**Java Annotations – Annotations in Java**

[**https://www.journaldev.com/721/java-annotations**](https://www.journaldev.com/721/java-annotations)

Annotations in java provide information about the code. Java annotations have no direct effect on the code they annotate. In java annotations tutorial, we will look into following;

1. see built-in Java annotation example
2. how to write custom annotation
3. annotations usage and how to parse annotations using [reflection](https://www.journaldev.com/1789/java-reflection-example-tutorial)

## Java Annotations

Annotations are introduced in Java 1.5 and now it’s heavily used in Java EE frameworks like Hibernate, [Jersey](https://www.journaldev.com/498/jersey-java-tutorial), [Spring](https://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial).

Java Annotation is metadata about the program embedded in the program itself. It can be parsed by the annotation parsing tool or by compiler. We can also specify annotation availability to either compile time only or till runtime also.

Before java annotations, program metadata was available through java comments or by javadoc but annotation offers more than that. Annotations metadata can be available at runtime too and annotation parsers can use it to determine the process flow.

For example, in [Jersey webservice](https://www.journaldev.com/498/jersey-java-tutorial) we add PATH annotation with URI string to a method and at runtime jersey parses it to determine the method to invoke for given URI pattern.

### Java Custom Annotation

Creating custom annotation in java is similar to writing an interface, except that it interface keyword is prefixed with **@** symbol. We can declare methods in annotation.

Let’s see java custom annotation example and then we will discuss it’s features and important points.

package com.journaldev.annotations;

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Inherited;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

**@Documented**

**@Target(ElementType.METHOD)**

**@Inherited**

**@Retention(RetentionPolicy.RUNTIME)**

public @interface MethodInfo{

String author() default "Pankaj";

String date();

int revision() default 1;

String comments();

}

Some important points about java annotations are:

1. Annotation methods can’t have parameters.
2. Annotation methods return types are limited to primitives, String, Enums, Annotation or array of these.
3. Java Annotation methods can have default values.
4. Annotations can have meta annotations attached to them. Meta annotations are used to provide information about the annotation.

### Meta annotations in java

J2SE 5.0 provides four annotations in the java.lang.annotation package that are used only when writing annotations:

**@Documented – Whether to put the annotation in Javadocs**

**@Retention – When the annotation is needed**

**@Target? – Places the annotation can go**

**@Inherited – Whether subclasses get the annotation.**

**@Documented** – A simple market annotations which tells whether to add Annotation in java doc or not.

**@Retention** – Defines for how long the annotation should be kept.

**RetentionPolicy.SOURCE** – Discard during the compile. These annotations don’t make any sense after the compile has completed, so they aren’t written to the bytecode. Examples @Override, @SuppressWarnings  
**RetentionPolicy.CLASS** – Discard during class load. Useful when doing bytecode-level post-processing. Somewhat surprisingly, this is the default.  
**RetentionPolicy.RUNTIME** – Do not discard. The annotation should be available for reflection at runtime. This is what we generally use for our custom annotations.

**@Target** – Where annotation can be placed. If you don’t specify this, annotation can be placed anywhere. Following are the valid values. One important point here is, it’s inclusive only which means if you want annotation on 7 attributes and just want to exclude only one attribute, you need to include all 7 while defining target.

**ElementType.TYPE (class, interface, enum)**

**ElementType.FIELD (instance variable)**

**ElementType.METHOD**

**ElementType.PARAMETER**

**ElementType.CONSTRUCTOR**

**ElementType.LOCAL\_VARIABLE**

**ElementType.ANNOTATION\_TYPE (on another annotation)**

**ElementType.PACKAGE (remember package-info.java)**

**@Inherited** – Controls whether annotation should affect subclass.

There are four types of meta annotations:

* 1. **@Documented** – indicates that elements using this annotation should be documented by javadoc and similar tools. This type should be used to annotate the declarations of types whose annotations affect the use of annotated elements by their clients. If a type declaration is annotated with Documented, its annotations become part of the public API of the annotated elements.
  2. **@Target** – indicates the kinds of program element to which an annotation type is applicable. Some possible values are TYPE, METHOD, CONSTRUCTOR, FIELD etc. If Target meta-annotation is not present, then annotation can be used on any program element.
  3. **@Inherited** – indicates that an annotation type is automatically inherited. If user queries the annotation type on a class declaration, and the class declaration has no annotation for this type, then the class’s superclass will automatically be queried for the annotation type. This process will be repeated until an annotation for this type is found, or the top of the class hierarchy (Object) is reached.
  4. **@Retention** – indicates how long annotations with the annotated type are to be retained. It takes RetentionPolicy argument whose Possible values are SOURCE, CLASS and RUNTIME

### Built-in annotations in Java

Java Provides three built-in annotations.

