L55

Monday, February 7, 2022 2:00 PM

Annotations & Assertions

Annotations

Q1. The dictionary meaning of an annotation is the extra note or comment added to a text or a diagram. Java extends the same conc ept of providing extra information to source code.

In Java annotations are used to provide extra information about elements/sections of the source code.

It is important to note that annotations by themselves do not change the logic/operation of the code.

The annotations are processed usually by the external tools. They could be compile-time or run-time tools or simple Java documentation generation tools.

Below are some of the most commonly used annotations bundled with Java:

1. **@Deprecated** - this annotation can be used for any section of code like method, fields including top-level classes and interfaces. When compiler sees that a we are using some code which is marked as **deprecated**, it immediately raises a warning to intimate the programmers to refrain from using such code.
2. **@Override** - this annotation when used for methods informs the compiler that this method is trying to override a method provided in the su per class.
3. **@SuppressWarnings** - this annotation informs the compiler to suppress warning information. For example, @SuppressWarnings("deprecation") informs t he compiler to suppress the warnings generated by because of using a deprecated code.

Java also allows for writing custom annotations which are processed either by user's custom annotation processors or by annot ation processors developed by third software developers.

Below is the syntax for using annotations in code:

```
@Override          //simple annotation for a method without values public String toString() {
  ...
}
@MyCustomAnnotation1   //simple annotation for a field without values private int someField;
@MyCustomAnnotation2(name1 = "value1", name2 = "value2")          //annotation for a method with values public String someMethod() {
  ...
}
```

As you can use from the above code, annotations always start with the @ (at-symbol). There can be annotations with or without values. Annotations can be created for any fragment of code.

See and retype the below code which demonstrates the usage of @override annotation.

```java
package q11358;
public class Student {
    private String id;
    private String name;
    public Student(String id, String name) {
        this.id = id;
        this.name = name;
    }
    @Override
    public String toString() {
        return "Student[ id = " + id + ", name=" + name + " ]";
    }
    public static void main(String[] args) {
        Student st1 = new Student("1007", "Ganga");
        System.out.println("st1 : " + st1);
    }
}

```

Q2. The Format of an Annotation: In its simplest form, an annotation looks like the following

The at sign character (@) indicates to the compiler that what an annotation follows

```
@Entity
```

Let us consider an example

```
@Override
void mySuperMethod() { ... }
```

In the above example annotation's name is Override

The annotation can include elements, which can be named or unnamed, and there are values for those elements.

```
@Author(
  name = "Bernard Moret",
  date = "3/27/2005"
)
```

or

They can be written as

```
@SuppressWarnings(value = "unchecked")
```

If there is just one element named value, then the name can be omitted,

It is also possible to use multiple annotations on the same declaration.

```
@Author(name = "Jane Doe")
@EBook
class MyClass { ... }
```

If the annotations have the same type, then this is called a repeating annotation. This can be written as follows

```
@Author(name = "Ram")
@Author(name = "Lakshman")
class MyClass { ... }
```

The annotation types that are defined in the java.lang or java.lang.annotation packages of the Java SE API.

In the above examples, Override and SuppressWarnings are predefined Java annotations. It is also possible to define our own a nnotation types called custom annotations. The Author and Ebook annotations in the above example are custom annotation types.

Uses of annotations.

Annotations have a number of uses, among them

- **Information for the compiler**: These can be used by the compiler to detect errors or suppress warnings.
- **Compile-time and deployment-time processing:** Software tools can process annotation information to generate code, XML files, and so on.
- **Runtime processing**: Some annotations are available to be examined at runtime.

- [x] In annotations @ indicates to compiler that what an annotation does.
- [x] Override is a predefined java annotation.
- [ ] We cannot define custom annotations in java
- [x] We can use multiple annotations in one declaration

Q3. After java SE 8 release, annotations can also be applied to any type use. This means that annotations can be used anywhere yo u use a type. A few examples of where types are used are class instance creation expressions (new), casts, implements clauses , and throws clauses. This form of annotation is called a type annotation. Here are some examples

- Class instance creation expression

```
new @Interned MyObject();
```

- Type cast
  myStr = (@NonNull String) str;
- Thrown exception declaration
  void temperatureMonitor() throws
  @Critical TemperatureException { ... }

Type annotations were created to support improved analysis of Java programs this way of ensuring stronger type checking. For example, you want to ensure that a particular variable in your program is never assigned to null; you want to avoid triggering a NullPointerException. You can write a custom plug-in to check for this. You would then modify your code to annotate that particular variable, indicating that it is never assigned to null.

