JUnit Tutorial | Testing Framework for Java

JUnit tutorial provides basic and advanced concepts of **unit testing in java** with examples. Our junit tutorial is designed for beginners and professionals.

It is an *open-source testing framework* for java programmers. The java programmer can create test cases and test his/her own code.

It is one of the unit testing framework. Current version is junit 4.

To perform unit testing, we need to create test cases. The **unit test case** is a code which ensures that the program logic works as expected.

Difference between structure and union in C

Keep Watching

The **org.junit** package contains many interfaces and classes for junit testing such as Assert, Test, Before, After etc.

Types of unit testing

There are two ways to perform unit testing: 1) manual testing 2) automated testing.

1) Manual Testing

If you execute the test cases manually without any tool support, it is known as manual testing. It is time consuming and less reliable.

2) Automated Testing

If you execute the test cases by tool support, it is known as automated testing. It is fast and more reliable.

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.

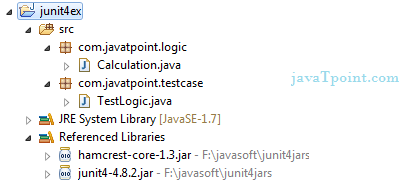
Required jar files

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

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

Simple JUnit example in eclipse IDE

Let's see the directory structure of this example.



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.

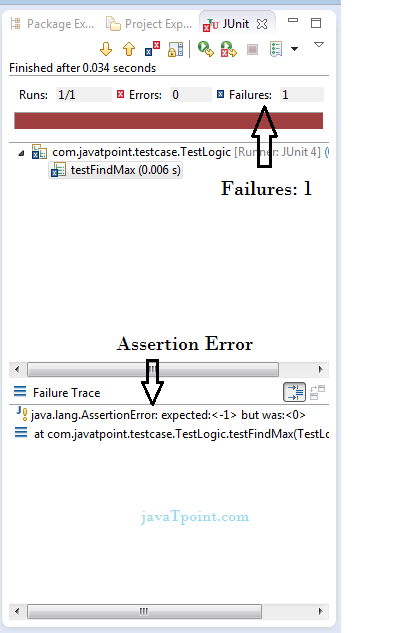
1. **package** com.javatpoint.testcase;
3. **import** **static** org.junit.Assert.\*;
4. **import** com.javatpoint.logic.\*;
5. **import** org.junit.Test;
7. **public** **class** TestLogic {
9. @Test
10. **public** **void** testFindMax(){
11. assertEquals(4,Calculation.findMax(**new** **int**[]{1,3,4,2}));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

[download this example](https://www.javatpoint.com/src/junit/junittesting.zip)

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.



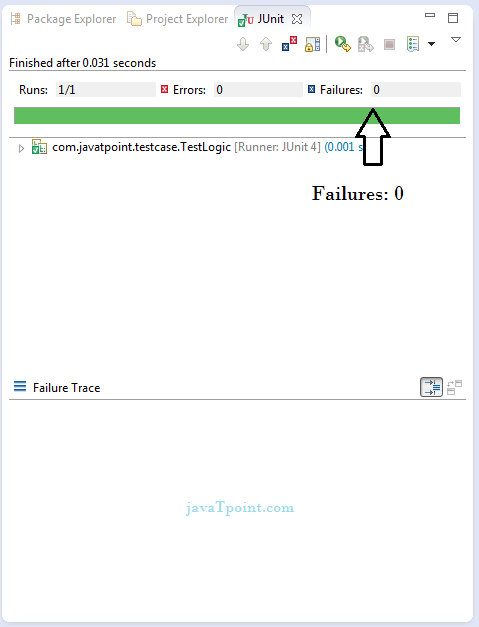
As you can see, when we pass the negative values, it throws AssertionError because second time findMax() method returns 0 instead of -1. It means our program logic is incorrect.

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.



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

Let's take the example of a JUnit test case to understand how testing is actually performed in Java.