1. @Override – When we want to override a method of Superclass, we should use this annotation to inform compiler that we are overriding a method. So when superclass method is removed or changed, compiler will show error message. Learn why we should always use [java override annotation](https://www.journaldev.com/817/overriding-methods-in-java-always-use-override-annotation) while overriding a method.
2. @Deprecated – when we want the compiler to know that a method is deprecated, we should use this annotation. Java recommends that in javadoc, we should provide information for why this method is deprecated and what is the alternative to use.
3. @SuppressWarnings – This is just to tell compiler to ignore specific warnings they produce, for example using raw types in [java generics](https://www.journaldev.com/1663/java-generics-example-method-class-interface" \o "Java Generics Tutorial – Example Class, Interface, Methods, Wildcards and much more). It’s retention policy is SOURCE and it gets discarded by compiler.

### Java Annotations Example

Let’s see a java example showing use of built-in annotations in java as well as use of custom annotation created by us in above example.

package com.journaldev.annotations;

import java.io.FileNotFoundException;

import java.util.ArrayList;

import java.util.List;

public class AnnotationExample {

public static void main(String[] args) {

}

@Override

@MethodInfo(author = "Pankaj", comments = "Main method", date = "Nov 17 2012", revision = 1)

public String toString() {

return "Overriden toString method";

}

@Deprecated

@MethodInfo(comments = "deprecated method", date = "Nov 17 2012")

public static void oldMethod() {

System.out.println("old method, don't use it.");

}

@SuppressWarnings({ "unchecked", "deprecation" })

@MethodInfo(author = "Pankaj", comments = "Main method", date = "Nov 17 2012", revision = 10)

public static void genericsTest() throws FileNotFoundException {

List l = new ArrayList();

l.add("abc");

oldMethod();

}

}

I believe above java annotation example is self explanatory and showing use of annotations in different cases.

### Java Annotations Parsing

We will use Reflection to parse java annotations from a class. Please note that Annotation Retention Policy should be RUNTIME otherwise it’s information will not be available at runtime and we wont be able to fetch any data from it.

package com.journaldev.annotations;

import java.lang.annotation.Annotation;

import java.lang.reflect.Method;

public class AnnotationParsing {

public static void main(String[] args) {

try {

for (Method method : AnnotationParsing.class.getClassLoader()

.loadClass(("com.journaldev.annotations.AnnotationExample")).getMethods()) {

// checks if MethodInfo annotation is present for the method

if (method.isAnnotationPresent(com.journaldev.annotations.MethodInfo.class)) {

try {

// iterates all the annotations available in the method

for (Annotation anno : method.getDeclaredAnnotations()) {

System.out.println("Annotation in Method '" + method + "' : " + anno);

}

MethodInfo methodAnno = method.getAnnotation(MethodInfo.class);

if (methodAnno.revision() == 1) {

System.out.println("Method with revision no 1 = " + method);

}

} catch (Throwable ex) {

ex.printStackTrace();

}

}

}

} catch (SecurityException | ClassNotFoundException e) {

e.printStackTrace();

}

}}

Output of the above program is:

Annotation in Method 'public java.lang.String com.journaldev.annotations.AnnotationExample.toString()' : @com.journaldev.annotations.MethodInfo(author=Pankaj, revision=1, comments=Main method, date=Nov 17 2012)

Method with revision no 1 = public java.lang.String com.journaldev.annotations.AnnotationExample.toString()

Annotation in Method 'public static void com.journaldev.annotations.AnnotationExample.oldMethod()' : @java.lang.Deprecated()

Annotation in Method 'public static void com.journaldev.annotations.AnnotationExample.oldMethod()' : @com.journaldev.annotations.MethodInfo(author=Pankaj, revision=1, comments=deprecated method, date=Nov 17 2012)

Method with revision no 1 = public static void com.journaldev.annotations.AnnotationExample.oldMethod()

Annotation in Method 'public static void com.journaldev.annotations.AnnotationExample.genericsTest() throws java.io.FileNotFoundException' : @com.journaldev.annotations.MethodInfo(author=Pankaj, revision=10, comments=Main method, date=Nov 17 2012)

Reflection API is very powerful and used widely in Java, J2EE frameworks like Spring, [Hibernate](https://www.journaldev.com/3793/hibernate-tutorial), JUnit, check out **[Reflection in Java](https://www.journaldev.com/1789/java-reflection-example-tutorial" \o "Java Reflection)**.

That’s all for the java annotations example tutorial, I hope you learned something from it.

### Quiz: Java Annotations

Answers are highlighted in Green Colour

Was there any annotation or similar functionality existing prior to 1.5 of Java ?

**Explanation:**Prior to java 1.5 , there were similar There were couple of annotation already available prior to javv 5those are @override,@author etc

* Yes
* No

Which of the following symbol is used to represent an annotation ?

