JAVA ANNOTATION

What is Java Annotation?

Java Annotation is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

First, we will learn some built-in annotations then we will move on creating and using custom annotations.

Annotations start with `@`. Its syntax is:

```
@AnnotationName
```

There are several built-in annotations in Java. Some annotations are applied to Java code and some to other annotations.

Built-In Java Annotations used in Java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

Understanding Built-In Annotations

Let's understand the built-in annotations first.

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden. It is not mandatory to use @Override when overriding a method. However, if we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.

```
class Animal {
  public void displayInfo() {
    System.out.println("I am an animal.");
  }
}

class Dog extends Animal {
  @Override
  public void displayInfo() {
    System.out.println("I am a dog.");
  }
}

class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  }
}
```

Output

```
I am a dog.
```

In this example, the method displayInfo() is present in both the superclass Animal and subclass Dog. When this method is called, the method of the subclass is called instead of the method in the superclass.

The @Override annotation can be useful for two reasons

If programmer makes any mistake such as wrong method name, wrong parameter types while overriding, you would get a compile time error. As by using this annotation you instruct compiler that

you are overriding this method. If you don't use the annotation then the sub class method would behave as a new method (not the overriding method) in sub class.

It can improve the readability of the source code. So if you change the signature of overridden method then all the sub classes that overrides the particular method would throw a compilation error, which would eventually help you to change the signature in the sub classes. If you have lots of classes in your application then this annotation would really help you to identify the classes that require changes when you change the signature of a method.

@SuppressWarnings

The @SuppressWarnings annotation type allows Java programmers to disable compilation warnings for a certain part of a program (type, field, method, parameter, constructor, and local variable). Normally warnings are good. For example, if a class is annotated to suppress a particular warning, then a warning generated in a method inside that class will also be separated. However in some cases they would be inappropriate and annoying. This annotation allows us to say which kinds of warnings to ignore. While warning types can vary by compiler vendor, the two most common are deprecation and unchecked. As a matter of style, programmers should always use this annotation on the most deeply nested element where it is effective. If you want to suppress a warning in a particular method, you should annotate that method rather than its class.

```
import java.util.*;
class TestAnnotation2{
@SuppressWarnings("unchecked")
public static void main(String args[]){
ArrayList list=new ArrayList();
list.add("sonoo");
list.add("vimal");
list.add("ratan");

for(Object obj:list)
System.out.println(obj);

}}
```

Output

```
Now no warning at compile time.
```

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection. This annotation is dangerous because a warning is something potentially wrong with the code. So if we're getting any warning, the first approach should be resolving those errors. But if we're suppressing any warnings, we have to have some solid reason. The reason should be commented near to the annotation every time it is used.

Possible Values Inside @SuppressWarnings Annotation Element are as follows:

Values	Description
All	It will suppress all warnings.

Cast	Suppress the warning while casting from a generic type to a nonqualified type or the other way around.
Deprecation	Ignores when we're using a deprecated(no longer important) method or type.
divzero	Suppresses division by zero warning.
empty	Ignores warning of a statement with an empty body.
unchecked	It doesn't check if the data type is Object or primitive.
fallthrough	Ignores fall-through on switch statements usually (if "break" is missing).
hiding	It suppresses warnings relative to locals that hide variable
serial	It makes the compiler shut up about a missing serialVersionUID.
finally	Avoids warnings relative to finally block that doesn't return.
unused	To suppress warnings relative to unused code.

Note: The primary and most important benefit of using @SuppressWarnings Annotation is that if we stuck because of some known warning, then this will ignore the warning and move ahead. E.g. – deprecated and unchecked warnings.

@Deprecated

Deprecated means, generally, that something is acknowledged but discouraged. In IT, deprecation means that although something is available or allowed, it is not recommended or that, in the case where something must be used, to say it is deprecated means that its failings are recognized.

@Deprecated annotation indicates that the marked element is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation. A program element annotated @Deprecated is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists. Compilers warn when a deprecated program element is used or overridden in non-deprecated code. To use it, you simply precede the class, method, or member declaration with "@Deprecated." Using the @Deprecated annotation to deprecate a class, method, or field ensures that all compilers will issue warnings when code uses that program element. A deprecated class or method is like that. It is no longer important. It is so unimportant, in fact, that you should no longer use it, since it has been superseded and may cease to exist in the future.

How to Deprecate?

Via Deprecated interface

Via Deprecated class

Via Deprecating a method

Via Deprecating a member variable

Via Deprecating a constructor

```
class A{
void m(){System.out.println("hello m");}

@Deprecated
void n(){System.out.println("hello n");}
}

class TestAnnotation3{
public static void main(String args[]){

A a=new A();
a.n();
}}
```

At Compile Time

```
Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.
```

At Run Time

```
hello n
```

Java Custom Annotations

Java Custom annotations or Java User-defined annotations are easy to create and use.
The *@interface* element is used to declare an annotation. For example:

@interface MyAnnotation{}

Here, MyAnnotation is the custom annotation name.

Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

Method should not have any throws clauses

Method should return one of the following: primitive data types, String, Class, enum or array of these data types.

Method should not have any parameter.

We should attach @ just before interface keyword to define annotation.

It may assign a default value to the method.

Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

- ## Marker Annotation

What is a marker annotation in Java?

A Java marker annotation is a special kind of annotation that contains no members. Its purpose is to mark a Java item. To determine if a marker annotation is present, use the method isAnnotationPresent(), which is defined by the AnnotatedElement interface. Here is an example that uses a marker annotation. As marker interfaces do not contain any members, just declaring the annotation in your code is sufficient for it to influence the output in whatever terms you want.

```
//Code to demonstrate the use of Marker Annotation
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention (RetentionPolicy.RUNTIME)
@interface MyMarker { }
class Marker
{
@MyMarker
public static void myMethod()
{
Marker obj = new Marker ();
try
{
Method m = obj.getClass().getMethod ("myMethod");
if(m.isAnnotationPresent (MyMarker.class))
System.out.println ("MyMarker is present");
}
catch(NoSuchMethodException exc)
{
System.out.println ("Method not found !!");
}
}
public static void main (String args [])
{
myMethod ();
}
}
```

Output

MyMarker is Present

- ## Single Value/Member Annotation

What is single-member annotation in Java?

A single-member annotation **contains only one member**. It allows a shorthand form of specifying the value of the member. When only one member is present, you don't need to specify the name of the member. In order to use this shorthand, the name of the member must be value.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention (RetentionPolicy.RUNTIME)
@interface MySingle {
int value ()
 }
class Single
{
@MySingle (10)
public static void myMethod()
{
Single obj = new Single ();
try
{
Method m = obj.getClass().getMethod (“myMethod”);
MySingle anno = m.getAnnotation (MySingle.class);
System.out.println (anno.value ());
}
catch (NoSuchMethodException exc)
{
System.out.println (“Method not found !!”);
}
}
public static void main (String args [])
{
myMethod ();
}
}
```

Output

10

- Multi Value Annotation

An annotation that has more than one method, is called Multi-Value annotation.