JUnit Tutorial

JUNIT TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

JUnit Tutorial

JUnit is a unit testing framework for the 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.

This tutorial will teach you how to use JUnit in your day-2-day life of any project unit testing while working with Java programming language.

Audience

This tutorial has been prepared for the beginners to help them understand basic functionality of JUnit tool. After completing this tutorial you will find yourself at a moderate level of expertise in using JUnit testing framework from where you can take yourself to next levels.

Prerequisites

We assume you are going to use JUnit to handle all levels of Java projects development. So it will be good if you have knowledge of software development using any programming language specially Java programming and software testing process.

Copyright & Disclaimer Notice

Table of Contents

CHAPTER

1

JUnit Overview

Testing is the process of checking the functionality of the application whether it is working as per requirements and to ensure that at developer level, unit testing comes into picture.

Unit testing is the testing of single entity (class or method). Unit testing is very essential to every software company to give a quality product to their customers.

Unit testing can be done in two ways

Manual testing	Automated testing
Executing the test cases manually without any tool support is known as manual testing.	Taking tool support and executing the test cases by using automation tool is known as automation testing.
Time consuming and tedious: Since test cases are executed by human resources so 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 so more testers are required in manual testing.	Less investment in human resources:Test cases are executed by using automation tool so less tester are required in automation testing.
Less reliable: Manual testing is less reliable as tests may not be performed with precision each time because of human errors.	More reliable: Automation tests perform precisely same operation each time they are run.
Non-programmable: No programming can be done to write sophisticated tests which fetch hidden information.	Programmable: Testers can program sophisticated tests to bring out hidden information.

What is JUnit ?

JUnit is a unit testing framework for the Java Programming Language. It is important in the test driven development, and is one of a family of unit testing frameworks collectively known as xUnit.

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

Features

- JUnit is an open source framework which is used for writing & running tests.
- Provides Annotation to identify the test methods.
- Provides Assertions for testing expected results.
- Provides Test runners for running tests.
- JUnit tests allow you to write code faster which increasing quality
- JUnit is elegantly simple. It is less complex & 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 test is going fine 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 the another part of code (method) works as expected. To achieve those desired results quickly, test framework is required .JUnit is perfect unit test framework for java programming language.

A formal written test-case is characterized by a known input and by 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 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.

CHAPTER

2

JUnit Environment Setup

JUnit is a framework for Java, so the very first requirement is to have JDK installed in your machine.

System Requirement

JDK	1.5 or above.
Memory	no minimum requirement.
Disk Space	no minimum requirement.
Operating System	no minimum requirement.

Step 1 - verify Java installation in your machine

Now open console and execute the following **java** command.

OS	Task	Command
Windows	Open Command Console	c:\> java -version
Linux	Open Command Terminal	$ java -version
Mac	Open Terminal	machine:~ joseph$ java -version

Let's verify the output for all the operating systems:

OS	Output
Windows	java version "1.6.0 21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Linux	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0 21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Mac	java version "1.6.0 21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode,

	sharing)

If you do not have Java installed, install the Java Software Development Kit (SDK) fromhttp://www.oracle.com/technetwork/java/javase/downloads/index.html. We are assuming Java 1.6.0_21 as installed version for this tutorial.

Step 2: Set JAVA environment

Set the **JAVA_HOME** environment variable to point to the base directory location where Java is installed on your machine. For example

OS	Output
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	export JAVA HOME=/usr/local/java-current
Mac	export JAVA HOME=/Library/Java/Home

Append Java compiler location to System Path.

OS	Output
Windows	Append the string ;C:\Program Files\Java\jdk1.6.0_21\bin to the end of the system variable, Path.
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac	not required

Verify Java Installation using **java -version** command explained above.

Step 3: Download Junit archive

Download latest version of JUnit jar file from http://www.junit.org. At the time of writing this tutorial, I downloaded *Junit-4.10.jar* and copied it into C:\>JUnit folder.

OS	Archive name
Windows	junit4.10.jar
Linux	junit4.10.jar
Mac	junit4.10.jar

Step 4: Set JUnit environment

Set the **JUNIT_HOME** environment variable to point to the base directory location where JUNIT jar is stored on your machine. Assuming, we've stored junit4.10.jar in JUNIT folder on various Operating Systems as follows.

OS	Output
Windows	Set the environment variable JUNIT_HOME to C:\JUNIT
Linux	export JUNIT_HOME=/usr/local/JUNIT
Mac	export JUNIT_HOME=/Library/JUNIT

Step 5: Set CLASSPATH variable

Set the **CLASSPATH** environment variable to point to the JUNIT jar location. Assuming, we've stored junit4.10.jar in JUNIT folder on various Operating Systems as follows.

OS	Output
Windows	Set the environment variable CLASSPATH to %CLASSPATH%;%JUNIT_HOME%\junit4.10.jar;.;
Linux	export CLASSPATH=$CLASSPATH:$JUNIT_HOME/junit4.10.jar:.
Mac	export CLASSPATH=$CLASSPATH:$JUNIT HOME/junit4.10.jar:.

Step 6: Test JUnit Setup

Create a java class file name TestJunit in **C:\ > JUNIT_WORKSPACE**

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestJunit {
   @Test
   public void testAdd() {
      String str= "Junit is working fine";
      assertEquals("Junit is working fine",str);
   }
}
```

Create a java class file name TestRunner in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Step 7: Verify the Result

Compile the classes using **javac** compiler as follows

```
C:\JUNIT_WORKSPACE>javac TestJunit.java TestRunner.java
```

Now run the Test Runner to see the result

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

CHAPTER

3

JUnit Test Framework

JUnit is a **Regression Testing Framework** used by developers to implement unit testing in Java and accelerate programming speed and increase the quality of code. JUnit Framework can be easily integrated with either of the followings:

- Eclipse
- Ant
- Maven

Features

JUnit test framework provides following important features

- 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.

Let's check one example:

```
import junit.framework.*;
```

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

   // assigning the values
   protected void setUp(){
      value1=3;
      value2=3;
   }

   // test method to add two values
   public void testAdd(){
      double result= value1 + value2;
      assertTrue(result == 6);
   }
}
```

Test suite

Test suite means bundle a few unit test cases and run it together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test. Here is an example which 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 {
}
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());
   }
}
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 runner

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

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

JUnit classes

JUnit classes are important classes which is used in writing and testing JUnits. Some of the important classes are

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

CHAPTER

4

JUnit Basic Usage

Now we'll show you a step by step process to get a kick start in JUnit using a basic example.