**Explanation:**the symbol @ is used to represent the followed word as an annotation

* #
* @
* $
* &

 Which of the following way is a valid annotation definition ?

**Explanation:**annotation declaration needs @interface as type it can't be private, but can be public or default scope

* public @annotation MyAnnotation{ }
* private @interface MyAnnotation{ }
* public @interface MyAnnotation{ }
* public @MyAnnotation{ }

Which of the following retention policy and the type that defines @override ?

**Explanation:**Since the retention of the override is only in the source files, After the file compiles to class the annotation doesn't retain in the .class file ,And override can only be applied to methods in java code.

* SOURCE,METHOD
* CLASS,METHOD
* CLASS,TYPE
* TYPE,CLASS

Which of the following are the valid retention policy type available in Java ?

**Explanation:**SOURCE retains the annotation in the source code usually helpful for document generation. CLASS, in the class file,and RUNTIME when the runtime environment would take advantage of the annotation.

* SOURCE
* CLASS
* RUNTIME
* CODE
* TOOLS

Which of the following is a benefit from using @override annotation ?

**Explanation:**override annotation is a method type and source retention which helps to identify if a method is being overriden is actually being overriden ( meaning that it follows the correct rule of overriding)

* Override annotation says that the method marked must be overriden by subclass
* It helps in runtime to dynamic binding the method to appropriate object
* Compiler can warn you if the method desingated with override is not a correct override

Can you apply more than one attonation in a target  ?

**Explanation:**It is possible to apply more than one annotation to a target.

* Yes
* No

Annotations are a form of an interface where the keyword interface is preceded by @, and whose body contains annotation type element declarations that look very similar to methods:

1. primitives, String, Enums, Annotation or array of these.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public @interface SimpleAnnotation {      String value();        int[] types();  } |

After the annotation is defined, yon can start using it in through your code:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @SimpleAnnotation(value = "an element", types = 1)  public class Element {      @SimpleAnnotation(value = "an attribute", types = { 1, 2 })      public Element nextElement;  } |

Note that, when providing multiple values for array elements, you must enclose them in brackets.

Optionally, default values can be provided as long as they are constant expressions to the compiler:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public @interface SimpleAnnotation {      String value() default "This is an element";        int[] types() default { 1, 2, 3 };  } |

Now, you can use the annotation without those elements:

|  |  |
| --- | --- |
| 1  2  3  4 | @SimpleAnnotation  public class Element {      // ...  } |

Or only some of them:

|  |  |
| --- | --- |
| 1  2 | @SimpleAnnotation(value = "an attribute")  public Element nextElement; |

### ****Q4. What object types can be returned from an annotation method declaration?****

The return type must be a primitive, String, Class, Enum, or an array of one of the previous types. Otherwise, the compiler will throw an error.

Here’s an example code that successfully follows this principle:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | enum Complexity {      LOW, HIGH  }    public @interface ComplexAnnotation {      Class<? extends Object> value();        int[] types();        Complexity complexity();  } |

The next example will fail to compile since Object is not a valid return type:

|  |  |
| --- | --- |
| 1  2  3 | public @interface FailingAnnotation {      Object complexity();  } |

### ****Q5. Which program elements can be annotated?****

Annotations can be applied in several places throughout the source code. They can be applied to declarations of classes, constructors, and fields:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @SimpleAnnotation  public class Apply {      @SimpleAnnotation      private String aField;        @SimpleAnnotation      public Apply() {          // ...      }  } |

Methods and their parameters:

|  |  |
| --- | --- |
| 1  2  3  4 | @SimpleAnnotation  public void aMethod(@SimpleAnnotation String param) {      // ...  } |

Local variables, including a loop and resource variables:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @SimpleAnnotation  int i = 10;    for (@SimpleAnnotation int j = 0; j < i; j++) {      // ...  }    try (@SimpleAnnotation FileWriter writer = getWriter()) {      // ...  } catch (Exception ex) {      // ...  } |

Other annotation types:

|  |  |
| --- | --- |
| 1  2  3  4 | @SimpleAnnotation  public @interface ComplexAnnotation {      // ...  } |

And even packages, through the package-info.java file:

|  |  |
| --- | --- |
| 1  2 | @PackageAnnotation  package com.baeldung.interview.annotations; |

As of Java 8, they can also be applied to the use of types. For this to work, the annotation must specify an @Target annotation with a value of ElementType.USE:

|  |  |
| --- | --- |
| 1  2  3  4 | @Target(ElementType.TYPE\_USE)  public @interface SimpleAnnotation {      // ...  } |

Now, the annotation can be applied to class instance creation:

|  |  |
| --- | --- |
| 1 | new @SimpleAnnotation Apply(); |

Type casts:

|  |  |
| --- | --- |
| 1 | aString = (@SimpleAnnotation String) something; |

Implements clause:

|  |  |
| --- | --- |
| 1  2  3  4 | public class SimpleList<T>    implements @SimpleAnnotation List<@SimpleAnnotation T> {      // ...  } |

And throws clause:

|  |  |
| --- | --- |
| 1  2  3 | void aMethod() throws @SimpleAnnotation Exception {      // ...  } |

### ****Q6. Is there a way to limit the elements in which an annotation can be applied?****

Yes, the @Target annotation can be used for this purpose. If we try to use an annotation in a context where it is not applicable, the compiler will issue an error.

