**Taken from:** [**http://tutorials.jenkov.com/java/annotations.html**](http://tutorials.jenkov.com/java/annotations.html)

**Java Annotation Purposes**

Java annotations are typically used for the following purposes:

* Compiler instructions
* Build-time instructions
* Runtime instructions

Java has 3 built-in annotations that you can use to give instructions to the Java compiler. These annotations are explained in more detail later in this text.

Java annotations can be used at build-time, when you build your software project. The build process includes generating source code, compiling the source, generating XML files (e.g. deployment descriptors), packaging the compiled code and files into a JAR file etc. Building the software is typically done by an automatic build tool like Apache Ant or Apache Maven. Build tools may scan your Java code for specific annotations and generate source code or other files based on these annotations.

Normally, Java annotations are not present in your Java code after compilation. It is possible, however, to define your own annotations that are available at runtime. These annotations can then be accessed via [Java Reflection](http://tutorials.jenkov.com/java-reflection/index.html), and used to give instructions to your program, or some third party API.

**Annotation Basics**

An annotation in its shortest form looks like this:

@Entity

The @ character signals to the compiler that this is an annotation. The name following the @ character is the name of the annotation. In the example above the annotation name is Entity.

**Annotation Elements**

Annotations can have elements for which you can set values. An element is like an attribute or parameter. Here is an example of an annotation with an element:

@Entity(tableName = "vehicles")

The annotation in this example contains a single element named tableName, with the value set to vehicles. Elements are enclosed inside the parentheses after the annotation name. Annotations without elements do not need the parentheses.

An annotation can contain multiple elements. Here is an example:

@Entity(tableName = "vehicles", primaryKey = "id")

In case an annotation contains just a single element, it is convention to name that element value, like this:

@InsertNew(value = "yes")

When an annotation just contains a single element named value, you can leave out the element name, and just provide the value. Here is an example:

@InsertNew("yes")

**Annotation Placement**

You can put Java annotations above classes, interfaces, methods, method parameters, fields and local variables. Here is an example annotation added above a class definition:

@Entity

public class Vehicle {

}

The annotation starts with the @ character, followed by the name of the annotation. In this case, the annotation name is Entity. The Entity annotation is an annotation I have made up. It doesn't have any meaning in Java.

Here is a bigger example with annotations above both the class, fields, methods, parameters and local variables:

**@Entity**

public class Vehicle {

**@Persistent**

protected String vehicleName = null;

**@Getter**

public String getVehicleName() {

return this.vehicleName;

}

public void setVehicleName(**@Optional** vehicleName) {

this.vehicleName = vehicleName;

}

public List addVehicleNameToList(List names) {

**@Optional**

List localNames = names;

if(localNames == null) {

localNames = new ArrayList();

}

localNames.add(getVehicleName());

return localNames;

}

}

The annotations are again just annotations I have made up. They have no specific meaning in Java.

**Java's Builtin Annotations**

Java comes with three annotations which are used to give the Java compiler instructions. These annotations are:

* @Deprecated
* @Override
* @SuppressWarnings

Each of these annotations are explained in the following sections.

**@Deprecated**

The @Deprecated annotation is used to mark a class, method or field as deprecated, meaning it should no longer be used. If your code uses deprecated classes, methods or fields, the compiler will give you a warning. Here is an example:

@Deprecated

public class MyComponent {

}

The use of the @Deprecated annotation above the class declaration marks the class as deprecated.

You can also use the @Deprecated annotation above method and field declarations, to mark the method or field as deprecated.

When you use the @Deprecated annotation, it is a good idea to also use the corresponding @deprecated JavaDoc symbol, and explain why the class, method or field is deprecated, and what the programmer should use instead.

**@Override**

The @Override annotation is used above methods that override methods in a superclass. If the method does not match a method in the superclass, the compiler will give you an error.

The @Override annotation is not necessary in order to override a method in a superclass. It is a good idea to use it still, though. In case someone changed the name of the overridden method in the superclass, your subclass method would no longer override it. Without the @Override annotation you would not find out. With the @Override annotation the compiler would tell you that the method in the subclass is not overriding any method in the superclass.

Here is an example use of the @Override annotation:

public class MySuperClass {

public void doTheThing() {

System.out.println("Do the thing");

}

}

public class MySubClass extends MySuperClass{

@Override

public void doTheThing() {

System.out.println("Do it differently");

}

}

In case the method doTheThing() in MySuperClass changes signature so that the same method in the supclass no longer overrides it, the compiler will generate an error.

