JUnit | Testing Framework for Java

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 5.

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.

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 5.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:

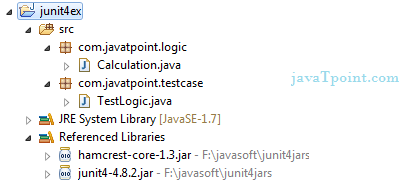
1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object 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.

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.

**package** com.java.logic;

**public** **class** Calculation {

**public** **static** **int** findMax(**int** arr[]){

**int** max=0;

**for**(**int** i=1;i<arr.length;i++){

**if**(max<arr[i])

                max=arr[i];

        }

**return** max;

    }

}

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.

**package** com.java.testcase;

**import** **static** org.junit.Assert.\*;

**import** com.javatpoint.logic.\*;

**import** org.junit.Test;

**public** **class** TestLogic {

    @Test

**public** **void** testFindMax(){

        assertEquals(4,Calculation.findMax(**new** **int**[]{1,3,4,2}));

        assertEquals(-1,Calculation.findMax(**new** **int**[]{-12,-1,-3,-4,-2}));

    }

}

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

**Output:**Assertion Error

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:

**package** com.java.logic;

**public** **class** Calculation {

**public** **static** **int** findMax(**int** arr[]){

**int** max=arr[0];//arr[0] instead of 0

**for**(**int** i=1;i<arr.length;i++){

**if**(max<arr[i])

                max=arr[i];

        }

**return** max;

    }

}

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

Success

Another example of Junit framework

Write the program code

**package** com.java.logic;

**public** **class** Calculation {

    //method that returns maximum number

**public** **static** **int** findMax(**int** arr[]){

**int** max=0;

**for**(**int** i=1;i<arr.length;i++){

**if**(max<arr[i])

                max=arr[i];

        }

**return** max;

    }

  //method that returns cube of the given number

**public** **static** **int** cube(**int** n){

**return** n\*n\*n;

    }

    //method that returns reverse words

**public** **static** String reverseWord(String str){

        StringBuilder result=**new** StringBuilder();

        StringTokenizer tokenizer=**new** StringTokenizer(str," ");

**while**(tokenizer.hasMoreTokens()){

        StringBuilder sb=**new** StringBuilder();

        sb.append(tokenizer.nextToken());

        sb.reverse();

        result.append(sb);

        result.append(" ");

        }

**return** result.toString();

    }

}

Write the test case

**package** com.java.testcase;

**import** **static** org.junit.Assert.assertEquals;

**import** org.junit.After;

**import** org.junit.AfterClass;

**import** org.junit.Before;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

**import** com.javatpoint.logic.Calculation;

**public** **class** TestCase2 {

    @BeforeClass

**public** **static** **void** setUpBeforeClass() **throws** Exception {

        System.out.println("before class");

    }

    @Before

**public** **void** setUp() **throws** Exception {

        System.out.println("before");

    }

    @Test

**public** **void** testFindMax(){

        System.out.println("test case find max");

        assertEquals(4,Calculation.findMax(**new** **int**[]{1,3,4,2}));

        assertEquals(-2,Calculation.findMax(**new** **int**[]{-12,-3,-4,-2}));

    }

    @Test

**public** **void** testCube(){

        System.out.println("test case cube");

        assertEquals(27,Calculation.cube(3));

    }

    @Test

**public** **void** testReverseWord(){

        System.out.println("test case reverse word");

assertEquals("ym eman si nahk",Calculation.reverseWord("my name is khan");

    }

    @After

**public** **void** tearDown() **throws** Exception {

        System.out.println("after");

    }

    @AfterClass

**public** **static** **void** tearDownAfterClass() **throws** Exception {

        System.out.println("after class");

    }

}

**Output:**before class

before

test case find max

after

before

test case cube

after

before

test case reverse word

after

after class

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

* 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 test case(s) using the instance variables as the source of the test data.