**JunitTestCaseExample.java**

1. **package** JavaTpoint. JunitExamples;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **public** **class** JunitTestCaseExample {
5. **private** List<String> students = **new** ArrayList<String>();
7. **public** **void** remove(String name) {
8. students.remove(name);
9. }
11. **public** **void** add(String name) {
12. students.add(name);
13. }
15. **public** **void** removeAll(){
16. students.clear();
17. }
19. **public** **int** sizeOfStudent() {
20. **return** students.size();
21. }
23. }

**TestJunitTestCaseExample.java**

1. **package** JavaTpoint.JunitExamples;
2. **import** **static** org.junit.Assert.assertEquals;
3. **import** org.junit.Test;
4. **public** **class** TestJunitTestCaseExample {
6. JunitTestCaseExample obj = **new** JunitTestCaseExample();
8. @Test
9. **public** **void** testAdd() {
10. obj.add("Emma");
11. obj.add("Ronan");
12. obj.add("Antonio");
13. obj.add("Paul");
14. assertEquals("Adding 4 student to list", 4, obj.sizeOfStudent());
15. }
17. @Test
18. **public** **void** testSize() {
19. obj.add("Emma");
20. obj.add("Ronan");
21. obj.add("Antonio");
22. assertEquals("Checking size of List", 3, obj.sizeOfStudent());
23. }
25. @Test
26. **public** **void** testRemove() {
27. obj.add("Antonio");
28. obj.add("Paul");
29. obj.remove("Paul");
30. assertEquals("Removing 1 student from list", 1, obj.sizeOfStudent());
31. }
33. @Test
34. **public** **void** removeAll() {
35. obj.removeAll();
36. }
37. }

**TestRunner.java**

1. **package** JavaTpoint.JunitExamples;
3. **import** org.junit.runner.Result;
4. **import** org.junit.runner.JUnitCore;
5. **import** org.junit.runner.notification.Failure;
7. **public** **class** TestRunner {
8. **public** **static** **void** main(String[] args) {
9. Result result = JUnitCore.runClasses(TestJunitTestCaseExample.**class**);
11. **for** (Failure fail : result.getFailures()) {
12. System.out.println(fail.toString());
13. }
15. System.out.println(result.wasSuccessful());
16. }
17. }

**Explanation**

We have created three java classes, i.e., **JunitTestCaseExample.java, TestJunitTestCaseExample.java**, and **TestRunner**.java. In the **JunitTestCaseExample.java** class, we created the code which we want to test. In this class, we create a list of names and four methods to add an element, remove an element, get the list's size, and remove all elements from the list.

In the **TestJunitTestCaseExample.java**, we write the test cases for the JunitTestCaseEample.java class. We create an object of the **JunitTestCaseExample.java** class, and by using its object, we will test all its methods. We use the **assertEquals()** method to check the actual result with the expected output.

We create the TestRunner.java class to execute the test cases. It contains the main() method in which we run the TestJunitTestCaseExample.java class using the runClasses() method of the JUnitCore. In this method, we pass the class file of the TestJunitTestCaseExample.java. The result returned by the runClasses() method will store into the result variable of type Result.

Now, we will run the TestRunner.java class to execute the test cases. We will get the output true when the test cases are passed and false when the test cases are not passed.

**Output**

# JUnit test case example in JavaHow To Write Test Cases In Java

Test Cases are the conditions that are to be tested when the software is created. Before writing test cases in Java, we need to understand what test cases are. This section will cover a brief introduction of Test cases and then how we can write test cases in [Java](https://www.javatpoint.com/java-tutorial)

.

What are Test Cases

Test Cases are the set of conditions that are tested by a software tester for the developed application. The tester checks and evaluates if the built software is fulfilling all the requirements of the customer. If any bug or error is found, the tester informs the development team. The need to build test cases and perform testing is to verify that all the customer's needs are fulfilled, and no bugs are present in the software.

**For example:** For a login module, the tester would make certain following test cases:

1. Verify the login page consists of username and password text fields and a login button.
2. Verify on successful login, the user is redirected to the Home page.
3. Verify the successful login of the user by providing a valid username and password.
4. Verify the unsuccessful login of the user by providing a valid username and invalid password.
5. Verify the password of the user is kept in an encrypted form in the database and so on.

With these, there can be as per test cases a tester can think of, and when all the test cases are passed successfully, then only the built software is handed to the customer.