**@SuppressWarnings**

The @SuppressWarnings annotation makes the compiler suppress warnings for a given method. For instance, if a method calls a deprecated method, or makes an insecure type cast, the compiler may generate a warning. You can suppress these warnings by annotating the method containing the code with the @SuppressWarnings annotation.

Here is an example:

@SuppressWarnings

public void methodWithWarning() {

}

**Creating Your Own Annotations**

It is possible to create your own annotations. Annotations are defined in their own file, just like a Java class or interface. Here is an example:

@interface MyAnnotation {

String value();

String name();

int age();

String[] newNames();

}

This example defines an annotation called MyAnnotation which has four elements.

Notice that each element is defined similarly to a method definition in an interface. It has a data type and a name. You can use all primitive data types as element data types. You can also use arrays as data type. You cannot use complex objects as data type.

To use the above annotation, you do like this:

@MyAnnotation(

value="123",

name="Jakob",

age=37,

newNames={"Jenkov", "Peterson"}

)

public class MyClass {

}

As you can see, I have to specify values for all elements of the MyAnnotation annotation.

**Element Default Values**

You can specify default values for an element. That way the element becomes optional and can be left out. Here is an example of how the annotation definition looks with a default value for an element:

@interface MyAnnotation {

String value() default "";

String name();

int age();

String[] newNames();

}

The value element can now be left out when using the annotation. If you leave it out, it will be considered as if you had used the default value for the value element. Here is an example:

@MyAnnotation(

name="Jakob",

age=37,

newNames={"Jenkov", "Peterson"}

)

public class MyClass {

}

Notice that the value element is no longer present.

**@Retention**

You can specify for your custom annotation if it should be available at runtime, for inspection via reflection. You do so by annotating your annotation definition with the @Retention annotation. Here is how that is done:

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnnotation {

String value() default "";

}

Notice the annotation added above the MyAnnotation definition:

@Retention(RetentionPolicy.RUNTIME)

This is what signals to the compiler and JVM that the annotation should be available via reflection at runtime. Accessing annotation at runtime is covered in my [Java Reflection and Annotations tutorial](http://tutorials.jenkov.com/java-reflection/annotations.html), which is part of my [Java Reflection Tutorial](http://tutorials.jenkov.com/java-reflection/index.html).

The RetentionPolicy class contains two more values you can use:

RetentionPolicy.CLASS means that the annotation is stored in the .class file, but not available at runtime. This is the default retention policy, if you do not specify any retention policy at all.

RetentionPolicy.SOURCE means that the annotation is only available in the source code, and not in the .class files and not a runtime. If you create your own annotations for use with build tools that scan the code, you can use this retention policy. That way the .class files are not poluted unnecessarily.

**@Target**

You can specify which Java elements your custom annotation can be used to annotate. You do so by annotating your annotation definition with the @Target annotation. Here is an example:

import java.lang.annotation.ElementType;

import java.lang.annotation.Target;

@Target({ElementType.METHOD})

public @interface MyAnnotation {

String value();

}

This example shows an annotation that can only be used to annotate methods.

The ElementType class contains the following possible targets:

* ElementType.ANNOTATION\_TYPE
* ElementType.CONSTRUCTOR
* ElementType.FIELD
* ElementType.LOCAL\_VARIABLE
* ElementType.METHOD
* ElementType.PACKAGE
* ElementType.PARAMETER
* ElementType.TYPE

Most of these are self explaining, but a few are not. Therefore is here an explanation of the few that are not obvious.

The ANNOTATION\_TYPE target means annotation definitions. Thus, the annotation can only be used to annotate other annotations. Like the @Target and @Retention annotations.

The TYPE target means any type. A type is either a class, interface, enum or annotation.

**@Inherited**

The @Inherited annotation signals that a custom annotation used in a class should be inherited by subclasses inheriting from that class. Here is an example:

java.lang.annotation.Inherited

@Inherited

public @interface MyAnnotation {

}

@MyAnnotation

public class MySuperClass { ... }

public class MySubClass extends MySuperClass { ... }

In this example the class MySubClass inherits the annotation @MyAnnotation because MySubClass inherits from MySuperClass, and MySuperClass has a @MyAnnotation annotation.

**@Documented**

The @Documented annotation is used to signal to the JavaDoc tool that your custom annotation should be visible in the JavaDoc for classes using your custom annotation. Here is an example:

java.lang.annotation.Documented

@Documented

public @interface MyAnnotation {

}

@MyAnnotation

public class MySuperClass { ... }

When generating JavaDoc for the MySuperClass class, the @MyAnnotation is now included in the JavaDoc.

You will not use the @Documented annotation often, but now you know it exists, if you should need it.