The test case will be invoked once for each row of data. Let us see parameterized tests in action.

Create a Class

Create a java class to be tested, say, **PrimeNumberChecker.java**

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**.

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));

}

}

**Test suite** is used to bundle a few unit test cases and run them together. In JUnit, both **@RunWith** and **@Suite** annotations are used to run the suite tests. This chapter takes an example having two test classes, **TestJunit1** & **TestJunit2**, that run together using Test Suite.

## Create a Class

Create a java class to be tested, say, **MessageUtil.java**

/\*

\* 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 Classes

Create a java class file named **TestJunit1.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());

}

}

Create a java class file named **TestJunit2.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());

}

}

## Create Test Suite Class

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

Create a java class file named **TestSuite.java**

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 named **TestRunner.java**

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());

}

}

Output

Inside testPrintMessage()

Robert

Inside testSalutationMessage()

Hi Robert

True

# JUnit – @Timeout

The **@Timeout**annotation in **JUnit**restricts the duration of test and lifecycle methods. It aids in ensuring that a test procedure is completed according to the planned timetable. **TimeoutExceptions** occur when the duration of the test method exceeds the specified limit.

Any JUnit test or lifecycle method can utilize @Timeout. Add the @Timeout annotation to the class declaration to set a **global timeout** for all tests. To override the global timeout, apply the @Timeout annotation to a specific test method.

### There are a few circumstances when using JUnit’s @Timeout annotation might be beneficial:

* **Slow network calls:**If your test procedure makes a network call to an outside service, you might wish to set up a timeout to ensure that the test method fails in the case that the outside service is down or runs slowly.
* **Extended testing:**If your test method takes a long time to finish, you might want to include a timeout to make sure it doesn’t affect the way other tests are run.
* **Deadlocks:**If the test function contains several threads and there’s a potential of a deadlock, you might want to add a timeout to ensure it doesn’t run forever.
* **Infinite loops:**If the test function has an infinite loop, you may wish to specify a timeout to ensure that it does not continue indefinitely. Section 1: Using Timeout() annotation

#### Test 1: Demonstrates using Timeout annotation with Thread.sleep

Checks if a method finishes within 6 seconds using Thread.sleep to delay for 5 seconds. It should pass.

* Java

|  |
| --- |
| // Test 1: Demonstrates using Timeout annotation with Thread.sleep  @Test  @Timeout(6) // Set timeout to 6 seconds  public void test1() throws InterruptedException {      Thread.sleep(5000); // Simulate a 5-second delay      System.out.println("Test1 Passed within the time");  } |

**Test 2: Demonstrates using Timeout annotation with TimeUnit.SECONDS.sleep**

Same as Test 1 but employs TimeUnit.SECONDS.sleep to imitate the delay. It should also pass.

* Java

|  |
| --- |
| // Test 2: Demonstrates using Timeout annotation with TimeUnit.SECONDS.sleep  @Test  @Timeout(6) // Set timeout to 6 seconds  public void test2() throws InterruptedException {      TimeUnit.SECONDS.sleep(5); // Another way to simulate a 5-second delay      System.out.println("Test2 Passed within the time");  } |

### Section 2: Using Assertions

#### Test 3: Demonstrates using Assertions.assertTimeout

Uses Assertions.assertTimeout to confirm a procedure completes inside 7 seconds. It mimics a 6-second delay using a helper method and should pass.

* Java

|  |
| --- |
| // Test 3: Demonstrates using Assertions.assertTimeout  @Test  public void test3() {      Assertions.assertTimeout(Duration.ofSeconds(7), () -> delaySeconds(6));        // Assert a timeout of 7 seconds for a      // 6-second delay          System.out.println("Test3 Passed within the time");  }    // Helper method to simulate delays  private void delaySeconds(int seconds) throws InterruptedException {      TimeUnit.SECONDS.sleep(seconds);  } |

### Section 3: Demonstrating timeout failures

#### Test 4: Using Assertions.assertTimeout to handle timeout failure.

Expects a timeout failure since it asserts a 4-second timeout but delays for 5 seconds using Thread.sleep.

