
- **Documentation:** Annotations can be used by software applications to measure the quality of the code like FindBugs or PMD do or generate reports automatically like Jenkins, Jira or Teamcity.
- **Code generation:** annotations can be used to generate code or XML files automatically using metadata information present in the code. A good example of this is the JAXB library.
- **Runtime processing:** Annotations that are examined in runtime can be used for different objectives like unit testing (Junit), dependency injection (Spring), validation, logging (Log4J) ,data access (Hibernate) etc.

In this tutorial we will show several possible usages of annotations and we will show how very well known Java libraries are using them

Chapter 8

Built in annotations

The Java language comes with a set of default annotations. In this chapter we are going to explain the most important ones. It is important to mention that this list refers only to the core packages of the Java language and does not include all the packages and libraries available in the standard JRE like JAXB or the Servlet specification

Some of the following standard annotations are called Meta annotations; their targets are other annotations and contain information about them:

- `@Retention`: This annotation annotates other annotations and it is used to indicate how to store the marked annotation. This annotation is a kind of Meta annotation, since it is marking an annotation and informing about its nature. Possible values are:
  - `SOURCE`: Indicates that this annotation is ignored by compiler and JVM (no available at run time) and it is only retained in the source.
  - `CLASS`: Indicates that the annotation is going to be retained by the compiler but ignored by the JVM and because of this, not going to be available at run time.
  - `RUNTIME`: Means that the annotation is going to be retained by the Java Virtual Machine and can be used in runtime via reflection.

We will see several examples of this annotation in this tutorial.

- `@Target`: This one restricts the elements that an annotation can be applied to. Only one type is possible. Here is a list of available types:
  - `ANNOTATION_TYPE` means that the annotation can be applied to other annotation.
  - `CONSTRUCTOR` can be applied to a constructor.
  - `FIELD` can be applied to a field or property.
  - `LOCAL_VARIABLE` can be applied to a local variable.
  - `METHOD` can be applied to a method-level annotation.
  - `PACKAGE` can be applied to a package declaration.
  - `PARAMETER` can be applied to the parameters of a method.
  - `TYPE` can be applied to any element of a class.
- `@Documented`: The annotated elements are going to be documented using the Javadoc tool. Per default annotations are not documented. This annotation can be applied to other annotation.
- `@Inherited`: By default annotations are not inherited by subclasses. This annotation marks an annotation to automatic inherit to all subclasses extending the annotated class. This annotation can be applied to class elements.
- `@Deprecated`: Indicates that the annotated element should not be used. This annotation gets the compiler to generate a warning message. Can be applied to methods, classes and fields. An explanation about why this element is deprecated and alternative usages should be provided when using this annotation.

- `@SuppressWarnings`: Indicates the compiler not to produce warnings for an specific reason or reasons. For example if we do not want to get warnings because of unused private methods we can write something like:

```
@SuppressWarnings( "unused")
private String myNotUsedMethod(){
        ...
}
```

Normally the compiler would produce a warning if this method is not used; using this annotation prevents that behavior. This annotation expects one or more parameters with the warnings categories to avoid.

- `@Override`: Indicates the compiler that the element is overriding an element of the super class. This annotation is not mandatory to use when overriding elements but it helps the compiler to generate errors when the overriding is not done correctly, for example if the sub class method parameters are different than the super class ones, or if the return type does not match.
- `@SafeVarargs`: Asserts that the code of the method or constructor does not perform unsafe operations on its arguments. Future versions of the Java language will make the compiler to produce an error at compilation time in case of potential unsafe operations while using this annotation. For more information about this one one SafeVarargs

Chapter 9

Java 8 and annotations

Java 8 comes out with several advantages. Also the annotations framework is improved. In this chapter we are going to explain and provide examples of the 3 main topics introduced in the eighth Java update: the `@Repeatable` annotation, the introduction of type annotation declarations and the functional interface annotation `@FunctionalInterface` (used in combination with Lambdas).