To know more about Test Cases, visit <https://www.javatpoint.com/test-case>

.

How to Create Test Cases in Java

Java is an object-oriented programming language, and building a Java project means creating lots of classes, each having certain methods. To build a good project/application, we need to test the project and check whether the application is fulfilling all the requirements. In Java, classes and methods play a vital role, and therefore we need to test the classes and methods, which together known as a **Unit**, and there comes the need to perform the **Unit testing** for them. So, for performing unit testing in Java, there are several testing frameworks. One such famous framework is **[JUnit](https://www.javatpoint.com/junit-tutorial)**

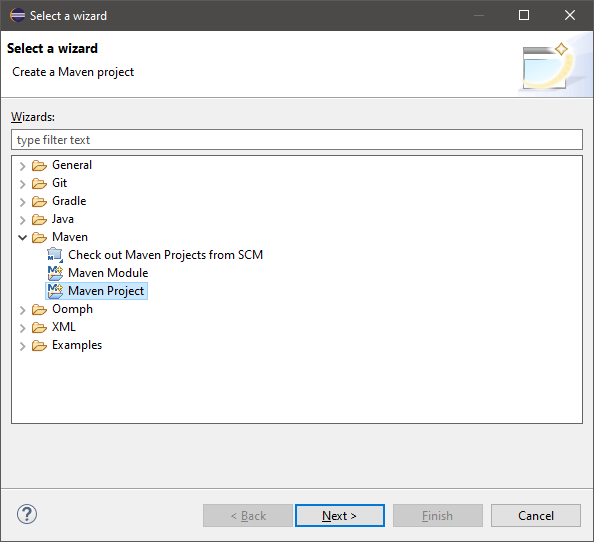
which is a framework for performing unit testing. In order to perform Junit testing, we need to create the test cases and test the project against each test case and determine the result.

**To understand how to create test cases, let's create a simple class and examine it.**

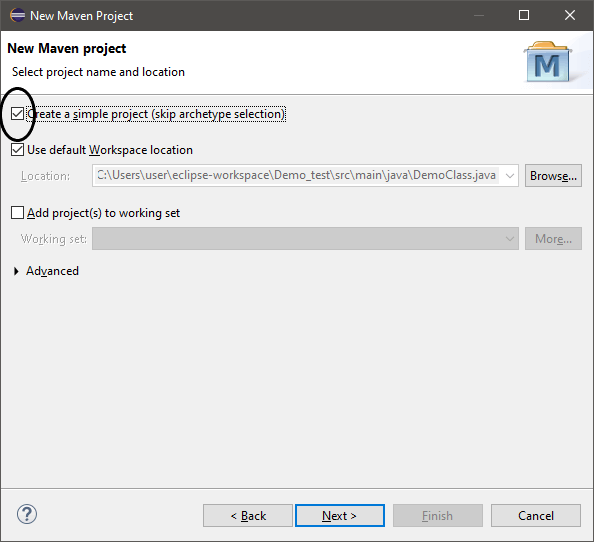
Let's create a class **DemoClass** in Eclipse where we have performed a simple addition method by following the below steps:

1) Open **Eclipse IDE** and create a new **[Maven](https://www.javatpoint.com/maven-tutorial)**

Project by clicking on **File> New > Other…** and then **Select a wizard** dialog box will open up. Under **Wizards**, you will see **the Maven** repository, click on the dropdown, click on **Maven Project**, and then click on **Next**. A snapshot is shown below:



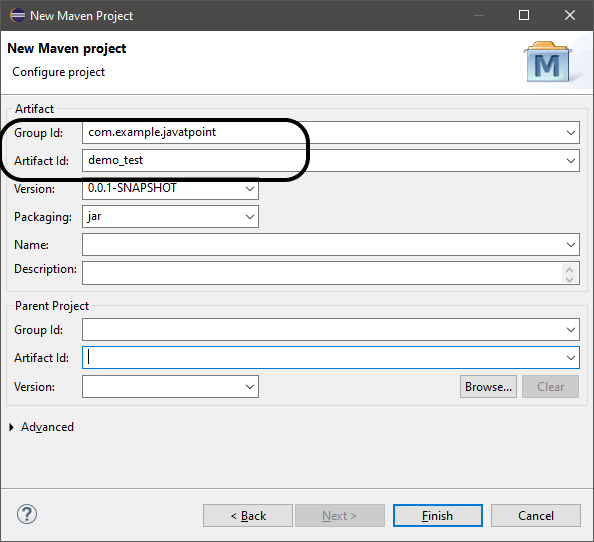
2) Select 'Create a simple project', or you can also make the archetype selection also. Here, we have done a simple project, so we have selected the 'Create a simple project' option and then click on **Next,** as you can see in the below snapshot:



3) Now, you can see **New Maven Project** dialog box will open where we have to provide the **Group Id** and **Artifact Id** by following:

**Group Id:** It is the unique name that helps in identifying one project group from other. It follows the Java naming rule convention, and so it must be provided as **com.example.javatpoint** or any other.

**Artifact Id:** It is the unique name that is given to the project we are going to create. So, for our project, we have provided the following Group Id and Artifact Id (i.e., the project name), as you can see in the below snapshot:



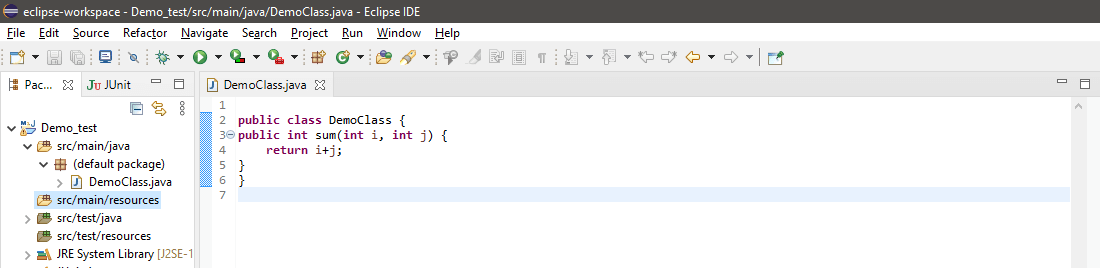
4) Click on **Finish**, and our Maven project will get created with certain files where the main File is the **xml** File that carries all the details of the created project.

5) Within the project name, you will see certain repositories, make a right-click on **src/main/java** repository, click on **New > Class**, and create a **[Java Class](https://www.javatpoint.com/object-and-class-in-java" \l "class)**