Create a Class

- Create a java class to be tested say MessageUtil.java in C:\ > JUNIT_WORKSPACE

```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public String printMessage(){
      System.out.println(message);
      return message;
   }
}
```

Create Test Case Class

- Create a java test class say TestJunit.java.
- Add a test method testPrintMessage() to your test class.
- Add an Annotaion @Test to method testPrintMessage().
- Implement the test condition and check the condition using assertEquals API of Junit.

Create a java class file name TestJunit.java in **C:\ > JUNIT_WORKSPACE**

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestJunit {

   String message = "Hello World";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      assertEquals(message,messageUtil.printMessage());
   }
}
```

Create Test Runner Class

- Create a TestRunner java class.
- Use runClasses method of JUnitCore class of JUnit to run test case of above created test class
- Get the result of test cases run in Result Object
- Get failure(s) using getFailures() methods of Result object
- Get Success result using wasSuccessful() methods of Result object

Create a java class file name TestRunner.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the MessageUtil, Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac MessageUtil TestJunit.java TestRunner.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
Hello World
true
```

Now update TestJunit in **C:\ > JUNIT_WORKSPACE** so that test fails. Change the message string.

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestJunit {

   String message = "Hello World";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      message = "New Word";
      assertEquals(message,messageUtil.printMessage());
   }
}
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
Hello World
testPrintMessage(TestJunit): expected:<[New Wor]d> but was:<[Hello Worl]d>
```

```
false
```

CHAPTER

5

JUnit API

The most important package in JUnit is **junit.framework** which contain all the core classses. Some of the important class are:

Serial No	Class Name	Functionality
1	Assert	A set of assert methods.
2	TestCase	A test case defines the fixture to run multiple tests.
3	TestResult	A TestResult collects the results of executing a test case.
4	TestSuite	A TestSuite is a Composite of Tests.

Assert Class

Following is the declaration for **org.junit.Assert** class:

```
public class Assert extends java.lang.Object
```

This class provides a set of assertion methods useful for writing tests. Only failed assertions are recorded. Some of the important methods of **Assert** class are:

S.N.	Methods & Description
1	**void assertEquals(boolean expected, boolean actual)** Check that two primitives/Objects are equal
2	**void assertFalse(boolean condition)** Check that a condition is false
3	**void assertNotNull(Object object)** Check that an object isn't null.
4	**void assertNull(Object object)** Check that an object is null
5	**void assertTrue(boolean condition)** Check that a condition is true.
6	**void fail()**

	Fails a test with no message.

Let's try to cover few of the above mentioned methods in an example. Create a java class file name TestJunit1.java in **C:\ > JUNIT_WORKSPACE**

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestJunit1 {
   @Test
   public void testAdd() {
      //test data
      int num= 5;
      String temp= null;
      String str= "Junit is working fine";

      //check for equality
      assertEquals("Junit is working fine", str);

      //check for false condition
      assertFalse(num > 6);

      //check for not null value
      assertNotNull(str);
   }
}
```

Next, let's create a java class file name TestRunner1.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner1 {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit1.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac TestJunit1.java TestRunner1.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner1
```

Verify the output.

```
true
```

TestCase Class

Following is the declaration for **org.junit.TestCaset** class:

```
public abstract class TestCase extends Assert implements Test
```

A test case defines the fixture to run multiple tests. Some of the important methods of **TestCase** class are

S.N.	Methods & Description
1	**int countTestCases()** Counts the number of test cases executed by run(TestResult result).
2	**TestResult createResult()** Creates a default TestResult object.
3	**String getName()** Gets the name of a TestCase.
4	**TestResult run()** A convenience method to run this test, collecting the results with a default TestResult object.
5	**void run(TestResult result)** Runs the test case and collects the results in TestResult.
6	**void setName(String name)** Sets the name of a TestCase.
7	**void setUp()** Sets up the fixture, for example, open a network connection.
8	**void tearDown()** Tears down the fixture, for example, close a network connection.
9	**String toString()** Returns a string representation of the test case.

Let's try to cover few of the above mentioned methods in an example. Create a java class file name TestJunit2.java in **C:\ > JUNIT_WORKSPACE**

```
import junit.framework.TestCase;
import org.junit.Before;
import org.junit.Test;
public class TestJunit2 extends TestCase  {
   protected double fValue1;
   protected double fValue2;

   @Before
   public void setUp() {
      fValue1= 2.0;
      fValue2= 3.0;
   }

   @Test
   public void testAdd() {

      //Count the number of test cases
            System.out.println("No of Test Case = "+ this.countTestCases());

          //test getName
          String name= this.getName();
          System.out.println("Test Case Name = "+ name);

          //test setName
```

```
            this.setName("testNewAdd");
            String newName= this.getName();
            System.out.println("Updated Test Case Name = "+ newName);
   }
   //tearDown used to close the connection or clean up activities
   public void tearDown(  ) {
   }
}
```

Next, let's create a java class file name TestRunner2.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner2 {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit2.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac TestJunit2.java TestRunner2.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner2
```

Verify the output.

```
No of Test Case = 1
Test Case Name = testAdd
Updated Test Case Name = testNewAdd
true
```

TestResult Class

Following is the declaration for **org.junit.TestResult** class:

```
public class TestResult extends Object
```

A TestResult collects the results of executing a test case. It is an instance of the Collecting Parameter pattern. The test framework distinguishes between failures and errors. A failure is anticipated and checked for with assertions. Errors are unanticipated problems like an ArrayIndexOutOfBoundsException. Some of the important methods of **TestResult** class are:

S.N.	Methods & Description
1	**void addError(Test test, Throwable t)** Adds an error to the list of errors.

2	**void addFailure(Test test, AssertionFailedError t)** Adds a failure to the list of failures.
3	**void endTest(Test test)** Informs the result that a test was completed.
4	**int errorCount()** Gets the number of detected errors.
5	**Enumeration<TestFailure> errors()** Returns an Enumeration for the errors.
6	**int failureCount()** Gets the number of detected failures.
7	**void run(TestCase test)** Runs a TestCase.
8	**int int runCount()** Gets the number of run tests.
9	**void startTest(Test test)** Informs the result that a test will be started.
10	**void stop()** Marks that the test run should stop.

