Unit Testing/Debugging Code

Topics Covered Under This Chapter:

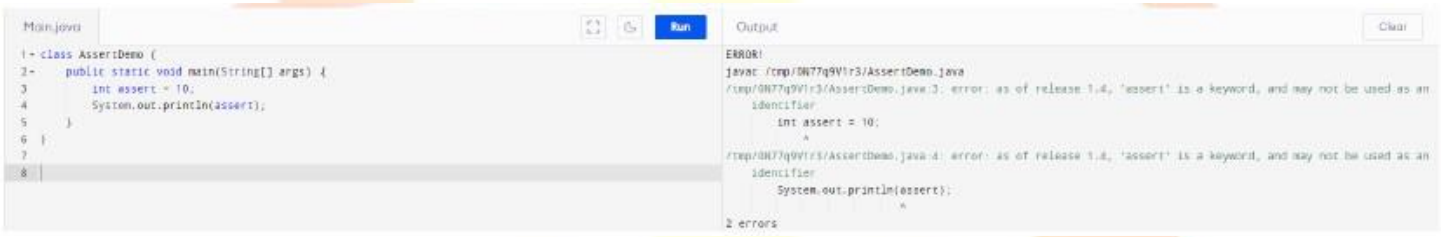
- Assertions
- Junit

1.. Assertions

- Unit Testing is nothing but testing the code and debugging the code done by the developer to ensure that there is no issue with the code running.
- The very common way of debugging any Java program is using of System.out.println() statements. But the problem with System.out.println() statements is after fixing the bug, it is compulsory to delete these unwanted System.out.println() statements use from the program. Otherwise, these unwanted statements will get executed during the Runtime for every request, which creates performance problem and disturbs server logs.
- To Overcome this problem, Assertions concept got introduced in 1.4 Version of Java.
- The main advantage of assertions when compared with System.out.println() statements is after fixing the bug, we are not required to remove assert statements. This is because they are not going to execute by default, at Runtime. Based on our requirement, we can enable or disable assertions. And by default, assertions is disabled.
- Hence, the main objective of assertions is to perform effective debugging.
- Usually, we can perform debugging in Dev and Test Environment but not in Prod environment. Hence, assertions concept is applicable only for Dev and Test but not for Prod Environment.

assert as a keyword and identifier

- Since, concept of assertion was introduced in Java version 1.4. So, before 1.4 version we can use assert as an identifier but after 1.4 version, we cannot use assert as an identifier as assert is a keyword from 1.4 version.



```
class AssertDemo {
    public static void main(String[] args) {
        int assert = 10;
        System.out.println(assert);
    }
}
```

```
ERROR!
javac /tmp/0N77q9V1r3/AssertDemo.java
/tmp/0N77q9V1r3/AssertDemo.java:3: error: as of release 1.4, 'assert' is a keyword, and may not be used as an
    identifier
        int assert = 10;
            ^
/tmp/0N77q9V1r3/AssertDemo.java:4: error: as of release 1.4, 'assert' is a keyword, and may not be used as an
    identifier
        System.out.println(assert);
                           ^
2 errors
```

- When an above program is compiled with respect to current version, it will give compile time error as mentioned in above screenshot. If we want to use assert as an identifier, then we can compile our java program by lowering the version by lower than 1.4.

  javac -source 1.3 <Java Program file>.java

Types of assert statements

There are two types of assert Statements:

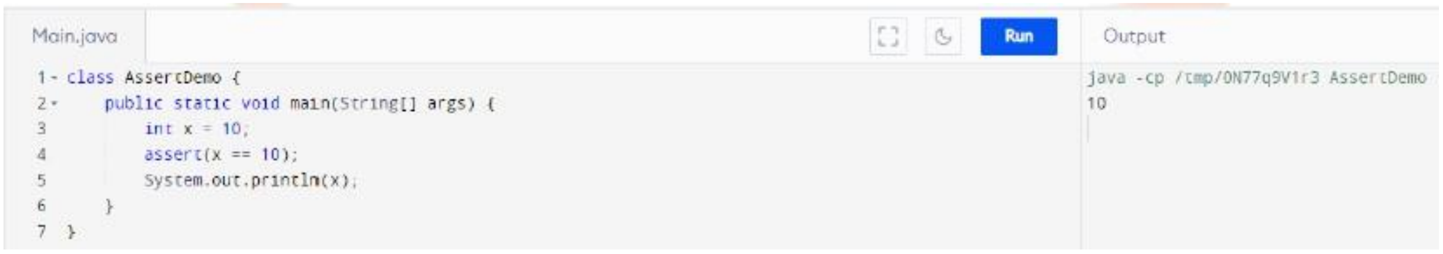
1. Simple Version

2. Augmented Version

Simple Version

The simple version of assert statement is: assert(boolean b);