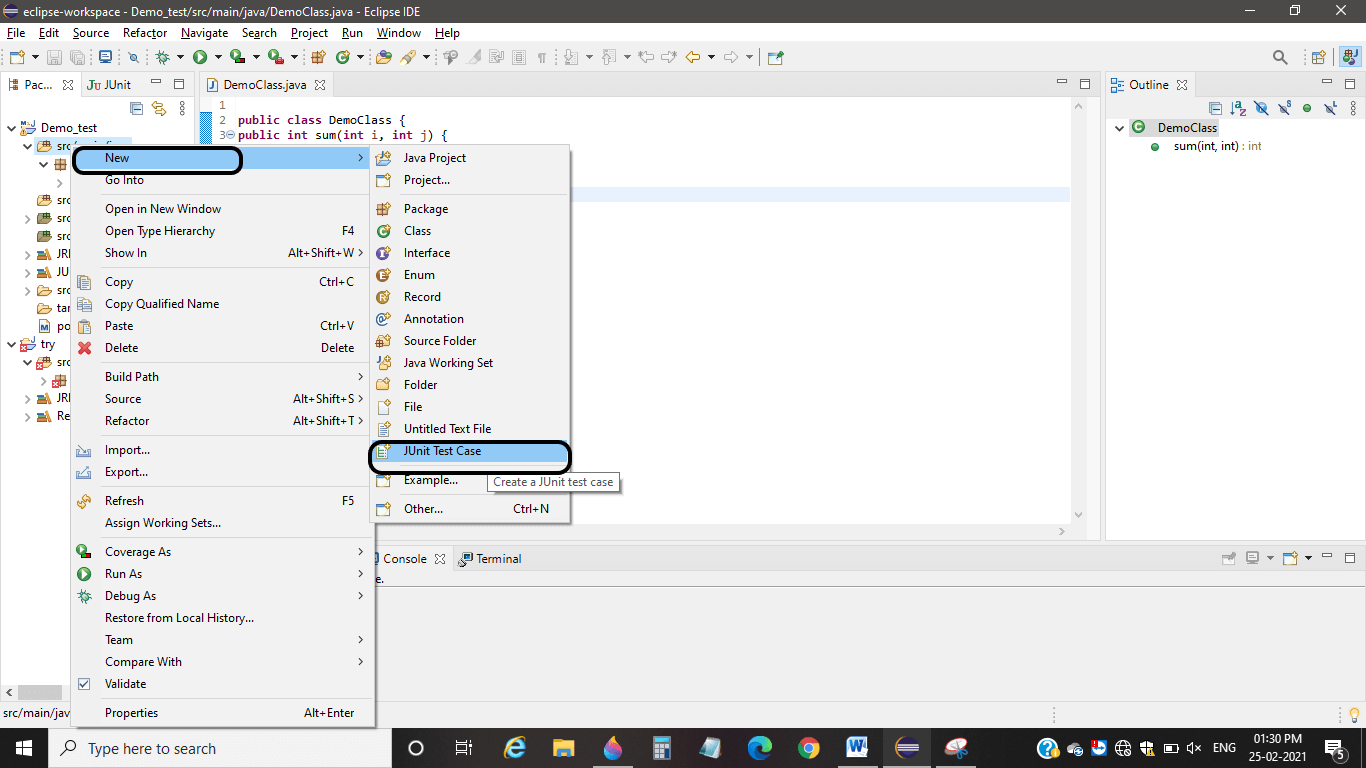
. Here, we have created **java**, where we have implemented a simple method of adding two numbers. The code is as follows:

1. **public** **class** DemoClass {
2. **public** **int** sum(**int** i, **int** j) {
3. **return** i+j;
4. }

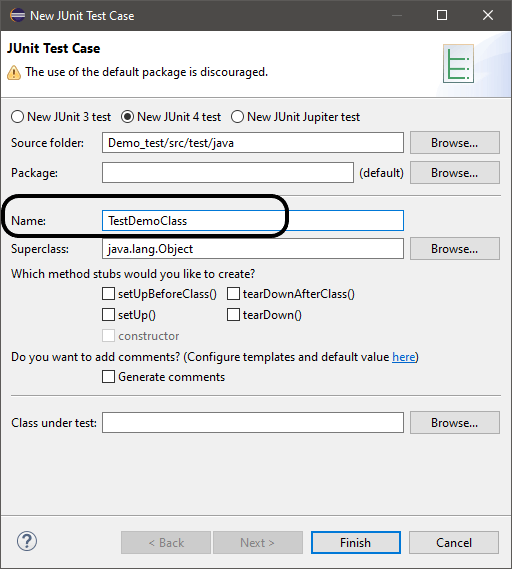
**The code snippet is shown below:**



6) Now, we have implemented a class, and it's time to test it, and for that, we need to use **Junit**. So, move to **the src/test/java** folder and make a right-click on it. Click on **New > Junit Test Case**.



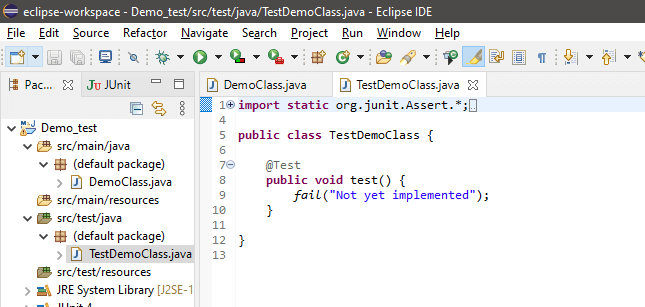
And create a JUnit test case following the naming rule conventions.