Create a java class file name TestJunit3.java in **C:\ > JUNIT_WORKSPACE**

```
import org.junit.Test;
import junit.framework.AssertionFailedError;
import junit.framework.TestResult;

public class TestJunit3 extends TestResult {
   // add the error
   public synchronized void addError(Test test, Throwable t) {
      super.addError((junit.framework.Test) test, t);
   }

   // add the failure
   public synchronized void addFailure(Test test, AssertionFailedError t) {
      super.addFailure((junit.framework.Test) test, t);
   }
   @Test
   public void testAdd() {
   // add any test
   }

   // Marks that the test run should stop.
   public synchronized void stop() {
   //stop the test here
   }
}
```

Next, let's create a java class file name TestRunner3.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner3 {
   public static void main(String[] args) {
```

```
      Result result = JUnitCore.runClasses(TestJunit3.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac TestJunit3.java TestRunner3.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner3
```

Verify the output.

```
true
```

TestSuite Class

Following is the declaration for **org.junit.TestSuite** class:

```
public class TestSuite extends Object implements Test
```

A TestSuite is a Composite of Tests. It runs a collection of test cases. Some of the important methods of **TestSuite** class are

S.N.	Methods & Description
1	**void addTest(Test test)** Adds a test to the suite.
2	**void addTestSuite(Class<? extends TestCase> testClass)** Adds the tests from the given class to the suite.
3	**int countTestCases()** Counts the number of test cases that will be run by this test.
4	**String getName()** Returns the name of the suite.
5	**void run(TestResult result)** Runs the tests and collects their result in a TestResult.
6	**void setName(String name)** Sets the name of the suite.
7	**Test testAt(int index)** Returns the test at the given index.
8	**int testCount()** Returns the number of tests in this suite.

9	**static Test warning(String message)** Returns a test which will fail and log a warning message.

Create a java class file name JunitTestSuite.java in **C:\ > JUNIT_WORKSPACE** to create Test suite

```
import junit.framework.*;
public class JunitTestSuite {
   public static void main(String[] a) {
      // add the test's in the suite
      TestSuite suite = new TestSuite(TestJunit1.class,
      TestJunit2.class, TestJunit3.class );
      TestResult result = new TestResult();
      suite.run(result);
      System.out.println("Number of test cases = " + result.runCount());
    }
}
```

Compile the Test suite classes using javac

```
C:\JUNIT_WORKSPACE>javac JunitTestSuite.java
```

Now run the Test Suite.

```
C:\JUNIT_WORKSPACE>java JunitTestSuite
```

Verify the output.

```
No of Test Case = 1
Test Case Name = testAdd
Updated Test Case Name = testNewAdd
Number of test cases = 3
```

CHAPTER

6

JUnit Writing Tests

Here we will see one complete example of JUnit testing using POJO class, Business logic class and a test class which will be run by test runner.

Create **EmployeeDetails.java** in **C:\ > JUNIT_WORKSPACE** which is a POJO class.

```
public class EmployeeDetails {

   private String name;
   private double monthlySalary;
   private int age;

   /**
   * @return the name
   */
   public String getName() {
      return name;
   }
   /**
   * @param name the name to set
   */
   public void setName(String name) {
      this.name = name;
   }
   /**
   * @return the monthlySalary
   */
   public double getMonthlySalary() {
      return monthlySalary;
   }
   /**
   * @param monthlySalary the monthlySalary to set
   */
   public void setMonthlySalary(double monthlySalary) {
      this.monthlySalary = monthlySalary;
   }
   /**
   * @return the age
   */
   public int getAge() {
      return age;
   }
   /**
   * @param age the age to set
   */
   public void setAge(int age) {
```

```
   this.age = age;
   }
}
```

EmployeeDetails class is used to

- get/set the value of employee's name.
- get/set the value of employee's monthly salary.
- get/set the value of employee's age.

Create a **EmpBusinessLogic.java** in **C:\ > JUNIT_WORKSPACE** which contains business logic

```
public class EmpBusinessLogic {
   // Calculate the yearly salary of employee
   public double calculateYearlySalary(EmployeeDetails employeeDetails){
      double yearlySalary=0;
      yearlySalary = employeeDetails.getMonthlySalary() * 12;
      return yearlySalary;
   }

   // Calculate the appraisal amount of employee
   public double calculateAppraisal(EmployeeDetails employeeDetails){
      double appraisal=0;
      if(employeeDetails.getMonthlySalary() < 10000){
         appraisal = 500;
      }else{
         appraisal = 1000;
      }
      return appraisal;
   }
}
```

EmpBusinessLogic class is used for calculating

- the yearly salary of employee.
- the appraisal amount of employee.

Create a **TestEmployeeDetails.java** in **C:\ > JUNIT_WORKSPACE** which contains test cases to be tested

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

public class TestEmployeeDetails {
   EmpBusinessLogic empBusinessLogic =new EmpBusinessLogic();
   EmployeeDetails employee = new EmployeeDetails();

   //test to check appraisal
   @Test
   public void testCalculateAppriasal() {
      employee.setName("Rajeev");
      employee.setAge(25);
      employee.setMonthlySalary(8000);
      double appraisal= empBusinessLogic.calculateAppraisal(employee);
      assertEquals(500, appraisal, 0.0);
   }

   // test to check yearly salary
```

```
   @Test
   public void testCalculateYearlySalary() {
      employee.setName("Rajeev");
      employee.setAge(25);
      employee.setMonthlySalary(8000);
      double salary= empBusinessLogic.calculateYearlySalary(employee);
      assertEquals(96000, salary, 0.0);
   }
}
```

TestEmployeeDetails class is used for testing the methods of **EmpBusinessLogic** class. It

- tests the yearly salary of the employee.
- tests the appraisal amount of the employee.

Next, let's create a java class file name **TestRunner.java** in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestEmployeeDetails.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac EmployeeDetails.java
EmpBusinessLogic.java TestEmployeeDetails.java TestRunner.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

CHAPTER

7

JUnit Using Assertion

All the assertion are in the Assert class.

```
public class Assert extends java.lang.Object
```

This class provides a set of assertion methods useful for writing tests. Only failed assertions are recorded. Some of the important methods of **Assert** class are:

S.N.	Methods & Description
1	**void assertEquals(boolean expected, boolean actual)** Check that two primitives/Objects are equal
2	**void assertTrue(boolean expected, boolean actual)** Check that a condition is true
3	**void assertFalse(boolean condition)** Check that a condition is false
4	**void assertNotNull(Object object)** Check that an object isn't null.
5	**void assertNull(Object object)** Check that an object is null
6	**void assertSame(boolean condition)** The assertSame() methods tests if two object references point to the same object
7	**void assertNotSame(boolean condition)** The assertNotSame() methods tests if two object references not point to the same object
8	**void assertArrayEquals(expectedArray, resultArray);** The assertArrayEquals() method will test whether two arrays are equal to each other.