- `@Repeatable`: indicates that an annotation annotated with this one can be applied more than once to the same element declaration.

Here is an example of usage. First of all we create a container for the annotation that is going to be repeated or that can be repeated:

```
/**
 * Container for the {@link CanBeRepeated} Annotation containing a list of values
*/
@Retention( RetentionPolicy.RUNTIME )
@Target( ElementType.TYPE_USE )
public @interface RepeatedValues
{
    CanBeRepeated[] value();
}
```

Afterwards, we create the annotation itself and we mark it with the Meta annotation `@Repeatable`:

```
@Retention( RetentionPolicy.RUNTIME )
@Target( ElementType.TYPE_USE )
@Repeatable( RepeatedValues.class )
public @interface CanBeRepeated
{

    String value();
}
```

Finally we can see how to use it (repeatedly) in a given class:

```
@CanBeRepeated( "the color is green" )
@CanBeRepeated( "the color is red" )
@CanBeRepeated( "the color is blue" )
public class RepeatableAnnotated
{

}
```

If we would try to do the same with a non repeatable annotation:

```
@Retention( RetentionPolicy.RUNTIME )
@Target( ElementType.TYPE_USE )
public @interface CannotBeRepeated
{

    String value();
}

@CannotBeRepeated( "info" )
/*
 * if we try repeat the annotation we will get an error: Duplicate annotation of non- ↩
    repeatable type
 *
 * @CannotBeRepeated. Only annotation types marked
 *
 * @Repeatable can be used multiple times at one target.
 */
// @CannotBeRepeated( "more info" )
public class RepeatableAnnotatedWrong
{

}
```

We would get an error from the compiler like:

```
Duplicate annotation of non-repeatable type
```

- Since Java 8 is also possible to use annotations within types. That is anywhere you can use a type, including the new operator, castings, implements and throws clauses. Type Annotations allow improved analysis of Java code and can ensure even stronger type checking. The following examples clarify this point:

```
@SuppressWarnings( "unused" )
public static void main( String[] args )
{
        // type def
        @TypeAnnotated
        String cannotBeEmpty = null;

        // type
        List<@TypeAnnotated String> myList = new ArrayList<String>();

        // values
        String myString = new @TypeAnnotated String( "this is annotated in java 8" );

}

// in method params
public void methodAnnotated( @TypeAnnotated int parameter )
{
        System.out.println( "do nothing" );
}
```

All this was not possible until Java 8.

- `@FunctionalInterface`: this annotation indicates that the element annotated is going to be a functional interface. A functional interface is an interface that has just one abstract method (not a default one). The compiler will handle the annotated element as a functional interface and will produce an error if the element does not comply with the needed requirements. Here is an example of functional interface annotation:

```
// implementing its methods
@SuppressWarnings( "unused" )
MyCustomInterface myFuncInterface = new MyCustomInterface()
{

        @Override
        public int doSomething( int param )
        {
                return param * 10;
        }
};

// using lambdas
@SuppressWarnings( "unused" )
        MyCustomInterface myFuncInterfaceLambdas = ( x ) -> ( x * 10 );
}

@FunctionalInterface
interface MyCustomInterface
{
/*
 * more abstract methods will cause the interface not to be a valid functional interface  ↩
    and
 * the compiler will thrown an error:Invalid '@FunctionalInterface' annotation;
 * FunctionalInterfaceAnnotation.MyCustomInterface is not a functional interface
 */
        // boolean isFunctionalInterface();

        int doSomething( int param );
}
```

This annotation can be applied to classes, interfaces, enums and annotations and it is retained by the JVM and available in Run time. Here is its declaration:

```
@Documented
 @Retention(value=RUNTIME)
 @Target(value=TYPE)
public @interface FunctionalInterface
```

For more information about this annotation http://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html.

Chapter 10

Custom annotations

As we mentioned several times in this tutorial, it is possible to define and implement custom annotations. In this chapter we are going to show how to do this.