* Java

|  |
| --- |
| // Test 4: Using Assertions.assertTimeout to handle timeout failure  @Test  public void test4() {      Assertions.assertTimeout(Duration.ofSeconds(4), () -> {          try {              Thread.sleep(5000); // Delay of 5 seconds, expected to fail          } catch (InterruptedException e) {              // Handle interruption if needed          }      });  } |

**Test 5: Same as Test 4, using TimeUnit.SECONDS.sleep**

Similar to Test 4 but utilizes TimeUnit.SECONDS.sleep for the delay. It should also fail.

* Java

|  |
| --- |
| //Test 5: Same as Test 4, using TimeUnit.SECONDS.sleep  @Test  public void test5() {      Assertions.assertTimeout(Duration.ofSeconds(4), () -> {          try {              TimeUnit.SECONDS.sleep(5); // Delay of 5 seconds, expected to fail          } catch (InterruptedException e) {              // Handle interruption if needed          }      });  } |

**Test 6: Using Assertions.assertTimeout with a shorter timeout than delay**

Another expected timeout failure, asserting a 5-second timeout for a 6-second delay using a helper function.

* Java

|  |
| --- |
| // Test 6: Using Assertions.assertTimeout with a shorter  // timeout than delay  @Test public void test6()  {      // Assert a timeout of 5 seconds for a      // 6-second delay      Assertions.assertTimeout(Duration.ofSeconds(5),() -> delaySeconds1(6));  }    // Helper method for Test 6  private void delaySeconds1(int seconds) throws InterruptedException  {      TimeUnit.SECONDS.sleep(seconds);  } |

#### Final timeOutText.java will be :

* Java

|  |
| --- |
| package timeoutTest;    // This package contains tests for timeout functionality.  import java.time.Duration; // Import for handling time durations.  import java.util.concurrent.TimeUnit; // Import for time-related utility classes.  import org.junit.jupiter.api.Assertions; // Import for JUnit assertions.  import org.junit.jupiter.api.Test; // Import for JUnit test annotations.  import org.junit.jupiter.api.Timeout; // Import for setting test timeouts.    // This class demonstrates different ways to apply timeouts in JUnit tests.  class TimeOutTest {    // Section 1: Using Timeout() annotation        // Test 1: Demonstrates using Timeout annotation with Thread.sleep      @Test      @Timeout(6) // Set timeout to 6 seconds      public void test1() throws InterruptedException {          Thread.sleep(5000); // Simulate a 5-second delay          System.out.println("Test1 Passed within the time");      }        // Test 2: Demonstrates using Timeout annotation with TimeUnit.SECONDS.sleep      @Test      @Timeout(6) // Set timeout to 6 seconds      public void test2() throws InterruptedException {          TimeUnit.SECONDS.sleep(5); // Another way to simulate a 5-second delay          System.out.println("Test2 Passed within the time");      }    // Section 2: Using Assertions        // Test 3: Demonstrates using Assertions.assertTimeout      @Test      public void test3() {          Assertions.assertTimeout(Duration.ofSeconds(7), () -> delaySeconds(6)); // Assert a timeout of 7 seconds for a                                                                                  // 6-second delay          System.out.println("Test3 Passed within the time");      }        // Helper method to simulate delays      private void delaySeconds(int seconds) throws InterruptedException {          TimeUnit.SECONDS.sleep(seconds);      }    // Section 3: Demonstrating timeout failures        //Test 4: Using Assertions.assertTimeout to handle timeout failure      @Test      public void test4() {          Assertions.assertTimeout(Duration.ofSeconds(4), () -> {              try {                  Thread.sleep(5000); // Delay of 5 seconds, expected to fail              } catch (InterruptedException e) {                  // Handle interruption if needed              }          });      }        //Test 5: Same as Test 4, using TimeUnit.SECONDS.sleep      @Test      public void test5() {          Assertions.assertTimeout(Duration.ofSeconds(4), () -> {              try {                  TimeUnit.SECONDS.sleep(5); // Delay of 5 seconds, expected to fail              } catch (InterruptedException e) {                  // Handle interruption if needed              }          });      }        // Test 6: Using Assertions.assertTimeout with a shorter timeout than delay      @Test      public void test6() {          Assertions.assertTimeout(Duration.ofSeconds(5), () -> delaySeconds1(6));            // Assert a timeout of 5 seconds for a 6-second delay      }        // Helper method for Test 6      private void delaySeconds1(int seconds) throws InterruptedException {          TimeUnit.SECONDS.sleep(seconds);      }  } |