Let's try to cover all of the above mentioned methods in an example. Create a java class file name TestAssertions.java in **C:\ > JUNIT_WORKSPACE**

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestAssertions {
```

```
   @Test
   public void testAssertions() {
      //test data
      String str1 = new String ("abc");
      String str2 = new String ("abc");
      String str3 = null;
      String str4 = "abc";
      String str5 = "abc";
      int val1 = 5;
      int val2 = 6;
      String[] expectedArray = {"one", "two", "three"};
      String[] resultArray =  {"one", "two", "three"};

      //Check that two objects are equal
      assertEquals(str1, str2);

      //Check that a condition is true
      assertTrue (val1 < val2);

      //Check that a condition is false
      assertFalse(val1 > val2);

      //Check that an object isn't null
      assertNotNull(str1);

      //Check that an object is null
      assertNull(str3);

      //Check if two object references point to the same object
      assertSame(str4,str5);

      //Check if two object references not point to the same object
      assertNotSame(str1,str3);

      //Check whether two arrays are equal to each other.
      assertArrayEquals(expectedArray, resultArray);
   }
}
```

Next, let's create a java class file name **TestRunner.java** in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner2 {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestAssertions.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac TestAssertions.java TestRunner.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
true
```

Annotation

Annotations are like meta-tags that you can add to you code and apply them to methods or in class. These annotation in JUnit gives us information about test methods , which methods are going to run before & after test methods, which methods run before & after all the methods, which methods or class will be ignore during execution. List of annotations and their meaning in JUnit :

S.N.	Annotation & Description
1	**@Test** The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.
2	**@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.
3	**@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.
4	**@BeforeClass** Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class.
5	**@AfterClass** This will perform the method after all tests have finished. This can be used to perform clean-up activities.
6	**@Ignore** The Ignore annotation is used to ignore the test and that test will not be executed.

Create a java class file name JunitAnnotation.java in **C:\ > JUNIT_WORKSPACE** to test annotation

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class JunitAnnotation {

   //execute before class
   @BeforeClass
   public static void beforeClass() {
      System.out.println("in before class");
   }

   //execute after class
   @AfterClass
   public static void  afterClass() {
      System.out.println("in after class");
```

```
   }

   //execute before test
   @Before
   public void before() {
      System.out.println("in before");
   }
   //execute after test
   @After
   public void after() {
      System.out.println("in after");
   }
   //test case
   @Test
   public void test() {
      System.out.println("in test");
   }

   //test case ignore and will not execute
   @Ignore
   public void ignoreTest() {
      System.out.println("in ignore test");
   }
}
```

Next, let's create a java class file name **TestRunner.java** in **C:\ > JUNIT_WORKSPACE** to execute annotations

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(JunitAnnotation.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac JunitAnnotation.java TestRunner.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
in before class
in before
in test
in after
in after class
true
```

CHAPTER

8

JUnit Execution Procedure

This tutorial explains the execution procedure of methods in JUnit which means that which method is called first and which one after that. Here is the execution procedure of the JUint test API methods with the example.

Create a java class file name JunitAnnotation.java in **C:\ > JUNIT_WORKSPACE** to test annotation

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class ExecutionProcedureJunit {

   //execute only once, in the starting
   @BeforeClass
   public static void beforeClass() {
      System.out.println("in before class");
   }

   //execute only once, in the end
   @AfterClass
   public static void  afterClass() {
      System.out.println("in after class");
   }

   //execute for each test, before executing test
   @Before
   public void before() {
      System.out.println("in before");
   }

   //execute for each test, after executing test
   @After
   public void after() {
      System.out.println("in after");
   }

   //test case 1
   @Test
   public void testCase1() {
      System.out.println("in test case 1");
   }
```

```
   //test case 2
   @Test
   public void testCase2() {
      System.out.println("in test case 2");
   }
}
```

Next, let's create a java class file name **TestRunner.java** in **C:\ > JUNIT_WORKSPACE** to execute annotations

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(ExecutionProcedureJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac ExecutionProcedureJunit.java TestRunner.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
in before class
in before
in test case 1
in after
in before
in test case 2
in after
in after class
```

See the above output and this is how the JUnit execution procedure is.

- First of all beforeClass() method execute only once
- Lastly, the afterClass() method executes only once.
- before() method executes for each test case but before executing the test case.
- after() method executes for each test case but after the execution of test case
- In between before() and after() each test case executes.

CHAPTER

9

JUnit Executing Tests

The test cases are executed using **JUnitCore** class. JUnitCore is a facade for running tests.

It supports running JUnit 4 tests, JUnit 3.8.x tests, and mixtures. To run tests from the command line, run java org.junit.runner.JUnitCore <TestClass>. For one-shot test runs, use the static method runClasses(Class[]).

Following is the declaration for **org.junit.runner.JUnitCore** class:

```
public class JUnitCore extends java.lang.Object
```

Here we will see how can we execute the tests with the help of JUnitCore.

Create a Class

- Create a java class to be tested say MessageUtil.java in C:\ > JUNIT_WORKSPACE

```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public String printMessage(){
      System.out.println(message);
      return message;
   }
}
```

Create Test Case Class

- Create a java test class say TestJunit.java.

- Add a test method testPrintMessage() to your test class.
- Add an Annotaion @Test to method testPrintMessage().
- Implement the test condition and check the condition using assertEquals API of Junit.

Create a java class file name TestJunit.java in **C:\ > JUNIT_WORKSPACE**

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestJunit {

   String message = "Hello World";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      assertEquals(message,messageUtil.printMessage());
   }
}
```

Create Test Runner Class

Next, let's create a java class file name TestRunner.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s) which import the JUnitCore class and uses the runClasses() method which take the test class name as parameter.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
Hello World
true
```

CHAPTER

10

JUnit Test Suite

Test suite means bundle a few unit test cases and run it together. In JUnit, both **@RunWith** and **@Suite** annotation are used to run the suite test. This tutorial will show you an example having two TestJunit1 & TestJunit2 test classes to run together using Test Suite.