The variable declaration might look like this:
@NonNull String str;

When you compile the code, including the NonNull module at the command line, the compiler prints a warning if it detects a potential problem, allowing you to modify the code to avoid the error. After you correct the code to remove all warnings, this particular error will not occur when the program runs.

- [x] Type annotations are supported only the release of Java SE 8
- [x] Type annotations supports improved analysis of Java programs
- [ ] Type annotations can support before release of Java SE 8

Q4. Annotations that apply to other annotations are called meta-annotations. There are several meta-annotation types defined in **java.lang.annotation**.

@Retention: This specifies how the marked annotation is stored
- RetentionPolicy.SOURCE – The marked annotation is retained only in the source level and is ignored by the compiler
- RetentionPolicy.CLASS – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM)
- RetentionPolicy.RUNTIME – The marked annotation is retained by the JVM so it can be used by the runtime environment.

@Documented: This annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool.

@Target: This annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:
- ElementType.ANNOTATION_TYPE can be applied to an annotation type
- ElementType.CONSTRUCTOR can be applied to a constructor
- ElementType.FIELD can be applied to a field or property
- ElementType.LOCAL_VARIABLE can be applied to a local variable
- ElementType.METHOD can be applied to a method-level annotation
- ElementType.PACKAGE can be applied to a package declaration
- ElementType.PARAMETER can be applied to the parameters of a method
- ElementType.TYPE can be applied to any element of a class

@Inherited: annotation indicates that the annotation type can be inherited from the super class.

- [x] Annotations that are applied to another annotations are called meta annotations.
- [ ] Meta annotations are present in java.lang package
- [x] @Retention specifies that how a marked annotations are stored
- [x] ElementType.METHOD can be applied to a method-level annotation

Q5.
```
public interface House {
   @Deprecated
   void open();
   void openFrontDoor();
   void openBackDoor()
}
public class MyHouse implements House {
   public void open() {}
   public void openFrontDoor() {}
   public void openBackDoor() {}
}
```
If you compile this program, the compiler produces a warning because open was deprecated in the interface. What can you do to get rid of that warning?. Choose the correct answer from the following.

- [ ] Delete the deprecation warning by using @Delete("deprecation")
- [x] Suppress the warning by using @SuppressWarnings("deprecation")
- [ ] Use @Suppress("deprecation")
- [ ] Use @SuppressWarnings("deprecated")

Assertions

Q1. An assertion is used to verify if an assumption made by the programmer is valid or not. For example, there could be a piece of code which assumes that the int value stored in a variable is always positive and greater than zero.

For example in the below code :
```
assert (x > 0);
int total = x * x;
```
The line containing assert statement evaluates the expression x > 0. If the expression evaluates to false, meaning x is not greater than 0, then AssertionError is thrown during the execution of the code. If the value of x is found to be greater than 0, the execution continues normally.

Assertions are used during the development and testing to capture bugs early. Assertions are usually disabled in the code when deployed in production. Disabling of assertions is done by passing a flag -enableassertions or -ea.

For example:
```
java -ea MyMainClassName
```
Please note assertions that should not be used to check validity of parameters for public methods. Exceptions are the correct way for signalling such parameter validation errors. Assertions are also used in many test frameworks while writing test cases.

The syntax for using the assert keyword is :
```
assert <boolean_expression> ;
assert <boolean_expression> : <reason_text_expression> ;
```
In the above syntax, the **<reason_text_expression>** is optional, however when provided, it is converted to a String and that text is used as the error message for the **AssertionError** thrown.

Note the AssertionError extends Error and hence is an *unchecked exception* and should not be handled in the code.

See and retype the below code. You will notice that the code in the method getPositiveInt() provides negative int values for numbers less and or equal to 2. This is written to demonstrate how assertions are written.

```java
package q11359;
public class AssertionDemo {
    public static void main(String[] args) {
        int x = getPositiveInt(7);
        int y = getPositiveInt(2);
        assert (x > 0);
        assert (y > 0);
        int total = x + y;
        System.out.println("total = " + total);
    }
    public static int getPositiveInt(int num) {
        return num - 3;
    }
}

```