**JUNIT & MOKITO**

**JUnit Tutorial**

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

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

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

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

## **Types of unit testing**

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

#### **1) Manual Testing**

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

#### **2) Automated Testing**

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

#### **Annotations for Junit testing**

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

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

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

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

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

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

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

## **Assert class**

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

#### Methods of Assert class

The common methods of Assert class are as follows:

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

## **Simple JUnit example in eclipse IDE**

#### **Write the program logic**

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

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

#### **Write the test case**

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

**1.package** com.javatpoint.testcase;

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

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:

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

ndividual function or procedure (program). The developers usually perform it during testing.

## **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.

39.3M

649

Hello Java Program for Beginners

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**.

## **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.

## **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)**

## **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

# Argument Matchers

Argument matchers are mainly used for performing flexible verification and stubbing in Mockito. It extends **ArgumentMatchers** class to access all the matcher functions. Mockito uses **equal()** as a legacy method for verification and matching of argument values. In some cases, we need more flexibility during the verification of argument values, so we should use argument matchers instead of **equal()** method. The ArgumentMatchers class is available in **org.mockito** package.

# Hamcrest Matchers

**Hamcrest** is a popular framework that help us to create the matcher objects. It is used for writing software tests and also performs unit testing in Java programming language. Hamcrest is mainly used with other unit testing frameworks like **JUnit, jMockit, Mockito,** etc.

Hamcrest framework is designed to make the test more readable and understandable. It makes the use of static methods to construct an assertion that is very easy to write and understand. It has been ported to C#, Python, PHP, JavaScript, C++, Rust, and Swift.

The Hamcrest framework was designed to accommodate different types of unit testing frameworks. For example, Hamcrest can be used with TestNG and JUnit (all versions). The Hamcrest framework is also used with mocking frameworks such as JMock, EasyMock, and Mockito.

Mockito Annotations

The Mockito framework provides a variety of annotations to make the code simple and easy to understand. Also, it reduces the lines of code that helps in focusing on the business logic. In Mockito, annotations are useful when we want to use the mocked object at different places to avoid calling the same methods multiple times.

The Mockito annotations are given below:

**@Mock:** It is used to mock the objects that helps in minimizing the repetitive mock objects. It makes the test code and verification error easier to read as parameter names (field names) are used to identify the mocks. The @Mock annotation is available in the **org.mockito** package.  
Following code snippet shows how to use the @mock annotation:

**@RunWith:** It is a class-level annotation. It is used to keep the test clean and improves debugging. It also detects the unused stubs available in the test and initialize mocks annotated with @Mock annotation. The @RunWith annotation is available in the **org.mockito.junit** package.  
Following code snippet shows how to use the @RunWith annotation:

**@InjectMocks:** It marks a field or parameter on which the injection should be performed. It allows shorthand mock and spy injections and minimizes the repetitive mocks and spy injection. In Mockito, the mocks are injected either by setter injection, constructor injection, and property injection. The @InjectMocks annotation is available in the **org.mockito** package.

**@Captor:** It allows the creation of a field-level argument captor. It is used with the Mockito's verify() method to get the values passed when a method is called. Like other annotations, @Captor annotation is also available in the **org.mockito** package.

**@Spy -** It allows the creation of partially mock objects. In other words, it allows shorthand wrapping of the field instances in a spy object. Like other annotations, @Spy annotation is also available in the **org.mockito** package.

## **JUnit Rules**

In the above examples, we have used the JUnit **runner (MockitoJUnitRunner)**. It makes the test dependent on that particular runner.

We cannot use multiple runners in the same test. To overcome this problem, we should follow **JUnit rules** that makes the test more flexible. It allows us to use multiple rules in the same test.

A JUnit rule is defined as a component that is used to obstruct the test method calls and allows us to perform something before and after the test method is invoked. The JUnit provides the following rules to:

* Create directories/files that are deleted after a test method has been run.
* Fail a test, if the described timeout has exceeded before a test method is invoked.
* Establish an external resource like a socket or a database connection before a test method is invoked.
* Free the configured external resource after a test method is invoked.

To use the JUnit rules, we need to add the **@Rule** annotation in the test.

**@Rule:** It annotates the fields. It refer to the rules or methods that returns a rule. The annotated fields must be public, non-static, and subtypes of the **TestRule** or **MethodRule.**