Create a Class

Create a java class to be tested say MessageUtil.java in **C:\ > JUNIT_WORKSPACE**

```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public void printMessage(){
      System.out.println(message);
      return message;
   }

   // add "Hi!" to the message
   public String salutationMessage(){
      message = "Hi!" + message;
      System.out.println(message);
      return message;
   }
}
```

Create Test Case Classes

Create a java class file name TestJunit1.java in **C:\ > JUNIT_WORKSPACE**

```
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());
   }
}
```

Create a java class file name TestJunit2.java in **C:\ > JUNIT_WORKSPACE**

```
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());
   }
}
```

Create Test Suite Class

- Create a java class.
- Attach @RunWith(Suite.class) Annotation with class.
- Add reference to Junit test classes using @Suite.SuiteClasses annotation

Create a java class file name TestSuite.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
   TestJunit1.class,
   TestJunit2.class
})
public class JunitTestSuite {
}
```

Create Test Runner Class

Create a java class file name TestRunner.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
```

```
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(JunitTestSuite.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile all java classes using javac

```
C:\JUNIT_WORKSPACE>javac MessageUtil.java TestJunit1.java
TestJunit2.java JunitTestSuite.java TestRunner.java
```

Now run the Test Runner which will run test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
Inside testPrintMessage()
Robert
Inside testSalutationMessage()
Hi Robert
true
```

CHAPTER

11

JUnit Ignore Test

Sometimes it happens that our code is not ready and test case written to test that method/code will fail if run. The **@Ignore** annotation helps in this regards.

- A test method annotated with @Ignore will not be executed.
- If a test class is annotated with @Ignore then none of its test methods will be executed.
- Now let's see @Ignore in action.

Create a Class

- Create a java class to be tested say MessageUtil.java in C:\ > JUNIT_WORKSPACE

```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public String printMessage(){
      System.out.println(message);
      return message;
   }

   // add "Hi!" to the message
   public String salutationMessage(){
      message = "Hi!" + message;
      System.out.println(message);
      return message;
   }
}
```

Create Test Case Class

- Create a java test class say TestJunit.java.
- Add a test methods testPrintMessage(),testSalutationMessage() to your test class.
- Add an Annotaion @Ignore to method testPrintMessage().

Create a java class file name TestJunit.java in **C:\ > JUNIT_WORKSPACE**

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

public class TestJunit {

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

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

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

Create Test Runner Class

Create a java class file name TestRunner.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the MessageUtil, Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java
```

Now run the Test Runner which will not run testPrintMessage() test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output. testPrintMessage() test case is not tested.

```
Inside testSalutationMessage()
Hi!Robert
true
```

Now update TestJunit in **C:\ > JUNIT_WORKSPACE** to ignore all test cases. Add @Ignore at class level

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

@Ignore
public class TestJunit {

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

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

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

Compile the Test case using javac

```
C:\JUNIT_WORKSPACE>javac TestJunit.java
```

Now run the Test Runner which will not run any test case defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output. No test case is tested.

```
true
```

CHAPTER

12

JUnit Time Test

Junit provides a handy option of Timeout. If a test case takes more time than specified number of milliseconds then Junit will automatically mark it as failed. The **timeout** parameter is used along with @Test annotation. Now let's see @Test(timeout) in action.

Create a Class

- Create a java class to be tested say MessageUtil.java in C:\ > JUNIT_WORKSPACE.
- Add a infinite while loop inside printMessage() method.

```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public void printMessage(){
      System.out.println(message);
      while(true);
   }

   // add "Hi!" to the message
   public String salutationMessage(){
      message = "Hi!" + message;
      System.out.println(message);
      return message;
   }
}
```

Create Test Case Class

- Create a java test class say TestJunit.java.

- Add timeout of 1000 to testPrintMessage() test case.

Create a java class file name TestJunit.java in **C:\ > JUNIT_WORKSPACE**

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

public class TestJunit {

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

   @Test(timeout=1000)
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      messageUtil.printMessage();
   }

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

Create Test Runner Class

Create a java class file name TestRunner.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the MessageUtil, Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java
```

Now run the Test Runner which will run test cases defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output. testPrintMessage() test case will mark unit testing failed.

```
Inside testPrintMessage()
Robert
Inside testSalutationMessage()
Hi!Robert
```

```
testPrintMessage(TestJunit): test timed out after 1000 milliseconds
false
```

CHAPTER

13

JUnit Exception Test

Junit provides a option of tracing the Exception handling of code. You can test the code whether code throws desired exception or not. The **expected** parameter is used along with @Test annotation. Now let's see *@Test(expected)* in action.

Create a Class

- Create a java class to be tested say MessageUtil.java in C:\ > JUNIT_WORKSPACE.
- Add a error condition inside printMessage() method.

```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public void printMessage(){
      System.out.println(message);
      int a =0;
      int b = 1/a;
   }

   // add "Hi!" to the message
   public String salutationMessage(){
      message = "Hi!" + message;
      System.out.println(message);
      return message;
   }
}
```

Create Test Case Class

- Create a java test class say TestJunit.java.

- Add expected exception ArithmeticException to testPrintMessage() test case.

Create a java class file name TestJunit.java in **C:\ > JUNIT_WORKSPACE**

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

public class TestJunit {

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

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

Create Test Runner Class

Create a java class file name TestRunner.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the MessageUtil, Test case and Test Runner classes using javac

```
C:\JUNIT_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java
```

Now run the Test Runner which will run test cases defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output. testPrintMessage() test case will be passed.

```
Inside testPrintMessage()
Robert
Inside testSalutationMessage()
Hi!Robert
true
```

CHAPTER

14

JUnit Parameterized Test

JUnit 4 has introduced a new feature **Parameterized tests**. Parameterized tests allow developer to run the same test over and over again using different values. There are five steps, that you need to follow to create **Parameterized tests**.

- Annotate test class with @RunWith(Parameterized.class)
- Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
- Create a public constructor that takes in what is equivalent to one "row" of test data.
- Create an instance variable for each "column" of test data.
- Create your tests case(s) using the instance variables as the source of the test data.

The test case will be invoked once per each row of data. Let's see Parameterized tests in action.

Create a Class

- Create a java class to be tested say PrimeNumberChecker.java in C:\ > JUNIT_WORKSPACE.

```
public class PrimeNumberChecker {
   public Boolean validate(final Integer primeNumber) {
      for (int i = 2; i < (primeNumber / 2); i++) {
         if (primeNumber % i == 0) {
            return false;
         }
      }
      return true;
   }
}
```

Create Parameterized Test Case Class

- Create a java test class say PrimeNumberCheckerTest.java.