Mockito Framework

Mockito is a mocking framework. It is a Java-based library used to create simple and basic test APIs for performing unit testing of Java applications. It can also be used with other frameworks such as **JUnit** and **TestNG**.

What is Mocking?

Mocking is a process of developing the objects that act as the **mock** or **clone** of the real objects. In other words, mocking is a testing technique where mock objects are used instead of real objects for testing purposes. Mock objects provide a specific (dummy) output for a particular (dummy) input passed to it.

The mocking technique is not only used in Java but also used in any object-oriented programming language. There are many frameworks available in Java for mocking, but Mockito is the most popular framework among them.

To mock objects, you need to understand the three key concepts of mocking, i.e., stub, fake, and mock. Some of the unit tests involve only stubs, whereas some involve fake and mocks.

The brief description of the mocking concepts is given below:

1. **Stub:** Stub objects hold predefined data and provide it to answer the calls during testing. They are referred to as a dummy object with a minimum number of methods required for a test. It also provides methods to verify other methods used to access the internal state of a stub, when necessary. Stub object is generally used for **state verification**.
2. **Fake:** Fake are the objects that contain working implementations but are different from the production one. Mostly it takes shortcuts and also contains the simplified version of the production code.
3. **Mock:** Mock objects act as a dummy or clone of the real object in testing. They are generally created by an open-source library or a mocking framework like Mockito, EasyMock, etc. Mock objects are typically used for **behavior verification**.

Need for mocking

Before using the Mocking technique, we should know the reasons for using mocking, which are as follows:

* If we want to test a component that depends on the other component, but it is under development. It generally uses when working in a team and parts are divided between several team-mates. In this case, mocking plays an essential role in the testing of that component. Without mocking, we need to wait for the completion of the required elements for testing.
* If the real components perform slow operations while dealing with database connections or another complex read/ write operation. Sometimes the database queries can take 10, 20, or more seconds to execute. In such cases, we require mock objects to perform testing, and it can be done via mocking.
* If there is an infrastructure concern that makes the testing impossible. It is very similar to the first case. For example, when we create a connection to the database, some issues related to configurations occur. It requires mocking for creating mock components to provide unit testing.

What is Mockito?

Mockito is a Java-based mocking framework used for unit testing of Java application. Mockito plays a crucial role in developing testable applications. Mockito was released as an open-source testing framework under the**MIT (Massachusetts Institute of Technology) License.** It internally uses the Java Reflection API to generate mock objects for a specific interface. Mock objects are referred to as the dummy or proxy objects used for actual implementations.

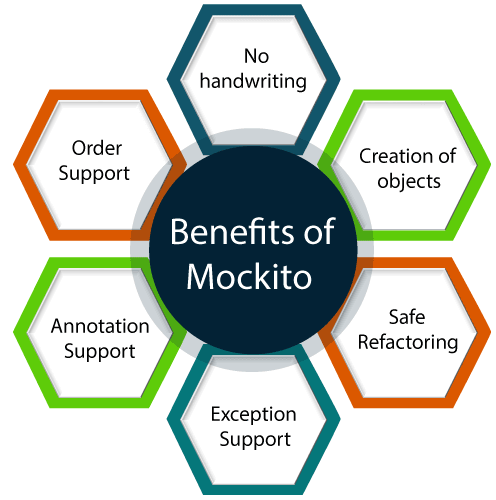
The main purpose of using the Mockito framework is to simplify the development of a test by mocking external dependencies and use them in the test code. As a result, it provides a simpler test code that is easier to read, understand, and modify. We can also use Mockito with other testing frameworks like **JUnit** and **TestNG**.