If b is true, that means our assumption is true and program should continue its execution while if b is false, that means our assumption is not true, program will stop its execution and throws Realtime exception called **AssertionError**. Once we encounter AssertionError, we can analyze the code and debug it and fix it.

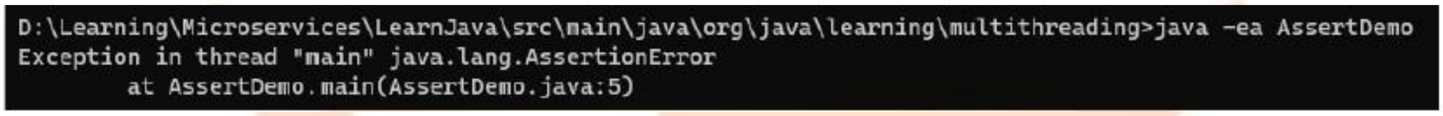


```java
class AssertDemo {
    public static void main(String[] args) {
        int x = 10;
        assert(x == 10);
        System.out.println(x);
    }
}
```

```
java -cp /tmp/0N77q9V1r3 AssertDemo
10
```

By default, assertion is disable. If we want to enable the assertion, while running java program, we have to pass -ea, then only we will see AssertionError else output will be same as above.

java -ea <Class Name>



```
D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>java -ea AssertDemo
Exception in thread "main" java.lang.AssertionError
        at AssertDemo.main(AssertDemo.java:5)
```

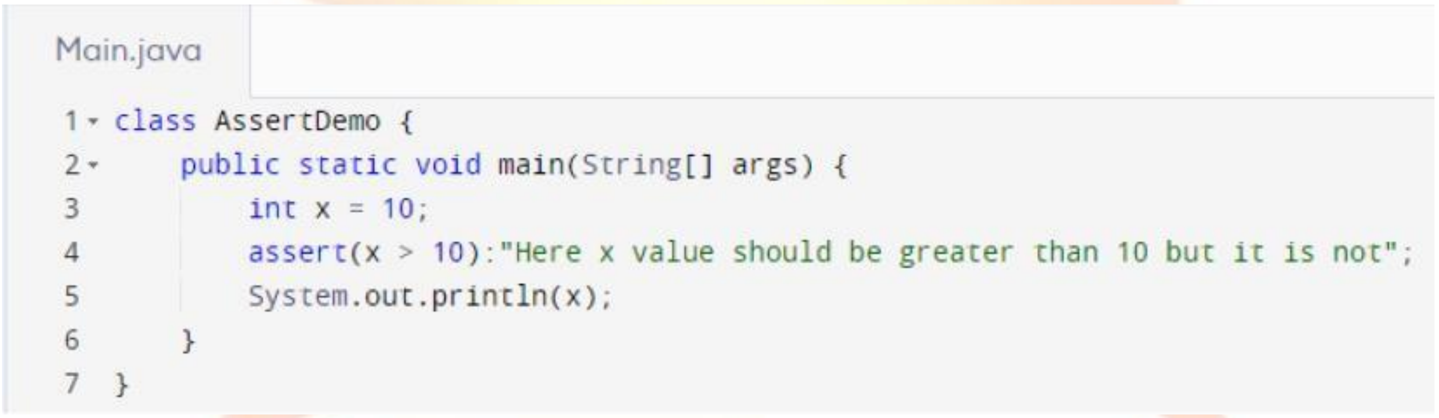
Augmented Version

We can augment some description with AssertionError by using Augmented version of assert statements.