First of all, define the new annotation:

```
public @interface CustomAnnotationClass
```

This creates a new type of annotation called CustomAnnotationClass. The special word used for this purpose is `@interface`, this indicates the definition of a custom annotation.

After this, you need to define a couple of mandatory attributes for this annotation, the retention policy and the target. There are other attributes that can be defined here, but these are the most common and important ones. These are declared in form of annotations of an annotation and were described in the chapter "Built in annotations" since they are annotations that came out of the box with Java.

So we define these properties for our new custom annotation:

```
@Retention( RetentionPolicy.RUNTIME )
@Target( ElementType.TYPE )
public @interface CustomAnnotationClass implements CustomAnnotationMethod
```

With the retention policy `RUNTIME` we indicate to the compiler that this annotation should be retained by the JVM and can be analyzed in runtime using reflection. With the element type `TYPE` we are indicating that this annotation can be applied to any element of a class.

Afterwards we define a couple of properties for this annotation:

```
@Retention( RetentionPolicy.RUNTIME )
@Target( ElementType.TYPE )
public @interface CustomAnnotationClass
{

    public String author() default "danibuiza";

    public String date();

}
```

Above we have just defined the property author, with the default value "danibuiza" and the property date, without default value. We should mention that all method declarations cannot have parameters and are not allowed to have a thrown clause. The return types are restricted to String, Class, enums, annotations and arrays of the types mentioned before.

Now we can use our fresh created custom annotation in the following way:

```
@CustomAnnotationClass( date = "2014-05-05" )
public class AnnotatedClass
{
...
}
```

In a similar way we can create an annotation to be used in method declarations, using the target `METHOD`:

```
@Retention( RetentionPolicy.RUNTIME )
@Target( ElementType.METHOD )
public @interface CustomAnnotationMethod
{

    public String author() default "danibuiza";

    public String date();

    public String description();

}
```

This one can be used in a method declaration like:

```
@CustomAnnotationMethod( date = "2014-06-05", description = "annotated method" )
public String annotatedMethod()
    {
        return "nothing niente";
}

@CustomAnnotationMethod( author = "friend of mine", date = "2014-06-05", description = " ↩
    annotated method" )
public String annotatedMethodFromAFriend()
{
        return "nothing niente";
}
```

There are many other properties that can be used with custom annotations, but target and retention policy are the most important ones.

Chapter 11

Retrieving Annotations

The Java reflection API contains several methods that can be used to retrieve in runtime annotations from classes, methods and other elements.

The interface that contains all these methods is AnnotatedElement and the most important ones are:

- `getAnnotations()`: Returns all annotations for the given element, also the ones that are not explicitly defined in the element definition.
- `isAnnotationPresent(annotation)`: Checks if the passed annotation in available or not in the current element.
- `getAnnotation(class)`: Retrieves an specific annotation passed as parameter. Returns null if this annotation is not present for the given element.

This class is implementing by `java.lang.Class`, `java.lang.reflect.Method` and `java.lang.reflect.Field` among others, so can be used basically with any kind of Java element.

Now we are going to see an example of how to read the annotations present in a class or method using the methods listed above:

We write a program that tries to read all the annotations present in a class and its methods (we use for this example the classes defined before):

```
public static void main( String[] args ) throws Exception
{

        Class<AnnotatedClass> object = AnnotatedClass.class;
        // Retrieve all annotations from the class
        Annotation[] annotations = object.getAnnotations();
        for( Annotation annotation : annotations )
        {
                System.out.println( annotation );
        }

        // Checks if an annotation is present
        if( object.isAnnotationPresent( CustomAnnotationClass.class ) )
        {

                // Gets the desired annotation
                Annotation annotation = object.getAnnotation( CustomAnnotationClass.class ) ↩
                    ;

                System.out.println( annotation );

        }
        // the same for all methods of the class
        for( Method method : object.getDeclaredMethods() )
```

```
        {

                if( method.isAnnotationPresent( CustomAnnotationMethod.class ) )
                {

                        Annotation annotation = method.getAnnotation(  ↩
                            CustomAnnotationMethod.class );

                        System.out.println( annotation );

                }

        }
}
```