The Mockito framework was developed by upgrading the syntax and functionalities of

EasyMock framework. It was developed by a team of developers consisting of **Szczepan Faber, Brice Dutheil, Rafael Winterhalter, Tim van der Lippe,** and others. The stable or latest version of Mockito is **version 3.0.6** was released in August 2019.

Benefits of Mockito

Below are given some benefits of the Mockito framework:



* **No handwriting:** In Mockito, there is no requirement for writing your mock objects.
* **No handwriting:** In Mockito, there is no requirement for writing your mock objects.
* **Safe refactoring:** While renaming the method name of an interface or interchanging the parameters do not change the test code, as mock objects are created at runtime.
* **Exception support:** It supports the exception. In Mockito, the stack trace is used to find the cause of the exception.
* **Annotation support:** It creates mock objects using annotations like @Mock.
* **Order support:** It provides a check on the order of the method calls

Methods of Mockito

The Mockito framework provides a variety of methods such as mock(), verify(), when(), etc., used to test Java applications. Using these predefined methods makes testing very easy.

Mockito mock() method

It is used to create mock objects of a given class or interface. Mockito contains five **mock()** methods with different arguments. When we didn't assign anything to mocks, they will return default values. All five methods perform the same function of mocking the objects.

Following are the mock() methods with different parameters:

* **mock() method with Class:** It is used to create mock objects of a concrete class or an interface. It takes a class or an interface name as a parameter.  
  **Syntax:** <T> mock(Class<T> classToMock)
* **mock() method with Answer:** It is used to create mock objects of a class or interface with a specific procedure. It is an advanced mock method, which can be used when working with legacy systems. It takes Answer as a parameter along with the class or interface name. The Answer is an enumeration of pre-configured mock answers.  
  **Syntax:** <T> mock(Class<T> classToMock, Answer defaultAnswer)
* **mock() method with MockSettings:** It is used to create mock objects with some non-standard settings. It takes MockSettings as an additional setting parameter along with the class or interface name. MockSettings allows the creation of mock objects with additional settings.  
  **Syntax:** <T> mock(Class<T> classToMock, MockSettings mockSettings)
* **mock() method with ReturnValues:** It allows the creation of mock objects of a given class or interface. Now, it is deprecated, as ReturnValues are replaced with Answer.  
  **Syntax:** <T> mock(Class<T> classToMock, ReturnValues returnValues)
* **mock() method with String:** It is used to create mock objects by specifying the mock names. In debugging, naming mock objects can be helpful whereas, it is a bad choice using with large and complex code.  
  **Syntax:** <T> mock(Class<T> classToMock, String name)

Following code snippet shows how to use **mock()** method:

1. ToDoService doService = mock(ToDoService.**class**);

Mockito when() method

It enables stubbing methods. It should be used when we want to mock to return specific values when particular methods are called. In simple terms, "**When** the XYZ() method is called, **then** return ABC." It is mostly used when there is some condition to execute.

**Syntax: <T> when(T methodCall)**

Following code snippet shows how to use when() method:

1. when(mock.someCode ()).thenReturn(5);

In the above code, **thenReturn()** is mostly used with the **when()** method.

Mockito verify() method

The **verify()** method is used to check whether some specified methods are called or not. In simple terms, it validates the certain behavior that happened once in a test. It is used at the bottom of the testing code to assure that the defined methods are called.