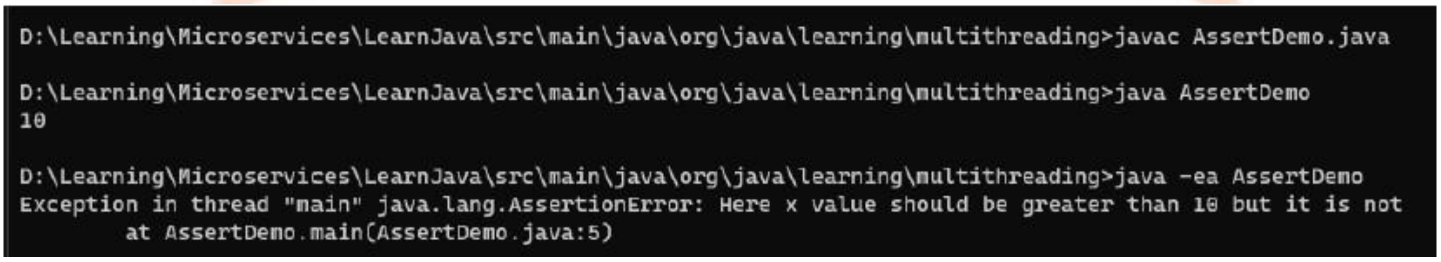
The augmented version of assert statement is: assert(boolean b) : e;

If b is true then compiler wont evaluate second argument after : part. After : part will be evaluated only if b is false.

Here e is a Description which we want to augment with AssertionError. Usually, we take e as String type but e can be of any type whatever we want to have. It can be String, Integer, Expression etc.

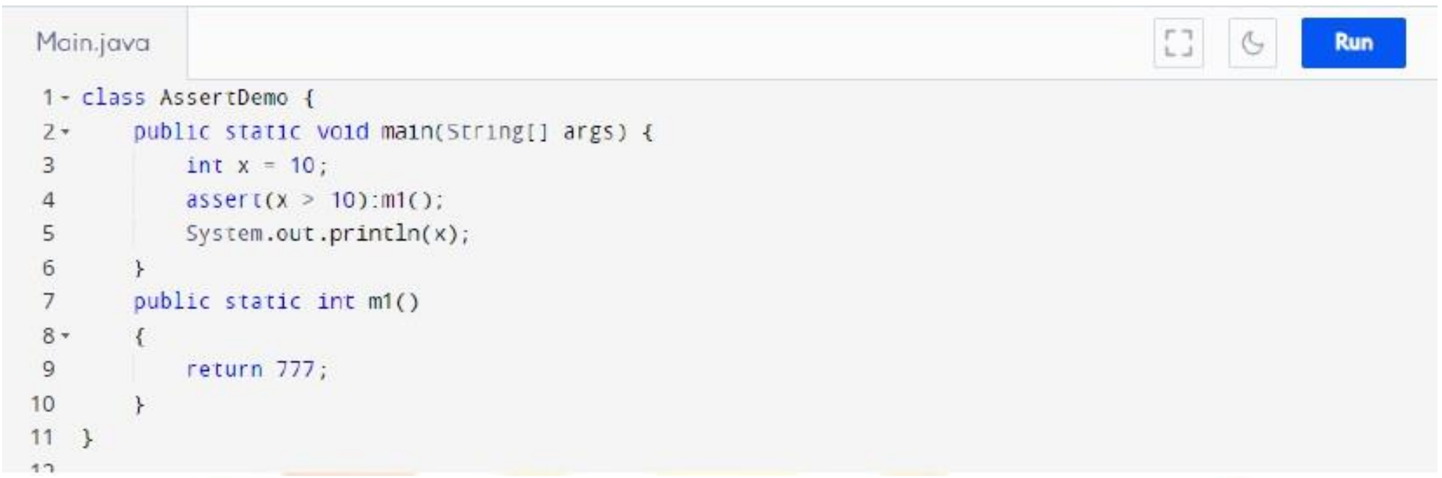


```java
class AssertDemo {
    public static void main(String[] args) {
        int x = 10;
        assert(x > 10):"Here x value should be greater than 10 but it is not";
        System.out.println(x);
    }
}
```