The output of this program would be:

```
@com.danibuiza.javacodegeeks.customannotations.CustomAnnotationClass(getInfo=Info, author= ↩
    danibuiza, date=2014-05-05)

@com.danibuiza.javacodegeeks.customannotations.CustomAnnotationClass(getInfo=Info, author= ↩
    danibuiza, date=2014-05-05)

@com.danibuiza.javacodegeeks.customannotations.CustomAnnotationMethod(author=friend of mine ↩
    , date=2014-06-05, description=annotated method)
@com.danibuiza.javacodegeeks.customannotations.CustomAnnotationMethod(author=danibuiza,  ↩
    date=2014-06-05, description=annotated method)
```

In the program above we can see the usage of the method `getAnnotations()` in order to retrieve all annotations for a given object (a method or a class). We also showed how to check if an specific annotation is present and to retrieve it in positive case using the methods `isAnnotationPresent()` and `getAnnotation()`.

Chapter 12

Inheritance in annotations

Annotations can use inheritance in Java. This inheritance has nothing or almost nothing in common with what we know by inheritance in an object oriented programming language where an inherited class inherits methods, elements and behaviors from his super class or interface.

If an annotation is marked as inherited in Java, using the reserved annotation `@Inherited`, indicates that the class that is annotated will pass this annotation to its subclasses automatically without need to declare the annotation in the sub classes. By default, a class extending a super class does not inherit its annotations. This is completely in line with the objective of annotations, which is to provide information about the code that they are annotating and not to modify its behavior.

We are going to see this using an example that makes thing clearer. First, we define a custom annotation that uses inheritance automatically

```
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface InheritedAnnotation
{

}
```

We have a super class called `AnnotatedSuperClass` with the `@InheritedAnnotation` annotation declared above:

```
@InheritedAnnotation
public class AnnotatedSuperClass
{

    public void oneMethod()
    {

    }

}
```

A subclass extending this one:

```
public class AnnotatedSubClass extends AnnotatedSuperClass
{

    @Override
    public void oneMethod(){

    }

}
```

The sub class `AnnotatedSubClass` shown above will automatically inherit the annotation `@InheritedAnnotation`. We can see this using the following test with the method `isAnnotationPresent()` present in the every class:

```
System.out.println( "is true: " + AnnotatedSuperClass.class.isAnnotationPresent(  ↩
    InheritedAnnotation.class ) );

System.out.println( "is true: " + AnnotatedSubClass.class.isAnnotationPresent(  ↩
    InheritedAnnotation.class ) );
```

The output to these lines is:

```
is true: true
is true: true
```

We can see how the annotation is inherited by the sub class automatically with no need of declaring it.

If we try to use this kind of annotation in an interface like this:

```
@InheritedAnnotation
public interface AnnotatedInterface
{

    public void oneMethod();

}
```

An implementation for it:

```
public class AnnotatedImplementedClass implements AnnotatedInterface
{

    @Override
    public void oneMethod()
    {

    }

}
```

And we check the result of the annotation inheritance using the `isAnnotationPresent()` method:

```
System.out.println( "is true: " + AnnotatedInterface.class.isAnnotationPresent(  ↩
    InheritedAnnotation.class ) );

System.out.println( "is true: " + AnnotatedImplementedClass.class.isAnnotationPresent(  ↩
    InheritedAnnotation.class ) );
```

The result of the previous program will be

```
is true: true
is true: false
```

This shows how inheritance works in relation with annotations and interfaces: it is just ignored. The implementing class do not inherit the annotation although it is an inherit annotation; it only applies to classes like in the case of the `AnnotatedSubClass` class above.