Create a java class file name PrimeNumberCheckerTest.java in **C:\ > JUNIT_WORKSPACE**

```
import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.Before;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;

@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
   private Integer inputNumber;
   private Boolean expectedResult;
   private PrimeNumberChecker primeNumberChecker;

   @Before
   public void initialize() {
      primeNumberChecker = new PrimeNumberChecker();
   }

   // Each parameter should be placed as an argument here
   // Every time runner triggers, it will pass the arguments
   // from parameters we defined in primeNumbers() method
   public PrimeNumberCheckerTest(Integer inputNumber,
      Boolean expectedResult) {
      this.inputNumber = inputNumber;
      this.expectedResult = expectedResult;
   }

   @Parameterized.Parameters
   public static Collection primeNumbers() {
      return Arrays.asList(new Object[][] {
         { 2, true },
         { 6, false },
         { 19, true },
         { 22, false },
         { 23, true }
      });
   }

   // This test will run 4 times since we have 5 parameters defined
   @Test
   public void testPrimeNumberChecker() {
      System.out.println("Parameterized Number is : " + inputNumber);
      assertEquals(expectedResult,
      primeNumberChecker.validate(inputNumber));
   }
}
```

Create Test Runner Class

Create a java class file name TestRunner.java in **C:\ > JUNIT_WORKSPACE** to execute Test case(s)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(PrimeNumberCheckerTest.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
```

```
      System.out.println(result.wasSuccessful());
   }
}
```

Compile the PrimeNumberChecker, PrimeNumberCheckerTest and Test Runner classes using javac

```
C:\JUNIT WORKSPACE>javac PrimeNumberChecker.java PrimeNumberCheckerTest.java
TestRunner.java
```

Now run the Test Runner which will run test cases defined in provided Test Case class.

```
C:\JUNIT_WORKSPACE>java TestRunner
```

Verify the output.

```
Parameterized Number is : 2
Parameterized Number is : 6
Parameterized Number is : 19
Parameterized Number is : 22
Parameterized Number is : 23
true
```

CHAPTER

15

JUnit Plug with ANT

In this example, we will demonstrate how to run JUnit using ANT. Let's follow the given steps:

Step 1: Download Apache Ant

Download Apache Ant

OS	Archive name
Windows	apache-ant-1.8.4-bin.zip
Linux	apache-ant-1.8.4-bin.tar.gz
Mac	apache-ant-1.8.4-bin.tar.gz

Step 2: Set Ant Environment

Set the **ANT_HOME** environment variable to point to the base directory location where ANT libraries is stored on your machine. For example, We've stored Ant libraries in apache-ant-1.8.4 folder on various Operating Systems as follows.

OS	Output
Windows	Set the environment variable ANT_HOME to C:\Program Files\Apache Software Foundation\apache-ant-1.8.4
Linux	export ANT_HOME=/usr/local/\apache-ant-1.8.4
Mac	export ANT_HOME=/Library/\apache-ant-1.8.4

Append Ant compiler location to System Path is as follows for different OS:

OS	Output
Windows	Append the string ;%ANT_HOME\bin to the end of the system variable, Path.
Linux	export PATH=$PATH:$ANT_HOME/bin/
Mac	not required

Step 3: Download Junit Archive

Download JUnit Archive

OS	Archive name
Windows	junit4.10.jar
Linux	junit4.10.jar
Mac	junit4.10.jar

Step 4: Create Project Structure

- Create folder TestJunitWithAnt in C:\ > JUNIT_WORKSPACE
- Create folder src in C:\ > JUNIT_WORKSPACE > TestJunitWithAnt
- Create folder test in C:\ > JUNIT_WORKSPACE > TestJunitWithAnt
- Create folder lib in C:\ > JUNIT_WORKSPACE > TestJunitWithAnt
- Create MessageUtil class in C:\ > JUNIT_WORKSPACE > TestJunitWithAnt > src folder

```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public void printMessage(){
      System.out.println(message);
      return message;
   }

   // add "Hi!" to the message
   public String salutationMessage(){
      message = "Hi!" + message;
      System.out.println(message);
      return message;
   }
}
```

- Create TestMessageUtil class in C:\ > JUNIT_WORKSPACE > TestJunitWithAnt > src folder

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

```
public class TestMessageUtil {

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

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

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

- Copy junit-4.10.jar in C:\ > JUNIT_WORKSPACE > TestJunitWithAnt > lib folder

Create ANT Build.xml

We'll be using **<junit>** task in Ant to execute our Junit test cases.

```
<project name="JunitTest" default="test" basedir=".">
   <property name="testdir" location="test" />
   <property name="srcdir" location="src" />
   <property name="full-compile" value="true" />
   <path id="classpath.base"/>
   <path id="classpath.test">
      <pathelement location="/lib/junit-4.10.jar" />
      <pathelement location="${testdir}" />
      <pathelement location="${srcdir}" />
      <path refid="classpath.base" />
   </path>
   <target name="clean" >
      <delete verbose="${full-compile}">
         <fileset dir="${testdir}" includes="**/*.class" />
      </delete>
   </target>
   <target name="compile" depends="clean">
      <javac srcdir="${srcdir}" destdir="${testdir}"
         verbose="${full-compile}">
         <classpath refid="classpath.test"/>
      </javac>
   </target>
   <target name="test" depends="compile">
      <junit>
         <classpath refid="classpath.test" />
         <formatter type="brief" usefile="false" />
         <test name="TestMessageUtil" />
      </junit>
   </target>
</project>
```