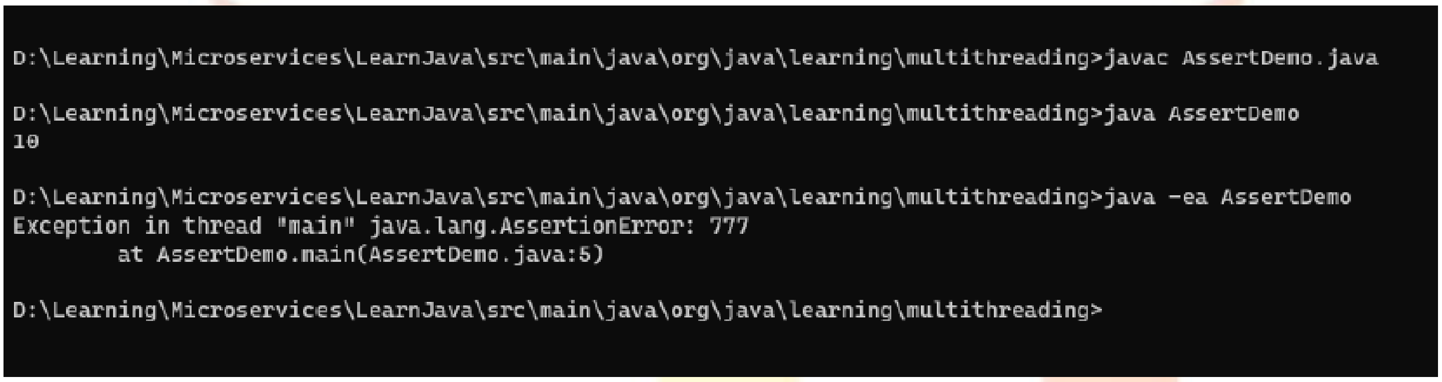


```
D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>javac AssertDemo.java

D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>java AssertDemo
10

D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>java -ea AssertDemo
Exception in thread "main" java.lang.AssertionError: Here x value should be greater than 10 but it is not
        at AssertDemo.main(AssertDemo.java:5)
```

We can have method call as well in the augmented version but it is compulsory that method should return something. It cannot be void.

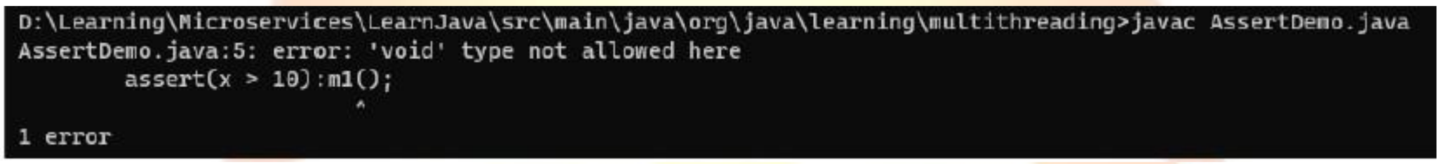


```java
class AssertDemo {
    public static void main(String[] args) {
        int x = 10;
        assert(x > 10):m1();
        System.out.println(x);
    }
    public static int m1()
    {
        return 777;
    }
}
```



```
D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>javac AssertDemo.java

D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>java AssertDemo
10

D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>java -ea AssertDemo
Exception in thread "main" java.lang.AssertionError: 777
        at AssertDemo.main(AssertDemo.java:5)

D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>
```

Now say method m1() doesn’t return anything.



```
D:\Learning\Microservices\LearnJava\src\main\java\org\java\learning\multithreading>javac AssertDemo.java
AssertDemo.java:5: error: 'void' type not allowed here
        assert(x > 10):m1();
                         ^
1 error
```

Various possible Runtime flags

We can use various possible runtime flags while running Java class.

- **-ea (-enableassertions)**
  Enables the assertion in all the non-System class. Non-System class means User-Defined class.

- **-da (-disableassertions)**
  Disables the assertion in all the non-System class. Non-System class means User-Defined class.

- **-esa (-enablesystemassertions)**
  Enables the assertion in all the System class or pre-defined class

- **-dsa (-disablesystemassertions)**
  Disables the assertion in all the System class or pre-defined class

Note: We can use any of these flags simultaneously while running java program and JVM will consider these flag from left to right.

Example:

java -ea -esa -ea -dsa -da -esa -ea -dsa Test

Here, at the end, assertions will be enabled only for non-System classes but will be disabled for System classes.

Case Study for Runtime Flags Usage

Suppose we have:

pack1 -> A.class, B.class, pack2

pack2 -> C.class, D.class

- **java -ea:pack1.B**
  Enable assertion only in B Class.

- **java -ea:pack1.B -ea:pack1.pack2.D**
  Enable assertion only in B and D Class.

- **java -ea:pack1…**
  Enable assertion in every Class of pack1.

- **java -ea:pack1… -da:pack1.B**
  Enable assertion in every Class of pack1 except Class B.

