Top of Form

**Annotations in Java**

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. See below code for example.

/\* Java program to demonstrate that annotations are not barely comments (This program throws compiler

error because we have mentioned override, but not overridden, we haver overloaded display) \*/

class Base

{

public void display()

{

System.out.println("Base display()");

}

}

class Derived extends Base

{

@Override

public void display(int x)

{

System.out.println("Derived display(int )");

}

public static void main(String args[])

{

Derived obj = new Derived();

obj.display();

}

}

* Output :
* 10: error: method does not override or implement
* a method from a supertype
* If we remove parameter (int x) or we remove @override, the program compiles fine.

**Categories of Annotations**

There are 3 categories of Annotations:-

**1. Marker Annotations:**

The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** is an example of Marker Annotation.

Example: - @TestAnnotation()

**2. Single value Annotations:**

These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don’t need to specify the name of the member. However in order to use this shorthand, the name of the member must be **value.**

Example: - @TestAnnotation(“testing”);

**3. Full Annotations:**

These annotations consist of multiple data members/ name, value, pairs.

Example:- @TestAnnotation(owner=”Rahul”, value=”Class Geeks”)

**4. Type Annotations:**

These annotations can be applied to any place where a type is being used. For example, we can annotate the return type of a method. These are declared annotated with **@Target** annotation.

// Program to demonstrate a type annotation

import java.lang.annotation.ElementType;

import java.lang.annotation.Target;

// Using target annotation to annotate a type

@Target(ElementType.TYPE\_USE)

// Declaring a simple type annotation

@interface TypeAnnoDemo{}

public class Main {

public static void main(String[] args) {

// Annotating the type of a string

@TypeAnnoDemo String string = "I am annotated with a type annotation";

System.out.println(string);

abc();

}

// Annotating return type of a function

static @TypeAnnoDemo int abc(){

System.out.println("This function's return type is annotated");

return 0;

}

}

**Output:**

I am annotated with a type annotation

This function's return type is annotated

**5. Repeating Annotations:**

These are the annotations that can be applied to a single item more than once. For an annotation to be repeatable it must be annotated with the **@Repeatable** annotation, which is defined in the **java.lang.annotation** package. Its value field specifies the **container type** for the repeatable annotation. **The container is specified as an annotation whose value field is an array of the repeatable annotation type.** Hence, to create a repeatable annotation, firstly the container annotation is created and then the annotation type is specified as an argument to the @Repeatable annotation.

// Program to demonstrate a repeatable annotation

import java.lang.annotation.Annotation;

import java.lang.annotation.Repeatable;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.reflect.Method;

// Make Words annotation repeatable

@Retention(RetentionPolicy.RUNTIME)

@Repeatable(MyRepeatedAnnos.class)

@interface Words{

String word() default "Hello";

int value() default 0;

}

// Create container annotation

@Retention(RetentionPolicy.RUNTIME)

@interface MyRepeatedAnnos{

Words[] value();

}

public class Main {

// Repeat Words on newMethod

@Words(word = "First", value = 1)

@Words(word = "Second", value = 2)

public static void newMethod(){

Main obj = new Main();

try{

Class<?> c = obj.getClass();

// Obtain the annotation for newMethod

Method m = c.getMethod("newMethod");

// Display the repeated annotation

Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);

System.out.println(anno);

}catch (NoSuchMethodException e){

System.out.println(e);

}

}

public static void main(String[] args) {

newMethod();

}

}

**Output:**

@MyRepeatedAnnos(value={@Words(value=1, word="First"), @Words(value=2, word="Second")})

**Predefined/ Standard Annotations**

Java defines seven built-in annotations.

* Four are imported from java.lang.annotation: **@Retention**, **@Documented**, **@Target**, and **@Inherited**.
* Three are included in java.lang: **@Deprecated, @Override**and **@SuppressWarnings**

**@Deprecated Annotation**

* It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.
* The Javadoc [@deprecated tag](http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/deprecation/deprecation.html#javadoc_tag)should be used when an element has been deprecated.
* @deprecated tag is for documentation and @Deprecated annotation is for runtime reflection.
* @deprecated tag has higher priority than @Deprecated annotation when both are together used.

public class DeprecatedTest

{

@Deprecated

public void Display()

{

System.out.println("Deprecatedtest display()");

}

public static void main(String args[])

{

DeprecatedTest d1 = new DeprecatedTest();

d1.Display();

}

}

**Output:**

Deprecatedtest display()

**@Override Annotation**

It is a marker annotation that can be used only on methods. A method annotated with **@Override**must override a method from a superclass. If it doesn’t, a compile-time error will result (see [this](https://ide.geeksforgeeks.org/ElmP5S) for example). It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

Example:-

class Base

{

public void Display()

{

System.out.println("Base display()");

}

public static void main(String args[])

{

Base t1 = new Derived();

t1.Display();

}

}

class Derived extends Base

{

@Override

public void Display()

{

System.out.println("Derived display()");

}

}

**Output:**

Deprecatedtest display()

**@Documented Annotations**

It is a marker interface that tells a tool that an annotation is to be documented. Annotations are not included by Javadoc comments. Use of @Documented annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.