Mockito framework keeps track of all the method calls with their parameters for mocking objects. After mocking, we can verify that the defined conditions are met or not by using the verify() method. This type of testing is sometimes known as **behavioral testing.** It checks that a method is called with the right parameters instead of checking the result of a method call.

The verify() method is also used to test the number of invocations. So we can test the exact number of invocations by using the **times method, at least once method,** and **at most method** for a mocked method.

There are two types of verify() methods available in the Mockito class, which are given below:

* **verify() method:** It verifies certain behavior happened once.  
  **Syntax:** <T> verify(T mock)
* **verify() method with VerificationMode:** It verifies some behavior happened at least once, exact number of times, or never.  
  **Syntax:** <T> verify(T mock, VerificationMode mode)

Mockito spy() method

Mockito provides a method to partially mock an object, which is known as the **spy** method. When using the spy method, there exists a real object, and spies or stubs are created of that real object. If we don't stub a method using spy, it will call the real method behavior. The main function of the spy() method is that it overrides the specific methods of the real object. One of the functions of the spy() method is it verifies the invocation of a certain method.

There are two types of spy() methods available in the Mockito class:

* **spy() method:** It creates a spy of the real object. The spy method calls the real methods unless they are stubbed. We should use the real spies carefully and occasionally, for example, when dealing with the legacy code.  
  **Syntax:** <T> spy(T object)
* **spy() method with Class:** It creates a spy object based on class instead of an object. The spy(T object) method is particularly useful for spying abstract classes because they cannot be instantiated.  
  **Syntax:** <T> spy(Class<T> classToSpy)

Following code snippet shows how to use the spy() method:

1. List spyArrayList = spy(ArrayList.**class**);

# Examples of Mockito and JUnit in Eclipse IDE

**Step 1:** Add Maven dependencies required by the application. The dependencies are always added in **pom.xml** file of the application. In this example you need to add the two dependencies:

JUnit dependency

Mockito dependency

<dependency>

    <groupId>junit</groupId>

    <artifactId>junit</artifactId>

    <version>4.13-rc-1</version>

    <scope>test</scope>

</dependency>

<dependency>

    <groupId>org.mockito</groupId>

    <artifactId>mockito-all</artifactId>

    <version>2.0.1-beta</version>

    <scope>test</scope>

</dependency>

**Step 1:** Create an interface named **ToDoService** that contains an unimplemented method.

**ToDoService.java**

**import** java.util.List;

**public** **interface** ToDoService {

**public** List<String> getTodos(String user);

 }

**Step 2:** Now, create an implementation class named **ToDoBusiness** for **ToDoService** interface.

**ToDoBusiness.java**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ToDoBusiness {

**public** ToDoService doService;

**public** ToDoBusiness(ToDoService doService) {

**this**.doService = doService;

    }

**public** List<String> getTodosforHibernate(String user) {

        List<String> hibernatelist = **new** ArrayList<String>();

        List<String> Combinedlist = doService.getTodos(user);

**for**(String todo: Combinedlist) {

**if**(todo.contains("Hibernate")) {

                hibernatelist.add(todo);

            }

        }

**return** hibernatelist;

        }

 }

**Step 3:** Create a JUnit test case named **ToDoBusinessMock** for unit testing.

**ToDoBusinessMock.java**

**import** **static** org.junit.Assert.assertEquals;

**import** **static** org.mockito.Mockito.mock;

**import** **static** org.mockito.Mockito.when;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.Test;

**public** **class** ToDoBusinessMock {

    @Test

**public** **void** testusing\_Mocks() {

        ToDoService doService = mock(ToDoService.**class**);

        List<String> combinedlist = Arrays.asList(" Use Core Java ", " Use Spring Core ", " Use w3eHibernate ", " Use Spring MVC ");

        when(doService.getTodos("dummy")).thenReturn(combinedlist);

        ToDoBusiness business = **new** ToDoBusiness(doService);

        List<String> alltd = business.getTodosforHibernate("dummy");

        System.out.println(alltd);

        assertEquals(1, alltd.size());

    }  }

Example of mocking a List class

Here, we are going to create an example of mocking a List class (**java.util.List**).