- **java -ea:pack1… -da:pack1.pack2…**
  Enable assertion in every Class of pack1 and disable assertions for every class of pack2.

Note: We can enable and disable assertions either class wise or package wise also.

Appropriate and Inappropriate use of assertions

- It's always inappropriate to define programming logic with assert statement or use assert statement in a programming business logic.
- While performing debugging, in our program if there is any place where the control is not allowed to reach. That is the best place to use assertion. For example – default in switch statement.
- It is always inappropriate for validating public method argument for assertions because outside person doesn't aware weather assertion is enabled or disabled in our system. It is always appropriate to validate the private method argument using assert statement because local person can aware weather assertion is enabled or disabled in our system.
- It is always inappropriate to validating command line arguments using assertions because these are arguments to main() method which is public.

AssertionError

- It is the child class of Error present in java.lang package and hence it is Unchecked.
- If assert statement fails i.e., if argument is false then we will get AssertionError.
- Getting an Assertion Error is a fastest way to debug a program.
- Even though it is legal to catch AssertionError but it is not a good programming practice.

Note: If we are working on IDE not executing from CMD, then we can enable the checkbox to run the program in debug mode which will internally run the program with enabling assertions.

Various Methods of Assertions

- **void assertEquals(boolean expected, boolean actual)**
  Checks that two primitives/objects are equal.

- **void assertTrue(boolean condition)**
  Checks that a condition is true.

- **void assertFalse(boolean condition)**
  Checks that a condition is false.

- **void assertNotNull(Object object)**
  Checks that an object isn't null.

- **void assertNull(Object object)**
  Checks that an object is null.

- **void assertSame(object1, object2)**
  The assertSame() method tests if two object references point to the same object.

- **void assertNotSame(object1, object2)**
  The assertNotSame() method tests if two object references do not point to the same object.

- **void assertArrayEquals(expectedArray, resultArray);**
  The assertArrayEquals() method will test whether two arrays are equal to each other.

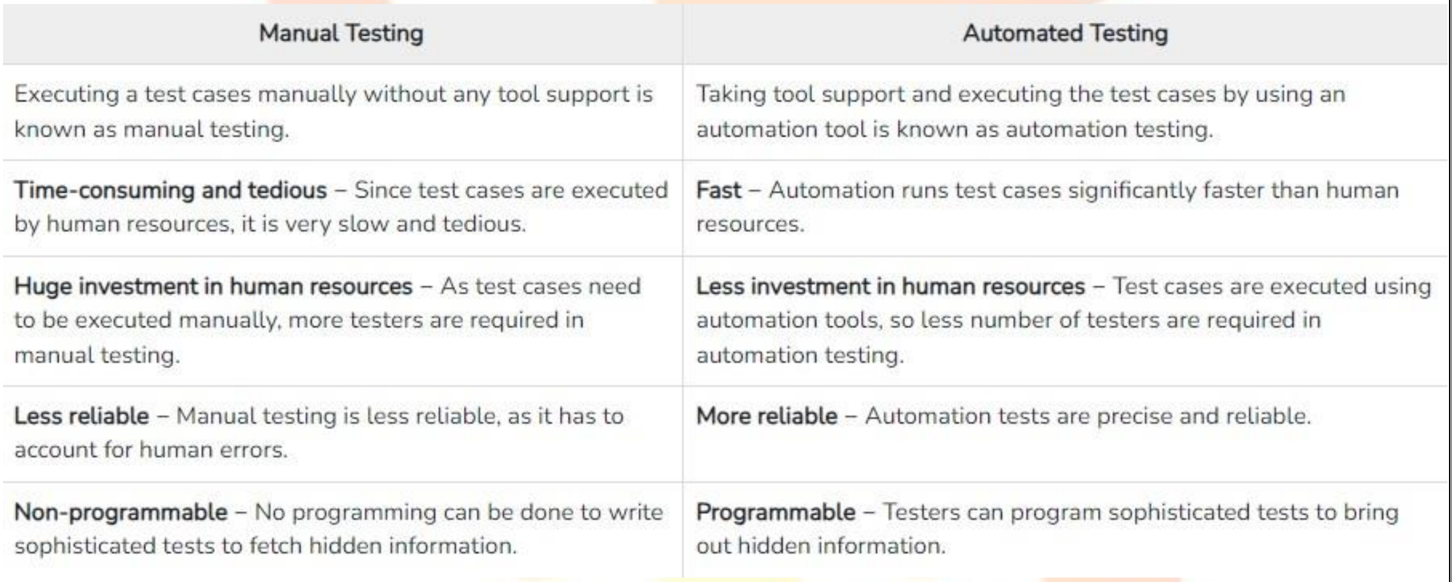
2.. Junit

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.