**@Target**

It is designed to be used only as an annotation to another annotation. **@Target**takes one argument, which must be constant from the **ElementType**enumeration. This argument specifies the type of declarations to which the annotation can be applied. The constants are shown below along with the type of declaration to which they correspond.

**@Inherited**

@Inherited is a marker annotation that can be used only on annotation declaration. It affects only annotations that will be used on class declarations. **@Inherited**causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited,**then that annotation will be returned.

**User-defined/ Custom Annotations**

User-defined annotations can be used to annotate program elements, i.e. variables, constructors, methods, etc. These annotations can be applied just before declaration of an element (constructor, method, classes, etc).

Syntax of Declaration:-

[Access Specifier] @interface<AnnotationName>

{

DataType <Method Name>() [default value];

}

* **AnnotationName**is an identifier.
* Parameter should not be associated with method declarations and **throws**clause should not be used with method declaration.
* Parameters will not have a null value but can have a default value.
* **default value**is optional.
* Return type of method should be either primitive, enum, string, class name or array of primitive, enum, string or class name type.

package source;

// A Java program to demonstrate user defined annotations

import java.lang.annotation.Documented;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

// user-defined annotation

@Documented

@Retention(RetentionPolicy.RUNTIME)

@ interface TestAnnotation

{

String Developer() default "Rahul";

String Expirydate();

} // will be retained at runtime

// Driver class that uses @TestAnnotation

public class Test

{

@TestAnnotation(Developer="Rahul", Expirydate="01-10-2020")

void fun1()

{

System.out.println("Test method 1");

}

@TestAnnotation(Developer="Anil", Expirydate="01-10-2021")

void fun2()

{

System.out.println("Test method 2");

}

public static void main(String args[])

{

System.out.println("Hello");

}

}

Output :

Hello

Bottom of Form

Top of Form

#### JUnit

JUnit is a unit testing framework for Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit, that originated with JUnit.

Testing is the process of checking the functionality of an application to ensure it runs as per requirements. Unit testing comes into picture at the developers’ level; it is the testing of single entity (class or method). Unit testing plays a critical role in helping a software company deliver quality products to its customers.

Unit testing can be done in two ways − manual testing and automated testing.

#### What is JUnit ?

JUnit is a unit testing framework for Java programming language. It plays a crucial role test-driven development, and is a family of unit testing frameworks collectively known as xUnit.

JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented. This approach is like "test a little, code a little, test a little, code a little." It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

#### Features of JUnit

* JUnit is an open source framework, which is used for writing and running tests.
* Provides annotations to identify test methods.
* Provides assertions for testing expected results.
* Provides test runners for running tests.
* JUnit tests allow you to write codes faster, which increases quality.
* JUnit is elegantly simple. It is less complex and takes less time.
* JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
* JUnit tests can be organized into test suites containing test cases and even other test suites.
* JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

#### What is a Unit Test Case ?

A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected. To achieve the desired results quickly, a test framework is required. JUnit is a perfect unit test framework for Java programming language.

A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.

There must be at least two unit test cases for each requirement − one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

#### Annotations for Junit testing

The Junit 4.x framework is annotation based, so let's see the annotations that can be used while writing the test cases.

**@Test** annotation specifies that method is the test method.

**@Test(timeout=1000)** annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

**@BeforeClass** annotation specifies that method will be invoked only once, before starting all the tests.

**@Before** annotation specifies that method will be invoked before each test.

**@After** annotation specifies that method will be invoked after each test.

**@AfterClass** annotation specifies that method will be invoked only once, after finishing all the tests.

#### Assert class

The org.junit.Assert class provides methods to assert the program logic.

#### ****Methods of Assert class****

The common methods of Assert class are as follows:

1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object obj)**: checks that object is null.
5. **void assertNotNull(Object obj)**: checks that object is not null.

#### Required jar files

You need to load **junit4.jar** and **hamcrest-core.jar** files.

[download the junit jar files](https://www.javatpoint.com/src/junit/junit4jars.zip)

#### Write the program logic

Let's write the logic to find the maximum number for an array.

1. **package** com.javatpoint.logic;
2. **public** **class** Calculation {
4. **public** **static** **int** findMax(**int** arr[]){
5. **int** max=0;
6. **for**(**int** i=1;i<arr.length;i++){
7. **if**(max<arr[i])
8. max=arr[i];
9. }
10. **return** max;
11. }
12. }

#### Write the test case

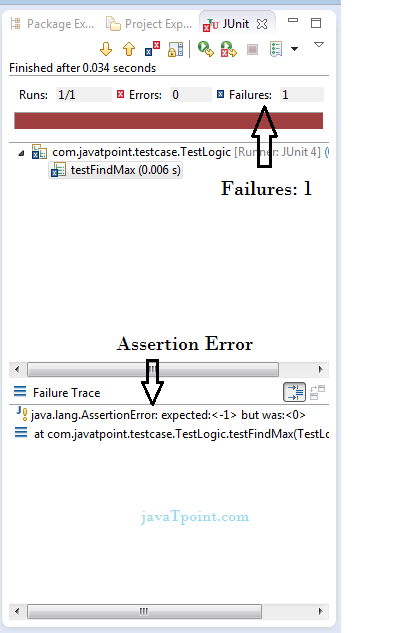
Here, we are using JUnit 4, so there is no need to inherit TestCase class. The main testing code is written in the testFindMax() method. But we can also perform some task before and after each test, as you can see in the given program.