Then click on **Finish**. The JUnit test case will be created. Here, we have created the JUnit test case as **TestDemoClass.java**, and we got the following output screen covering the below code:

1. **import** **static** org. junit.Assert.\*;
3. **import** org.junit.Test;
5. **public** **class** TestDemoClass {
7. @Test
8. **public** **void** test() {
9. fail("Not yet implemented");
10. }
12. }

**The code snippet is shown below:**



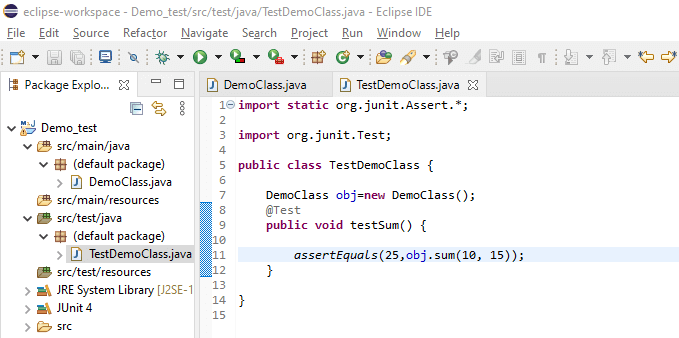
Here, we have used the New Junit 4 test. To test a class or method, we need to use the testing annotations, and here we have **@Test** annotation before the **test ()** method.

Note: We have taken a sample project that can be tested manually, but when doing a big project, it may consist of numerous classes with numerous methods. To check and test all those manually might not be flexible, and change in one class will definitely affect the other classes. Thus, there comes the role of automation testing, i.e., Junit testing.

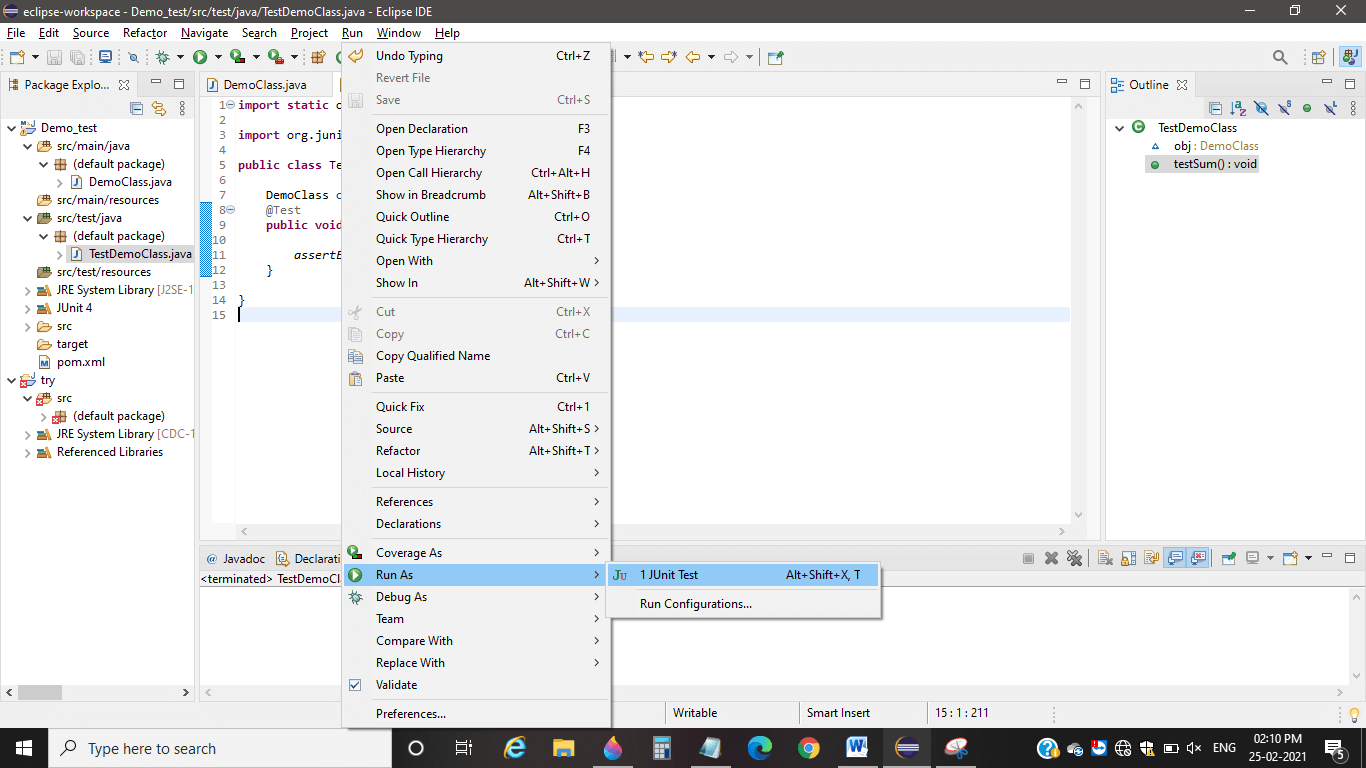
7) Create an object of **DemoClass** within the **TestDemoClass** and we need to test the **sum** method and for that we have created the following test case code:

1. **public** **class** TestDemoClass {
2. DemoClass obj=**new** DemoClass();
3. @Test
4. **public** **void** testSum() {
5. assertEquals(25,obj.sum(10, 15));
6. }
7. }

**The code snippet is shown below:**

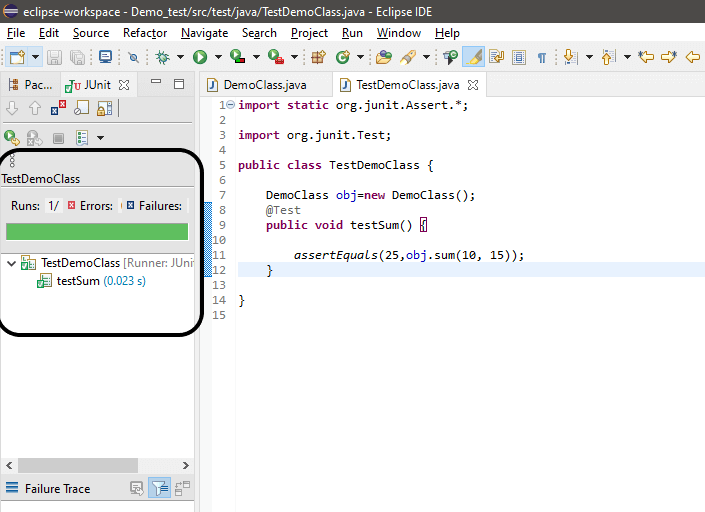


8) The use of the **assertEquals(expectedVal, calculatedVal)** method is used for determining the status of a test case, whether failed or passed. To run/execute a test case, right-click on the code, then **Run As > Junit Test**.



9) We have experimented with the following test cases:

**Test Case 1:** We have created a test case where we have tested whether on providing two values, we get the respective sum or not. If not, the test case will be failed, but for the above values, we got the following output:

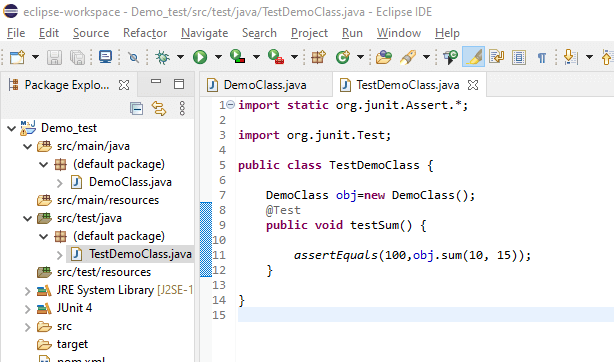


Hence, our test case passed successfully, which is indicated by the **green** symbol.