The `@Inherited` annotation is only applicable to classes and annotations present in the interfaces have no effect in the implementing classes. The same happens with methods, variables, packages, etc. Only classes can be used in conjunction with this annotation. A very good explanation can be found in the Javadoc of the `@Inherited` annotation: http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Inherited.html.

Annotations cannot inherit from other annotations; if you try to do this you will get a compilation error:

```
Annotation type declaration cannot have explicit superinterfaces
```

Chapter 13

Known libraries using annotations

In this chapter we are going to show how very well known Java libraries make usage of annotations. Several libraries like JAXB, Spring Framework, Findbugs, Log4j, Hibernate, and Junit... (The list can be infinite) use them for several different things like code quality analysis, unit testing, XML parsing, dependency injection and many others.

In this tutorial we are going to show some of these use cases:

13.1 Junit

This framework is used for unit testing in Java. Since its version 4 annotations are widely used and are one of the pillars of Junit design.

Basically the Junit processor reads using reflection the classes and suites that may contain unit tests and execute them depending on the annotations found at the beginning of each method or class. There are Junit annotations that modify the way a test is executed; others are used exactly for test execution, for prevention of execution, changing order of execution, etc.

The list of possible annotations is very large but we are going to see here the most important ones:

- `@Test`: This annotation indicates Junit that the annotated method has to be executed as unit test. It is applicable for methods only (using the target element type `METHOD`) and is retained in runtime by the Java Virtual Machine (using the retention policy RUNTIME).

```
@Test
public void testMe()
{
        //test assertions
        assertEquals(1,1);
}
```

In the example above we can see how to use this kind of annotation in Junit.

- `@Before`: the before annotation is used to indicate Junit that the marked method should be executed before every test. This is very useful for set up methods where test context are initialized. Is applicable to methods only:

```
@Before
public void setUp()
 {
        // initializing variables
        count = 0;
        init();

}
```

- `@After`: This annotation is used to indicate the Junit processor that all marked methods should be executed after every unit test. This annotation is normally used for destroying, closing or finalizing methods where resources are cleared and reset:

```
@After
public void destroy()
{
        // closing input stream
        stream.close();
}
```

- `@Ignore`: This one indicates Junit that the methods marked should not be executed as unit test. Even if they are annotated as test. They should be just ignored:

```
@Ignore
@Test
public void donotTestMe()
{
        count = -22;
        System.out.println( "donotTestMe(): " + count );
}
```

This method should be used during development and debugging phases but it is not common to leave ignored tests once the code is ready to go to production.

- `@FixMethodOrder`: Indicates what order of execution should be used, normally the Junit processor takes care of this and the default execution order is completely unknown and random for the programmer. This annotation is not really recommended since Junit methods and tests should be completely independent from each other and the order of execution should not affect the results. However, there are cases and scenarios where the order of unit tests should follow some rules where this annotation may be very useful.

```
@FixMethodOrder( MethodSorters.NAME_ASCENDING )
public class JunitAnnotated
```

There are other test suites and libraries making use of annotations like Mockito or JMock where annotations are used for the creation of test objects and methods expectations.

For a complete list of available annotations in Junit https://github.com/junit-team/junit/wiki/Getting-started

13.2 Hibernate ORM

Hibernate is probably the most used library for object relational mapping in Java. it provides a framework for mapping object model and relational databases. It makes use of annotations as part of its design.

In this chapter we are going to see a couple of annotations provided by Hibernate and to explain how its processor handles them.

The snippet bellow has the annotations `@Entity` and `@Table`. These are used to indicate the consumer (the Hibernate processor) that the annotated class is an entity bean and indicates what SQL table should be used for the objects of this class. Actually this annotation just explains which one is the primary table; there are annotations for the secondary as well.

```
@Entity
@Table( name = "hibernate_annotated" )
public class HibernateAnnotated
```

In the following piece of code we show how to indicate the Hibernate processor that the marked element is a table id with the name "id" and that it should be auto generated (the typical auto increment SQL ids):

```
@Id
@GeneratedValue
@Column( name = "id" )
private int    id;
```

In order to specify a standard SQL table column we can write something like this before an element:

```
@Column( name = "description" )
private String description;
```

This indicates that the marked element is a column with the name "description" in the table specified at the beginning of the class.