1. **package** com.javatpoint.testcase;
3. **import** **static** org.junit.Assert.\*;
4. **import** com.javatpoint.logic.\*;
5. **import** org.junit.Test;
7. **public** **class** TestLogic {
9. @Test
10. **public** **void** testFindMax(){
11. assertEquals(4,Calculation.findMax(**new** **int**[]{1,3,4,2}));
12. assertEquals(-1,Calculation.findMax(**new** **int**[]{-12,-1,-3,-4,-2}));
13. }
14. }

To run this example, **right click on TestLogic class -> Run As -> 1Junit Test**.

**Output:**Assertion Error

Let's see the output displayed in eclipse IDE.



As you can see, when we pass the negative values, it throws AssertionError because second time findMax() method returns 0 instead of -1. It means our program logic is incorrect.

#### Correct program logic

As you can see, program logic to find the maximum number for the given array is not correct because it doesn't return -1 in case of negative values. The correct program logic is given below:

1. **package** com.javatpoint.logic;
2. **public** **class** Calculation {
4. **public** **static** **int** findMax(**int** arr[]){
5. **int** max=arr[0];//arr[0] instead of 0
6. **for**(**int** i=1;i<arr.length;i++){
7. **if**(max<arr[i])
8. max=arr[i];
9. }
10. **return** max;
11. }
12. }

If you run the junit program again, you will see the following output.

#### example of junit framework with output

Another example of Junit framework

#### Write the program code

1. **package** com.javatpoint.logic;
2. **public** **class** Calculation {
3. //method that returns maximum number
4. **public** **static** **int** findMax(**int** arr[]){
5. **int** max=0;
6. **for**(**int** i=1;i<arr.length;i++){
7. **if**(max<arr[i])
8. max=arr[i];
9. }
10. **return** max;
11. }
12. //method that returns cube of the given number
13. **public** **static** **int** cube(**int** n){
14. **return** n\*n\*n;
15. }
16. //method that returns reverse words
17. **public** **static** String reverseWord(String str){
19. StringBuilder result=**new** StringBuilder();
20. StringTokenizer tokenizer=**new** StringTokenizer(str," ");
22. **while**(tokenizer.hasMoreTokens()){
23. StringBuilder sb=**new** StringBuilder();
24. sb.append(tokenizer.nextToken());
25. sb.reverse();
27. result.append(sb);
28. result.append(" ");
29. }
30. **return** result.toString();
31. }
32. }

#### Write the test case

1. **package** com.javatpoint.testcase;
3. **import** **static** org.junit.Assert.assertEquals;
4. **import** org.junit.After;
5. **import** org.junit.AfterClass;
6. **import** org.junit.Before;
7. **import** org.junit.BeforeClass;
8. **import** org.junit.Test;
9. **import** com.javatpoint.logic.Calculation;
11. **public** **class** TestCase2 {
13. @BeforeClass
14. **public** **static** **void** setUpBeforeClass() **throws** Exception {
15. System.out.println("before class");
16. }
17. @Before
18. **public** **void** setUp() **throws** Exception {
19. System.out.println("before");
20. }
22. @Test
23. **public** **void** testFindMax(){
24. System.out.println("test case find max");
25. assertEquals(4,Calculation.findMax(**new** **int**[]{1,3,4,2}));
26. assertEquals(-2,Calculation.findMax(**new** **int**[]{-12,-3,-4,-2}));
27. }
28. @Test
29. **public** **void** testCube(){
30. System.out.println("test case cube");
31. assertEquals(27,Calculation.cube(3));
32. }
33. @Test
34. **public** **void** testReverseWord(){
35. System.out.println("test case reverse word");
36. assertEquals("ym eman si nahk",Calculation.reverseWord("my name is khan");
37. }
38. @After
39. **public** **void** tearDown() **throws** Exception {
40. System.out.println("after");
41. }
43. @AfterClass
44. **public** **static** **void** tearDownAfterClass() **throws** Exception {
45. System.out.println("after class");
46. }
48. }

**Output:**before class

before

test case find max

after

before

test case cube

after

before

test case reverse word

after

after class

Bottom of Form

Top of Form

**Reference Links:**

https://www.geeksforgeeks.org/annotations-in-java/

https://www.javatpoint.com/java-annotation

https://docs.oracle.com/javase/tutorial/java/annotations/basics.html

https://beginnersbook.com/2014/09/java-annotations/

https://www.programiz.com/java-programming/annotation-types

**Video Links:**

https://youtu.be/JV0atjBcUv4

https://youtu.be/0VPRkVWkM70

Bottom of Form