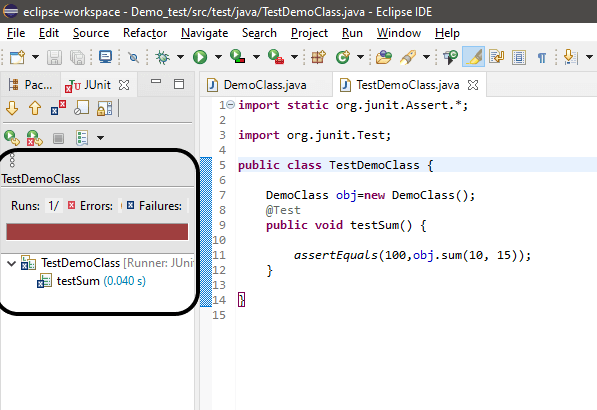
**Test Case 2:** If we pass the same code for the calculated parameters **(10, 15)** but the expected value as **100** and have the following values to test, we get our test case failed. The code is as:

1. **public** **class** TestDemoClass {
3. DemoClass obj=**new** DemoClass();
4. @Test
5. **public** **void** testSum() {
7. assertEquals(100,obj.sum(10, 15));
8. }
10. }

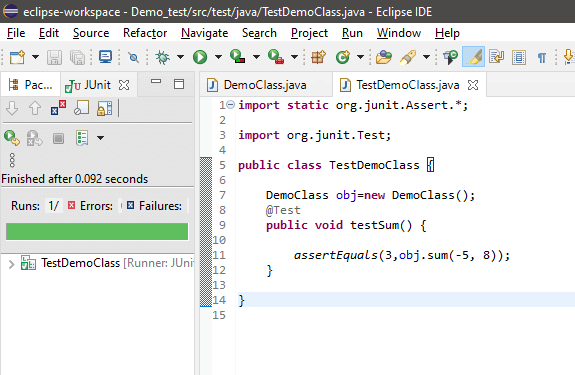
**The code snippet is shown below:**



So, we got our test case failed, which means our code is correct, and it is indicated by the **Red signal,** as you can see in the below snippet:

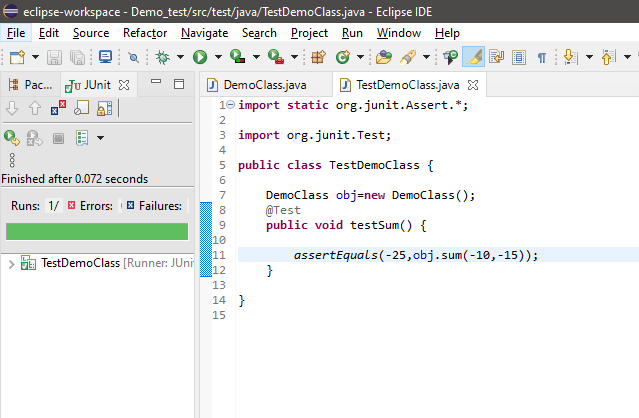


**Test Case 3:** If we pass a negative and a positive value for the calculated parameters as **(-5, 8),** and expected parameter value as **3**, we get our test case passed successfully, as you can see in the below snippet:

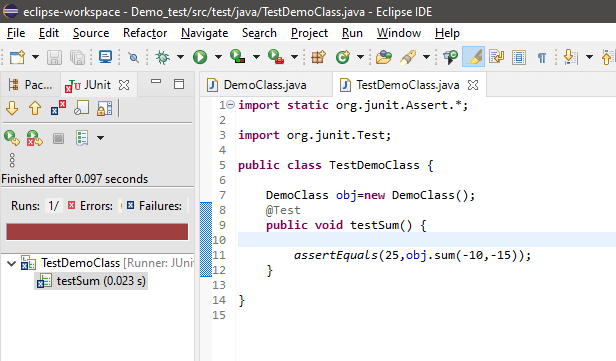


**However, if we pass (5, -8) as calculated value and expected value as 3, the test case would fail definitely.**

**Test Case 4:** If we pass both negative values for the calculated parameter as **(-10,-15)** and expected parameter value as **-25**, we get our test case passed, as you can see in the below snippet:



**However, if we pass the expected value as 25, the test case will definitely fail. Let's see in the below snippet:**



In this way, we can think of and create the test cases as per our thinking, judging, and testing ability. Apart from the example explained above, a software tester works with very large applications to test them and create certain test cases and test them. To record all the test cases, they make use of the excel sheet and examine whether their created test cases meet the customer requirements or are having some bugs. A software tester creates and writes the test cases by thinking self as a normal user, and so examine the application/software.