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.

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.



Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing.	Taking tool support and executing the test cases by using an automation tool is known as automation testing.
Time-consuming and tedious – Since test cases are executed by human resources, it is very slow and tedious.	**Fast** – Automation runs test cases significantly faster than human resources.
Huge investment in human resources – As test cases need to be executed manually, more testers are required in manual testing.	**Less investment in human resources** – Test cases are executed using automation tools, so less number of testers are required in automation testing.
Less reliable – Manual testing is less reliable, as it has to account for human errors.	**More reliable** – Automation tests are precise and reliable.
Non-programmable – No programming can be done to write sophisticated tests to fetch hidden information.	**Programmable** – Testers can program sophisticated tests to bring out hidden information.

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.

Components of JUnit Test Framework

JUnit test framework Consists of Following −

- Fixtures
- Test suites
- Test runners
- JUnit classes

Fixtures

Fixtures is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable. It includes −

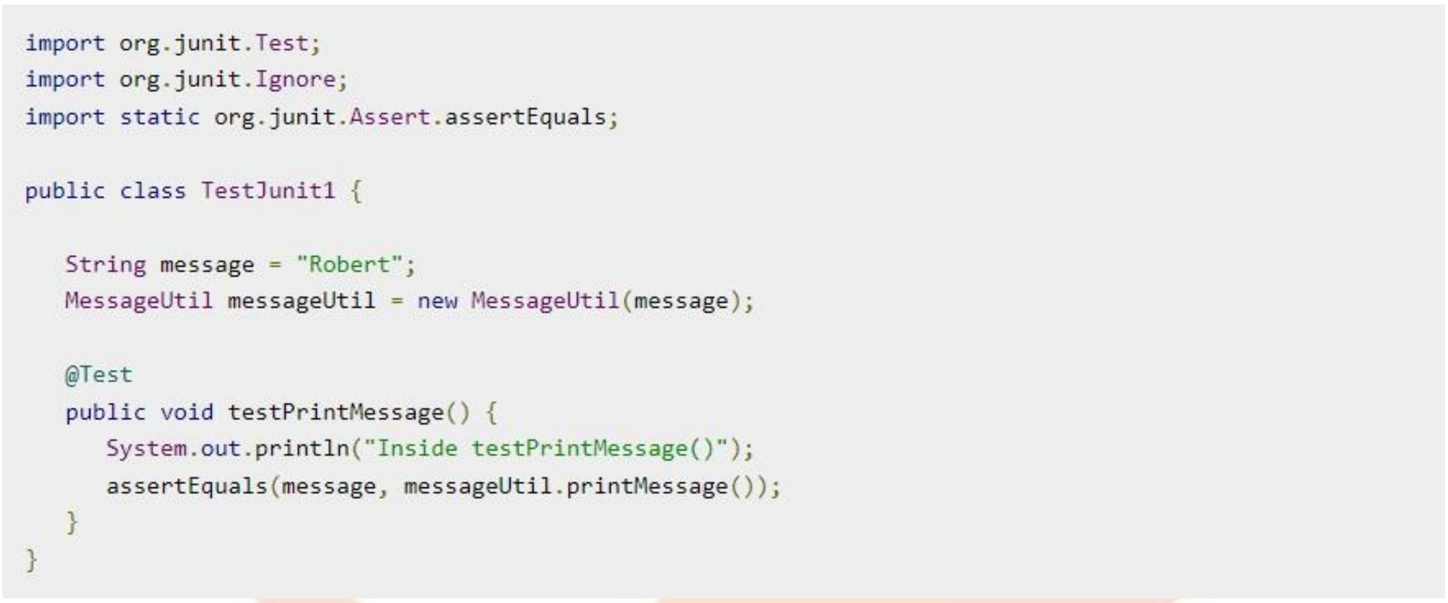
- setUp() method, which runs before every test invocation.
- tearDown() method, which runs after every test method.

```
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

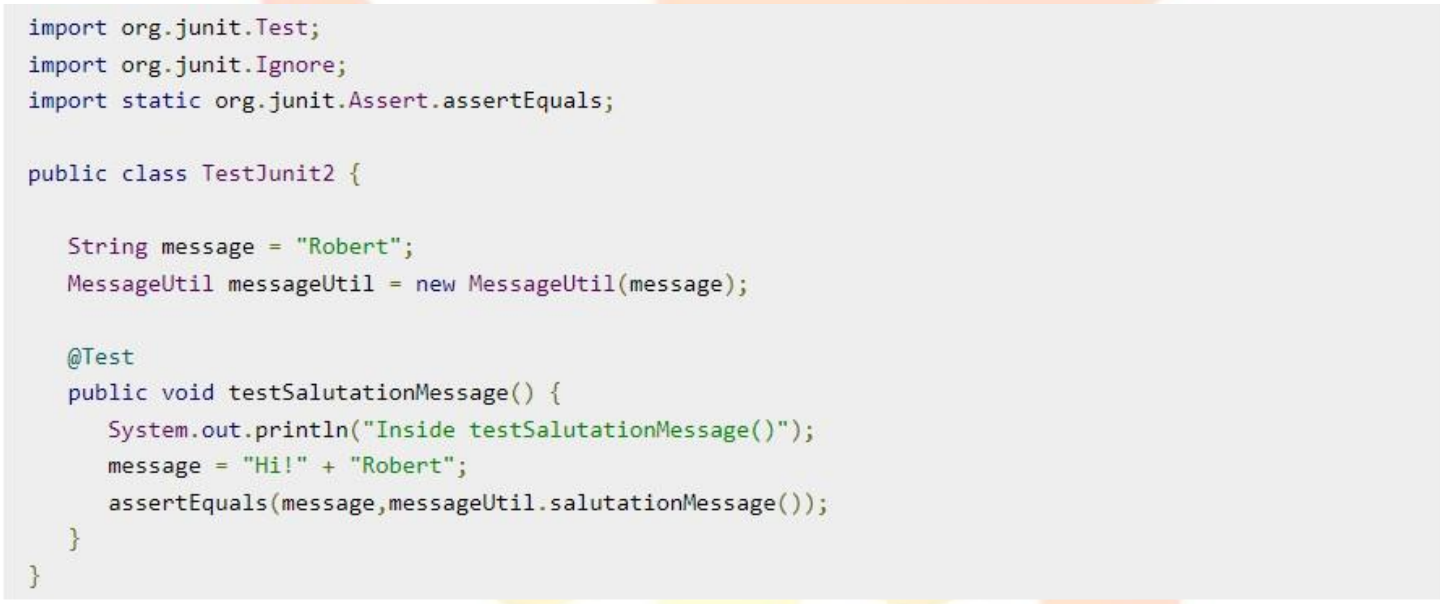
Test Suites

A test suite bundles a few unit test cases and runs them together. In JUnit, both **@RunWith** and **@Suite** annotation are used to run the suite test. Given below is an example that uses TestJunit1 & TestJunit2 test classes.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