Run the following ant command

```
C:\JUNIT_WORKSPACE\TestJunitWithAnt>ant
```

Verify the output.

```
Buildfile: C:\JUNIT_WORKSPACE\TestJunitWithAnt\build.xml

clean:

compile:
   [javac] Compiling 2 source files to C:\JUNIT_WORKSPACE\TestJunitWithAnt\test
   [javac] [parsing started C:\JUNIT WORKSPACE\TestJunitWithAnt\src\
   MessageUtil.java]
   [javac] [parsing completed 18ms]
   [javac] [parsing started C:\JUNIT_WORKSPACE\TestJunitWithAnt\src\
   TestMessageUtil.java]
   [javac] [parsing completed 2ms]
   [javac] [search path for source files: C:\JUNIT WORKSPACE\
   TestJunitWithAnt\src]
   [javac] [loading java\lang\Object.class(java\lang:Object.class)]
   [javac] [loading java\lang\String.class(java\lang:String.class)]
   [javac] [loading org\junit\Test.class(org\junit:Test.class)]
   [javac] [loading org\junit\Ignore.class(org\junit:Ignore.class)]
   [javac] [loading org\junit\Assert.class(org\junit:Assert.class)]
   [javac] [loading java\lang\annotation\Retention.class
   (java\lang\annotation:Retention.class)]
   [javac] [loading java\lang\annotation\RetentionPolicy.class
   (java\lang\annotation:RetentionPolicy.class)]
   [javac] [loading java\lang\annotation\Target.class
   (java\lang\annotation:Target.class)]
   [javac] [loading java\lang\annotation\ElementType.class
   (java\lang\annotation:ElementType.class)]
   [javac] [loading java\lang\annotation\Annotation.class
   (java\lang\annotation:Annotation.class)]
   [javac] [checking MessageUtil]
   [javac] [loading java\lang\System.class(java\lang:System.class)]
   [javac] [loading java\io\PrintStream.class(java\io:PrintStream.class)]
   [javac] [loading java\io\FilterOutputStream.class
   (java\io:FilterOutputStream.class)]
   [javac] [loading java\io\OutputStream.class(java\io:OutputStream.class)]
   [javac] [loading java\lang\StringBuilder.class
   (java\lang:StringBuilder.class)]
   [javac] [loading java\lang\AbstractStringBuilder.class
   (java\lang:AbstractStringBuilder.class)]
   [javac] [loading java\lang\CharSequence.class(java\lang:CharSequence.class)]
   [javac] [loading java\io\Serializable.class(java\io:Serializable.class)]
   [javac] [loading java\lang\Comparable.class(java\lang:Comparable.class)]
   [javac] [loading java\lang\StringBuffer.class(java\lang:StringBuffer.class)]
   [javac] [wrote C:\JUNIT_WORKSPACE\TestJunitWithAnt\test\MessageUtil.class]
   [javac] [checking TestMessageUtil]
   [javac] [wrote
C:\JUNIT WORKSPACE\TestJunitWithAnt\test\TestMessageUtil.class]
   [javac] [total 281ms]

test:
    [junit] Testsuite: TestMessageUtil
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.008 sec
    [junit]
    [junit] ------------- Standard Output ---------------
    [junit] Inside testPrintMessage()
    [junit] Robert
    [junit] Inside testSalutationMessage()
    [junit] Hi!Robert
    [junit] ------------- ---------------- ---------------

BUILD SUCCESSFUL
Total time: 0 seconds
```

CHAPTER

16

JUnit Plug with Eclipse

To setup JUnit with eclipse following steps need to be followed

Step 1: Download Junit archive

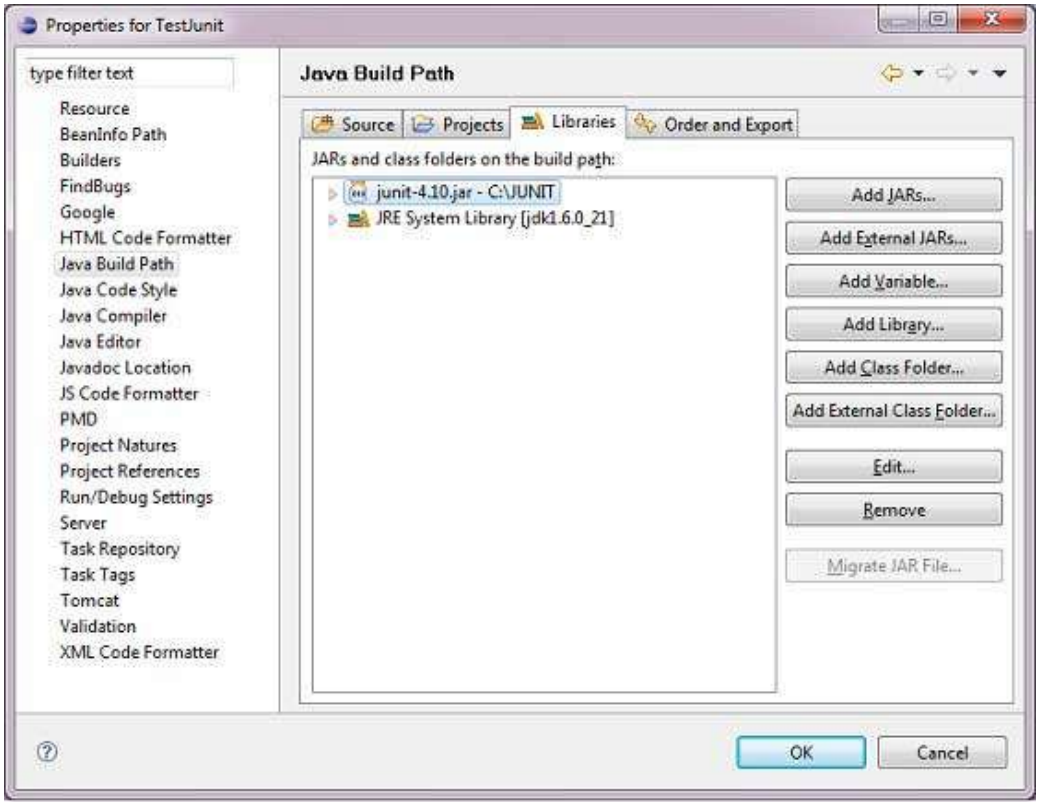
Download JUnit

OS	Archive name
Windows	junit4.10.jar
Linux	junit4.10.jar
Mac	junit4.10.jar

Assume you copied above JAR file in C:\>JUnit folder.

Step 2: Set Eclipse environment

- Open eclipse -> right click on project and click on property > Build Path > Configure Build Path and add the junit-4.10.jar in the libraries using Add External Jar button.



- We assume that your eclipse has inbuilt Junit plugin if it is not available in C:\>eclipse/plugins directory, then you can download it from JUnit Plugin. Unzip the downloaded zip file in the plugin folder of the eclipse. Finally restart eclipse.

- Now your eclipse is ready for the development of JUnit test cases.

Step 3: Verify Junit installation in Eclipse

- Create a projet TestJunit in eclipse at any location.

- Create a class MessageUtil to test in the project

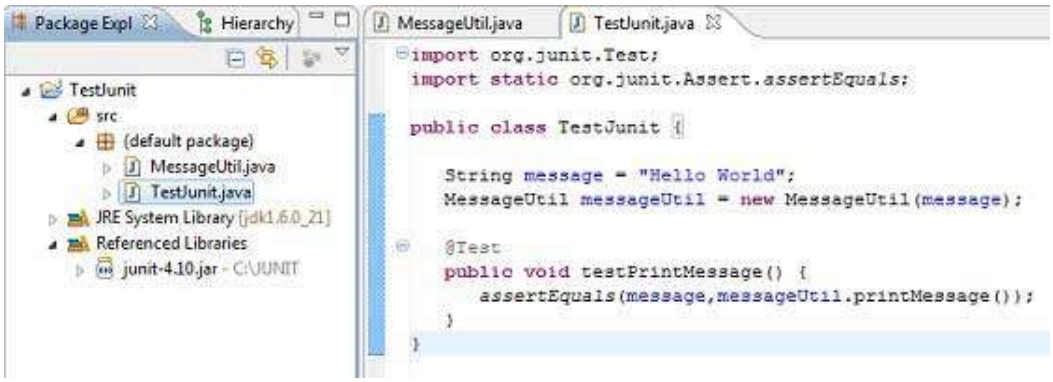
```
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public String printMessage(){
      System.out.println(message);
```

```
      return message;
   }
}
```

