UNIT-4

Annotations

Annotations

- Annotations are used to provide supplement information about a program.
- Annotations start with '@'.
- Annotations do not change action of a compiled program.
- Annotations help to associate metadata (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by compiler.

Important Annotations In Java :

Built-In Java Annotations used in Java code

- @Override
- @SuppressWarnings
- @Deprecated
- @FunctionalInterface

Built-In Java Annotations used in other annotations

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

Understanding Built-In Annotations

1. @Override:

This annotation makes sure that the sub class method is successfully overiding the parent class method.

While overriding a class, there is a chance of typing errors or spelling mistakes. In such cases, the method will not get overridden and you will get an error.

Override exception helps us to encounter such situtations by extracting a warning from the compiler.

Example :

```
class KeyPadPhone{
    void sendMessage(){
        System.out.println("Text message sent!");
    }
}

class AndroidPhone extends KeyPadPhone{
    @Override
    void sendMessage(){
        System.out.println("Message sent via WhatsApp!");
    }
}

public class CWH{
        public static void main(String args[]){
            AndroidPhone Samsung = new AndroidPhone();
            Samsung.sendMessage();
}
}
```

Output :

```
Message sent via WhatsApp!
```

2. @Deprecated :

This annotation is used to mark a deprecated method.

If developer uses the deprecated method then the compiler generated a warning.

There high chance of removal of deprecated methods in future versions therefore it is better to not use them.

Example :

```
class KeyPadPhone{
   @Deprecated
   void sendMessage(){
      System.out.println("Text message sent!");
   }
}

class AndroidPhone extends KeyPadPhone{
   @Override
   void sendMessage(){
      System.out.println("Message sent via WhatsApp!");
   }
}

public class CWH{
      public static void main(String args[]){
         AndroidPhone Samsung = new AndroidPhone();
         Samsung.sendMessage();
}
}
```

Output :

```
java: sendMessage() in KeyPadPhone has been deprecated.
```

3. @SupressWarnings :

This annotation helps us to supress some warnings that are being generated by compiler.

Example :

```
class KeyPadPhone{
   @Deprecated
   void sendMessage(){
      System.out.println("Text message sent!");
   }
}

class AndroidPhone extends KeyPadPhone{
   @Override
   void sendMessage(){
      System.out.println("Message sent via WhatsApp!");
   }
}

public class CWH{
      public static void main(String args[]){
         @SuppressWarnings("deprecation")
         AndroidPhone Samsung = new AndroidPhone();
         Samsung.sendMessage();
}
}
```

Output :

This time no warning is generated because we've suppressed the deprecation warning.

4. @FunctionalInterface :

- An interface which contains only one abstract method is known as functional interface.
- @FunctionInterface annotation helps us to make sure that a functional interface is not having more than one abstract method.

Example :

```
@FunctionalInterface
interface myFunctionalInterface {
   void method1();
   void methodd2();
}
public class CWH{
      public static void main(String args[]){
         System.out.println("Functional interface annotation");

}
}
```

Output :

```
java: Unexpected @FunctionalInterface annotation
myFunctionalInterface is not a functional interface
multiple non-overriding abstract methods found in interface myFunctionalInterface.
```

The above code generates error because the myFunctionalInterface is containing more than one abstract method.

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.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. 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

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```
@interface MyAnnotation{}
```

The @Override and @Deprecated are marker annotations.

2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

```
@interface MyAnnotation{
int value();
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation{
int value() default 0;
```

```
}
```

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

@MyAnnotation(value=10)

The value can be anything.

3) Multi-Value Annotation

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

```
@interface MyAnnotation{
int value1();
String value2();
String value3();
}
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation{
int value1() default 1;
String value2() default "";
String value3() default "xyz";
}
```

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

@MyAnnotation(value1=10, value2="JAVA",value3="Ghaziabad")

Built-in Annotations used in custom annotations in java

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

1. @Target

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation. **ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	Fields
METHOD	Methods
CONSTRUCTOR	Constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	Parameter

Example to specify annotation for a class

```
@Target(ElementType.TYPE)
@interface MyAnnotation{
int value1();
```

```
String value2();
}
```

Example to specify annotation for a class, methods or fields

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
@interface MyAnnotation{
int value1();
String value2();
}
```

2. @Retention

@Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM .

Example to specify the RetentionPolicy

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation{
int value1();
String value2();
}
```

Example of custom annotation: creating, applying and accessing annotation

Let's see the simple example of creating, applying and accessing annotation.

```
//Creating annotation
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation{
int value();
}

//Applying annotation
class Hello{
@MyAnnotation(value=10)
public void sayHello()
{
System.out.println("hello annotation");
}
}

//Accessing annotation
class TestCustomAnnotation1{
public static void main(String args[])throws Exception{

Hello h=new Hello();
Method m=h.getClass().getMethod("sayHello");

MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
System.out.println("value is: "+manno.value());
}}
```

Output:
value is: 10

How built-in annotaions are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

3. @Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

```
@Inherited
@interface ForEveryone { }//Now it will be available to subclass also


@interface ForEveryone { }
class Superclass{}


class Subclass extends Superclass{}
```

4. @Documented

- The @Documented Marks the annotation for inclusion in the documentation.
- Tells a tool that an annotation is to be documented.
- Annotations are not included in 'Javadoc' comments.
- The use of @Documented annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.