These annotations belong to the http://docs.oracle.com/javaee/6/api/javax/persistence/package-summary.html package from the Java Enterprise Edition and basically it covers all the available annotations that hibernate uses (or at least the most common ones).

13.3 Spring MVC

Spring is a framework widely used for implementing Java Enterprise applications. One of its most important features is the usage of dependency injection in Java programs.

Spring uses annotations as an alternative to XML based configuration (the first Spring versions used only XML based configurations). Currently both options are available; you can configure your projects using annotations and XML configuration files. In my opinion both approaches have benefits and inconveniences.

Here we are just going to show two of the multiple annotations available in Spring. In the following example:

```
@Component
public class DependencyInjectionAnnotation
{

    private String description;

    public String getDescription()
    {
        return description;
    }

    @Autowired
    public void setDescription( String description )
    {
        this.description = description;
    }

}
```

In the snippet above we can find two kinds of annotations applied to the whole class and to a method respectively:

- `@Component`: Indicates that the element marked by this annotation, in this case a class, is a candidate for auto detection. This means that the annotated class may be a bean and should be taken into consideration by the Spring container.
- `@Autowired`: The Spring container will try to perform byType auto wiring (this is a kind of property matching using the elements type) on this setter method. It can be applied to constructor and properties as well and the actions the Spring container takes in these cases are different.

For more information about dependency injection and the Spring framework in general please visit: http://projects.spring.io/spring-framework/.

13.4 Findbugs

This library is used in order to measure the quality of the code and provide a list of possibilities to improve it. It checks the code against a list of predefined (or customized) quality violations. Findbugs provide a list of annotations that allow programmers to change its default behavior.

Findbugs mainly reads the code (and its contained annotations) using reflection and decides what actions should be taken depending on them.

One example is the annotation `edu.umd.cs.findbugs.annotations.SuppressFBWarnings` that expects a violation key as parameter (one or more values as parameter have to be provided, no need for the key, since it is the default "value" key). It is very similar to the `java.lang.SuppressWarnings` one. It is used to indicate the Findbugs processor to ignore specific violations while executing the code analysis.

Here is an example of use:

```
@SuppressFBWarnings( "HE_EQUALS_USE_HASHCODE" )
public class FindBugsAnnotated
{

    @Override
    public boolean equals( Object arg0 )
    {
        return super.equals( arg0 );
    }

}
```

The class above overrides the `equals()` method of the `Object` class but does not do the same with the `hashCode()` method. This is normally a problem because `hashCode()` and `equals()` should be override both of them in order not to have problems while using the element as key in a `HashMap` for example. So Findbugs will create an error entry in the violations report for it.

If the annotation `@SuppressFBWarnings` with the value `HE_EQUALS_USE_HASHCODE` would not be there the FindBugs processor would throw an error of the type:

```
Bug: com.danibuiza.javacodegeeks.findbugsannotations.FindBugsAnnotated defines equals and ↩
    uses Object.hashCode()
Bug: com.danibuiza.javacodegeeks.findbugsannotations.FindBugsAnnotated defines equals and ↩
    uses Object.hashCode()
```

This class overrides `equals(Object)`, but does not override `hashCode()`, and inherits the implementation of `hashCode()` from `java.lang.Object` (which returns the identity hash code, an arbitrary value assigned to the object by the VM). Therefore, the class is very likely to violate the invariant that equal objects must have equal hashcodes.