//JUnit Suite Test
@RunWith(Suite.class)

@Suite.SuiteClasses({
   TestJunit1.class ,TestJunit2.class
})

public class JunitTestSuite {
}
```



```java
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit1 {

   String message = "Robert";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      assertEquals(message, messageUtil.printMessage());
   }
}
```



```java
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit2 {

   String message = "Robert";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testSalutationMessage() {
      System.out.println("Inside testSalutationMessage()");
      message = "Hi!" + "Robert";
      assertEquals(message,messageUtil.salutationMessage());
   }
}
```

Test Runners

Test runner is used for executing the test cases. Here is an example that assumes the test class TestJunit already exists.

JUnit Classes

JUnit classes are important classes, used in writing and testing JUnits. Some of the important classes are –

- Assert – Contains a set of assert methods.
- TestCase – Contains a test case that defines the fixture to run multiple tests.
- TestResult – Contains methods to collect the results of executing a test case.

Annotations Used in JUnit

- **@Test**
  The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.

- **@Before**
  Several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before each Test method.

- **@After**
  If you allocate external resources in a Before method, you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method.

- **@BeforeClass**
  Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class.

- **@AfterClass**
  This will perform the method after all tests have finished. This can be used to perform clean-up activities.

- **@Ignore**
  The Ignore annotation is used to ignore the test and that test will not be executed.