**Step 1:** Create a mock test class named **TestList** for testing the List class.

**TestList.java**

**import** **static** org.junit.Assert.\*;

**import** **static** org.mockito.Mockito.when;

**import** **static** org.mockito.Mockito.*mock*;

**import** java.util.List;

**import** org.junit.Test;

**import** org.mockito.Mock;

**public** **class** TestList {

    @Test

**public** **void** testList\_ReturnsSingle\_value() {

        List mocklist = mock(List.**class**);

                           when(mocklist.size()).thenReturn(1);

        assertEquals(1, mocklist.size());

        assertEquals(1, mocklist.size());

                          System.out.println( mocklist.size());

        System.out.println(mocklist);

    }

 }

Example of multiple return values of a List

Here, we are going to create an example of mocking a List class (**java.util.List**) with multiple return values. In the previous example, the list returns only one object (as it contains only one), whereas, in the following example, it returns multiple values (as it contains three items).

**Step 1:** Create a JUnit test case named **TestList** for testing the List class.

**TestList.java**

**import** **static** org.junit.Assert.\*;

**import** **static** org.mockito.Mockito.when;

**import** **static** org.mockito.Mockito.mock;

**import** java.util.List;

**import** org.junit.Test;

**public** **class** TestList {

      @Test

**public** **void** testList\_Returns\_MultipleValues() {

      List mocklist = mock(List.**class**);

      when(mocklist.size()).thenReturn(1).thenReturn(2).thenReturn(3);

      assertEquals(1, mocklist.size());

      assertEquals(2, mocklist.size());

      assertEquals(3, mocklist.size());

      System.out.println(mocklist.size());

      System.out.println(mocklist);

      }

 }

Example of mocking List.get() method

In this example, we are going to mock a List.get() method. To use the **get()** method, we need to pass a value (number) in it, as shown in the example.

**Step 1:** Create a test class named **TestList** for mocking the List.get() method.

**TestList.java**

**import** **static** org.junit.Assert.\*;

**import** **static** org.mockito.Mockito.mock;

**import** **static** org.mockito.Mockito.when;

**import** java.util.List;

**import** org.junit.Test;

**public** **class** TestList {

        @Test

**public** **void** testList\_get() {

      List mocklist = mock(List.**class**);

      when(mocklist.get(0)).thenReturn("Mockito");

      assertEquals("Mockito", mocklist.get(0));

      System.out.println(mocklist.get(0));

      }

 }

What is Test-Driven Development (TDD)?

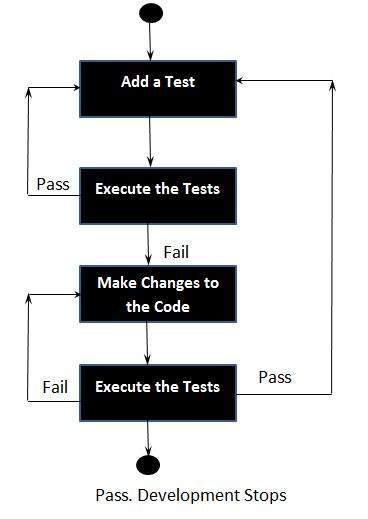
Test-driven development starts with developing test for each one of the features. The test might fail as the tests are developed even before the development. Development team then develops and refactors the code to pass the test.

Test-driven development is related to the test-first programming evolved as part of extreme programming concepts.