If you don't think instances of this class will ever be inserted into a `HashMap`/`HashTable`, the recommended `hashCode` implementation to use is:

```
public int hashCode() {
  assert false : "hashCode not designed";
  return 42; // any arbitrary constant will do
  }
```

```
Rank: Troubling (14), confidence: High
Pattern: HE_EQUALS_USE_HASHCODE
Type: HE, Category: BAD_PRACTICE (Bad practice)
```

This error contains an explanation of the problem and hints about how to solve it. In this case the solution is basically to implement the `hashCode()` method as well.

For a complete list of all FindBugs violations that can be used as value in the `SuppressFBWarnings` annotation (doc).

13.5 JAXB

JAXB is a library used for conversion and mapping of XML files into Java objects and vice versa. Actually this library comes with the standard JRE and there is no need to download it or configure it in any way. It can be used directly by importing the classes in the package `javax.xml.bind.annotation` in your applications.

JAXB uses annotations to inform its processor (or the JVM) of the XML to code (and viceversa) conversion. For example there are annotations used to indicate XML nodes in the code, XML attributes, values, etc. We are going to see an example:

First of all, we declare a class indicating that it should be a node in the XML:

```
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
@XmlType( propOrder = { "brand", "model", "year", "km" } )
@XmlRootElement( name = "Car" )
class Car
...
```

The annotations used here are `@XmlType` and `@XmlRootElement`. They inform the JAXB processor that the class `Car` is going to be a node in the XML produced in the result of the conversion. The `@XmlType` indicates the order of the properties in the resultant XML. JAXB will perform the proper actions based on these annotations.

Apart of a setters and getters for the desired properties, nothing else is needed in this class in order to make it available for conversion. Now we need a consumer program that executes the conversion to XML:

```
Car car = new Car();
car.setBrand( "Mercedes" );
car.setModel( "SLK" );
car.setYear( 2011 );
car.setKm( 15000 );

Car carVW = new Car();
carVW.setBrand( "VW" );
carVW.setModel( "Touran" );
carVW.setYear( 2005 );
carVW.setKm( 150000 );

/* init jaxb marshaler */
JAXBContext jaxbContext = JAXBContext.newInstance( Car.class );
Marshaller jaxbMarshaller = jaxbContext.createMarshaller();

/* set this flag to true to format the output */
jaxbMarshaller.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, true );

/* marshaling of java objects in xml (output to standard output) */
jaxbMarshaller.marshal( car, System.out );
jaxbMarshaller.marshal( carVW, System.out );
```

The output of this program will be something like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Car>
    <brand>Mercedes</brand>
    <model>SLK</model>
    <year>2011</year>
    <km>15000</km>
</Car>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Car>
    <brand>VW</brand>
    <model>Touran</model>
    <year>2005</year>
```

```
    <km>150000</km>
</Car>
```

There is a list of annotations that can be used in JAXB for XML to Java conversion. More information can be found in https://jaxb.java.net/

Chapter 14

Summary

In this article we explained that annotations in Java are a very important feature available since the update 5, and we listed several uses cases where they can be used. Basically annotations are metadata that contain information about the marked code. They do not change or affect the code by any meaning and they can be used by 3rd applications called consumers in order to analyze the code using reflection.

We listed the built in annotations that are available per default in Java, some of them also called meta annotations like `@Target` or `@Retention`, or others like `@Override` or `@SuppressWarnings`, and the new features coming out in Java 8 related to annotations like the `@Repeteable` annotation, the `@FunctionalInterface` annotation and the annotations by type.

We also shown a couple of code examples where annotations were used in combination with reflection and we mentioned and described several real life libraries that are doing extensive use of annotations in Java like Junit, Spring or Hibernate.

Annotations are a very powerful mechanism in Java to analyze the Meta data of any kind of programs and can be applicable in different scopes like validation, dependency injection or unit testing.

Chapter 15

Download

You can download the full source code of this tutorial here: customAnnotations

Chapter 16

Resources

Here is a list of very useful resources related to Java annotations:

- Official Java annotations site
- Wikipedia article about annotations in Java
- Java Specification Request 250
- Annotations white paper from Oracle
- Annotations API