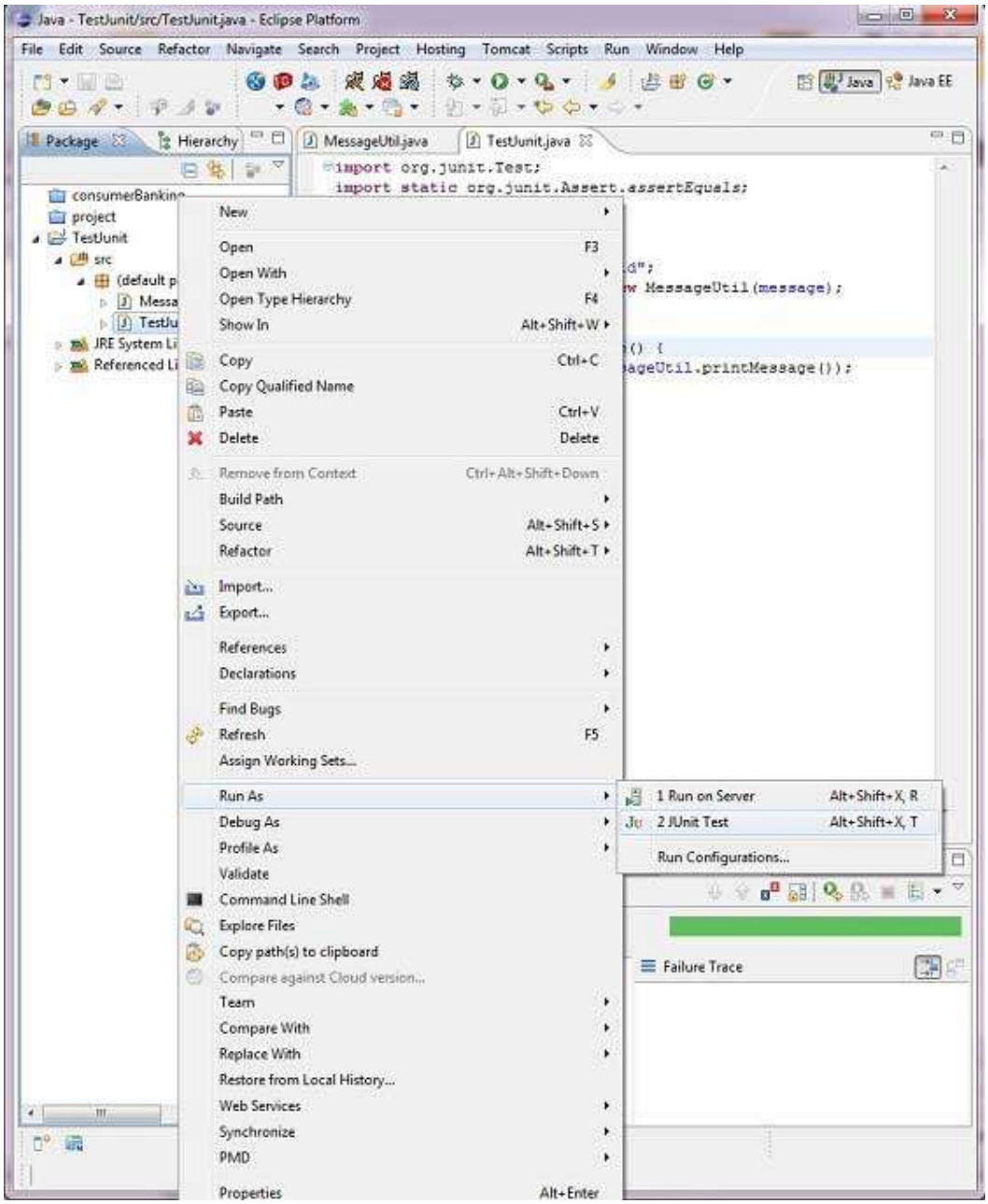
- Create a test class TestJunit in the project

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

public class TestJunit {

   String message = "Hello World";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      assertEquals(message,messageUtil.printMessage());
   }
}
```

Following should be the project structure



Finally, verify the output of the program by right click on program and run as junit



Verify the result

CHAPTER

17

JUnit Extensions

Following are few important JUnit extensions

- Cactus
- JWebUnit
- XMLUnit
- MockObject

Cactus

Cactus is a simple test framework for unit testing server-side java code (Servlets, EJBs, Tag Libs, Filters). The intent of Cactus is to lower the cost of writing tests for server-side code. It uses JUnit and extends it. Cactus implements an in-container strategy, meaning that tests are executed inside the container.

The Cactus Ecosystem is made of several components:

- The Cactus Framework: This is the heart of Cactus. It is the engine that provides the API to write Cactus tests.
- The Cactus Integration Modules: They are front ends and frameworks that provide easy ways of using the Cactus Framework (Ant scripts, Eclipse plugin, Maven plugin).
- The Cactus Samples: They are simple projects that demonstrate how to write Cactus tests and how to use some of the Integration Modules.

JWebUnit

JWebUnit is a Java-based testing framework for web applications. It wraps existing testing frameworks such as HtmlUnit and Selenium with a unified, simple testing interface to allow you to quickly test the correctness of your web applications.

JWebUnit provides a high-level Java API for navigating a web application combined with a set of assertions to verify the application's correctness. This includes navigation via links, form entry and submission, validation of table contents, and other typical business web application features.

The simple navigation methods and ready-to-use assertions allow for more rapid test creation than using only JUnit or HtmlUnit. And if you want to switch from HtmlUnit to other plugins such as Selenium (available soon), there is no need to rewrite your tests.

XMLUnit

XMLUnit provides classes to validate or compare XML files or to assert the value of XPath expressions applied to them. XMLUnit for Java provides integration with JUnit while XMLUNit for .NET integrates with NUnit to simplify unit testing code that generates XML.

MockObject

In a unit test, mock objects can simulate the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test. The common coding style for testing with mock objects is to:

- Create instances of mock objects
- Set state and expectations in the mock objects
- Invoke domain code with mock objects as parameters
- Verify consistency in the mock objects