Test-Driven Development Process:

* Add a Test
* Run all tests and see if the new one fails
* Write some code
* Run tests and Refactor code
* Repeat

Example:



Context of Testing:

* Valid inputs
* Invalid inputs
* Errors, exceptions, and events
* Boundary conditions
* Everything that might break

Benefits of TDD:

* Much less debug time
* Code proven to meet requirements
* Tests become Safety Net
* Near zero defects
* Shorter development cycles

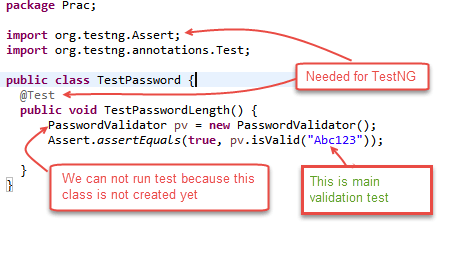
**Example of TDD**

Here in this example, we will define a class password. For this class, we will try to satisfy following conditions.

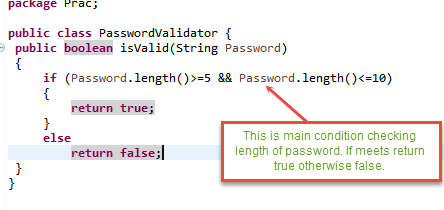
A condition for Password acceptance:

* The password should be between 5 to 10 characters.

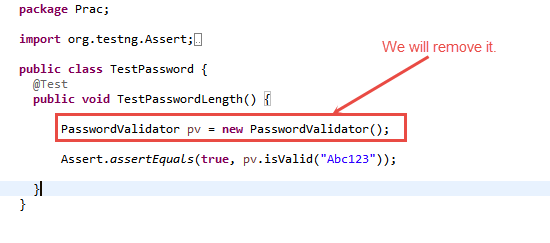
First, we write the code that fulfills all the above requirements.

[](https://www.guru99.com/images/8-2016/081216_0811_TestDrivenD5.png)

**Scenario 1**: To run the test, we create class PasswordValidator ();

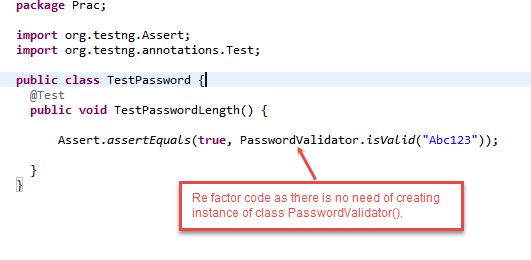
[](https://www.guru99.com/images/8-2016/081216_0811_TestDrivenD6.png)

**Scenario 2**: Here we can see in method TestPasswordLength () there is no need of creating an instance of class PasswordValidator. Instance means creating an [object](https://www.guru99.com/java-oops-class-objects.html) of class to refer the members (variables/methods) of that class.

[](https://www.guru99.com/images/8-2016/081216_0811_TestDrivenD8.png)

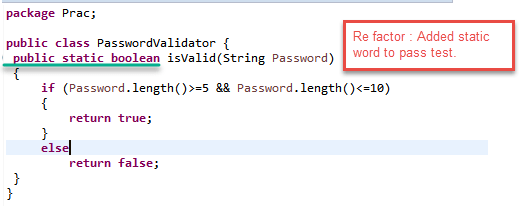
We will remove class PasswordValidator pv = new PasswordValidator () from the code. We can call the **isValid ()** method directly by **PasswordValidator. IsValid ("Abc123")**. (See image below)

So we Refactor (change code) as below:

[](https://www.guru99.com/images/8-2016/081216_0811_TestDrivenD9.png)

**Scenario 3**: After refactoring the output shows failed status (see image below) this is because we have removed the instance. So there is no reference to [non –static](https://www.guru99.com/java-static-variable-methods.html) method **isValid ().**

So we need to change this method by adding "static" word before Boolean as public static boolean isValid (String password). Refactoring Class PasswordValidator () to remove above error to pass the test.

[](https://www.guru99.com/images/8-2016/081216_0811_TestDrivenD11.png)

**Output:**

After making changes to class PassValidator () if we run the test then the output will be PASSED