**Java Annotations with examples**

[Java Annotations](https://docs.oracle.com/javase/tutorial/java/annotations/index.html) allow us to add metadata information into our source code, although they are not a part of the program itself. Annotations were added to the java from JDK 5. Annotation has no direct effect on the operation of the code they annotate (i.e. it does not affect the execution of the program).

In this tutorial we are going to cover following topics: Usage of annotations, how to apply annotations, what predefined annotation types are available in the Java and how to create custom annotations.

**What’s the use of Annotations?**

**1) Instructions to the compiler**: There are three built-in annotations available in Java (@Deprecated, @Override & @SuppressWarnings) that can be used for giving certain instructions to the compiler. For example the @override annotation is used for instructing compiler that the annotated method is overriding the method. More about these built-in annotations with example is discussed in the next sections of this article.

**2) Compile-time instructors**: Annotations can provide compile-time instructions to the compiler that can be further used by sofware build tools for generating code, XML files etc.

**3) Runtime instructions**: We can define annotations to be available at runtime which we can access using [java reflection](https://docs.oracle.com/javase/tutorial/reflect/) and can be used to give instructions to the program at runtime. We will discuss this with the help of an example, later in this same post.

**Annotations basics**

An annotation always starts with the symbol @ followed by the annotation name. The symbol @ indicates to the compiler that this is an annotation.

For e.g. @Override  
Here @ symbol represents that this is an annotation and the Override is the name of this annotation.

**Where we can use annotations?**  
Annotations can be applied to the classes, interfaces, methods and fields. For example the below annotation is being applied to the method.

@Override

void myMethod() {

//Do something

}

What this annotation is exactly doing here is explained in the next section but to be brief it is instructing compiler that myMethod() is a overriding method which is overriding the method (myMethod()) of super class.

**Built-in Annotations in Java**

Java has three built-in annotations:

* @Override
* @Deprecated
* @SuppressWarnings

**1) @Override:**

While overriding a method in the child class, we should use this annotation to mark that method. This makes code readable and avoid maintenance issues, such as: while changing the method signature of parent class, you must change the signature in child classes (where this annotation is being used) otherwise compiler would throw compilation error. This is difficult to trace when you haven’t used this annotation.

Example:

public class MyParentClass {

public void justaMethod() {

System.out.println("Parent class method");

}

}

public class MyChildClass extends MyParentClass {

@Override

public void justaMethod() {

System.out.println("Child class method");

}

}

I believe the example is self explanatory. To read more about this annotation, refer this article: [@Override built-in annotation](https://beginnersbook.com/2014/07/override-annotation-in-java/).

**2) @Deprecated**

@Deprecated annotation indicates that the marked element (class, method or field) is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field that has already been marked with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example. Make a note of case difference with @Deprecated and @deprecated. @deprecated is used for documentation purpose.

Example:

/\*\*

\* @deprecated

\* reason for why it was deprecated

\*/

@Deprecated

public void anyMethodHere(){

// Do something

}

Now, whenever any program would use this method, the compiler would generate a warning. To read more about this annotation, refer this article: [Java – @Deprecated annotation](https://beginnersbook.com/2014/07/deprecated-annotation-in-java/).

**3) @SuppressWarnings**

This annotation instructs compiler to ignore specific warnings. For example in the below code, I am calling a deprecated method (lets assume that the method deprecatedMethod() is marked with @Deprecated annotation) so the compiler should generate a warning, however I am using @@SuppressWarnings annotation that would suppress that deprecation warning.

@SuppressWarnings("deprecation")

void myMethod() {

myObject.deprecatedMethod();

}

**Creating Custom Annotations**

* Annotations are created by using @interface, followed by annotation name as shown in the below example.
* An annotation can have elements as well. They look like methods. For example in the below code, we have four elements. We should not provide implementation for these elements.
* All annotations extends java.lang.annotation.Annotation interface. Annotations cannot include any extends clause.

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 MyCustomAnnotation{

int studentAge() default 18;

String studentName();

String stuAddress();

String stuStream() default "CSE";

}

Note: All the elements that have default values set while creating annotations can be skipped while using annotation. For example if I’m applying the above annotation to a class then I would do it like this:

@MyCustomAnnotation(

studentName="Chaitanya",

stuAddress="Agra, India"

)

public class MyClass {

...

}

As you can see, we have not given any value to the studentAge and stuStream elements as it is optional to set the values of these elements (default values already been set in Annotation definition, but if you want you can assign new value while using annotation just the same way as we did for other elements). However we have to provide the values of other elements (the elements that do not have default values set) while using annotation.

**Note**: We can also have array elements in an annotation. This is how we can use them:  
Annotation definition:

@interface MyCustomAnnotation {

int count();

String[] books();

}

Usage:

@MyCustomAnnotation(

count=3,

books={"C++", "Java"}

)

public class MyClass {

}

Lets back to the topic again: In the custom annotation example we have used these four annotations: @Documented, @Target, @Inherited & @Retention. Lets discuss them in detail.

**@Documented**

@Documented annotation indicates that elements using this annotation should be documented by JavaDoc. For example:

java.lang.annotation.Documented

@Documented

public @interface MyCustomAnnotation {

//Annotation body

}

@MyCustomAnnotation

public class MyClass {

//Class body

}

While generating the javadoc for class MyClass, the annotation @MyCustomAnnotation would be included in that.

**@Target**

It specifies where we can use the annotation. For example: In the below code, we have defined the target type as METHOD which means the below annotation can only be used on methods.

import java.lang.annotation.ElementType;

import java.lang.annotation.Target;

@Target({ElementType.METHOD})

public @interface MyCustomAnnotation {

}

public class MyClass {

@MyCustomAnnotation

public void myMethod()

{

//Doing something

}

}

**Note**: 1) If you do not define any Target type that means annotation can be applied to any element.  
2) Apart from ElementType.METHOD, an annotation can have following possible Target values.  
ElementType.METHOD  
ElementType.PACKAGE  
ElementType.PARAMETER  
ElementType.TYPE  
ElementType.ANNOTATION\_TYPE  
ElementType.CONSTRUCTOR  
ElementType.LOCAL\_VARIABLE  
ElementType.FIELD

**@Inherited**

The @Inherited annotation signals that a custom annotation used in a class should be inherited by all of its sub classes. For example:

java.lang.annotation.Inherited

@Inherited

public @interface MyCustomAnnotation {

}

@MyCustomAnnotation

public class MyParentClass {

...

}

public class MyChildClass extends MyParentClass {

...

}

Here the class MyParentClass is using annotation @MyCustomAnnotation which is marked with @inherited annotation. It means the sub class MyChildClass inherits the @MyCustomAnnotation.

**@Retention**

It indicates how long annotations with the annotated type are to be retained.

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)

@interface MyCustomAnnotation {

}

Here we have used RetentionPolicy.RUNTIME. There are two other options as well. Lets see what do they mean:  
RetentionPolicy.RUNTIME: The annotation should be available at runtime, for inspection via java reflection.  
RetentionPolicy.CLASS: The annotation would be in the .class file but it would not be available at runtime.  
RetentionPolicy.SOURCE: The annotation would be available in the source code of the program, it would neither be in the .class file nor be available at the runtime.