Here’s an example to limit the usage of the @SimpleAnnotation annotation to field declarations only:

|  |  |
| --- | --- |
| 1  2  3  4 | @Target(ElementType.FIELD)  public @interface SimpleAnnotation {      // ...  } |

We can pass multiple constants if we want to make it applicable in more contexts:

|  |  |
| --- | --- |
| 1 | @Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PACKAGE }) |

We can even make an annotation so it cannot be used to annotate anything. This may come in handy when the declared types are intended solely for use as a member type in complex annotations:

|  |  |
| --- | --- |
| 1  2  3  4 | @Target({})  public @interface NoTargetAnnotation {      // ...  } |

### ****Q7. What are meta-annotations?****

Are annotations that apply to other annotations.

All annotations that aren’t marked with @Target, or are marked with it but include ANNOTATION\_TYPE constant are also meta-annotations:

|  |  |
| --- | --- |
| 1  2  3  4 | @Target(ElementType.ANNOTATION\_TYPE)  public @interface SimpleAnnotation {      // ...  } |

### ****Q8. What are repeating annotations?****

These are annotations that can be applied more than once to the same element declaration.

For compatibility reasons, since this feature was introduced in Java 8, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. For the compiler to do this, there are two steps to declared them.

First, we need to declare a repeatable annotation:

|  |  |
| --- | --- |
| 1  2  3  4 | @Repeatable(Schedules.class)  public @interface Schedule {      String time() default "morning";  } |

Then, we define the containing annotation with a mandatory value element, and whose type must be an array of the repeatable annotation type:

|  |  |
| --- | --- |
| 1  2  3 | public @interface Schedules {      Schedule[] value();  } |

Now, we can use @Schedule multiple times:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Schedule  @Schedule(time = "afternoon")  @Schedule(time = "night")  void scheduledMethod() {      // ...  } |

### ****Q9. How can you retrieve annotations? How does this relate to its retention policy?****

You can use the Reflection API or an annotation processor to retrieve annotations.

The @Retention annotation and its RetentionPolicy parameter affect how you can retrieve them. There are three constants in RetentionPolicy enum:

* RetentionPolicy.SOURCE – makes the annotation to be discarded by the compiler but annotation processors can read them
* RetentionPolicy.CLASS – indicates that the annotation is added to the class file but not accessible through reflection
* RetentionPolicy.RUNTIME –Annotations are recorded in the class file by the compiler and retained by the JVM at runtime so that they can be read reflectively
* SRC

Here’s an example code to create an annotation that can be read at runtime:

|  |  |
| --- | --- |
| 1  2  3  4 | @Retention(RetentionPolicy.RUNTIME)  public @interface Description {      String value();  } |

Now, annotations can be retrieved through reflection:

|  |  |
| --- | --- |
| 1  2  3 | Description description    = AnnotatedClass.class.getAnnotation(Description.class);  System.out.println(description.value()); |

An annotation processor can work with RetentionPolicy.SOURCE, this is described in the article [Java Annotation Processing and Creating a Builder](http://www.baeldung.com/java-annotation-processing-builder).

RetentionPolicy.CLASS is usable when you’re writing a Java bytecode parser.

### ****Q10. Will the following code compile?****

|  |  |
| --- | --- |
| 1  2  3  4 | @Target({ ElementType.FIELD, ElementType.TYPE, ElementType.FIELD })  public @interface TestAnnotation {      int[] value() default {};  } |

No. It’s a compile-time error if the same enum constant appears more than once in an @Target annotation.

Removing the duplicate constant will make the code to compile successfully:

|  |  |
| --- | --- |
| 1 | @Target({ ElementType.FIELD, ElementType.TYPE}) |

### ****Q11. Is it possible to extend annotations?****

**No. Annotations always extend java.lang.annotation.Annotation,** as stated in the [Java Language Specification](http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.6).

If we try to use the extends clause in an annotation declaration, we’ll get a compilation error:

|  |  |
| --- | --- |
| 1  2  3 | public @interface AnAnnotation extends OtherAnnotation {      // Compilation error |