Mockito

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 Unit testing?

Unit testing is a software testing technique in which individual components/parts of the software is tested, i.e., a group of computer programs, usage procedure, etc. Unit testing of an object is done during the development of an application or project.

The aim of unit testing is to isolate a segment of code (unit) and verifies its correctness.

A unit is referred to as an individual 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.

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.

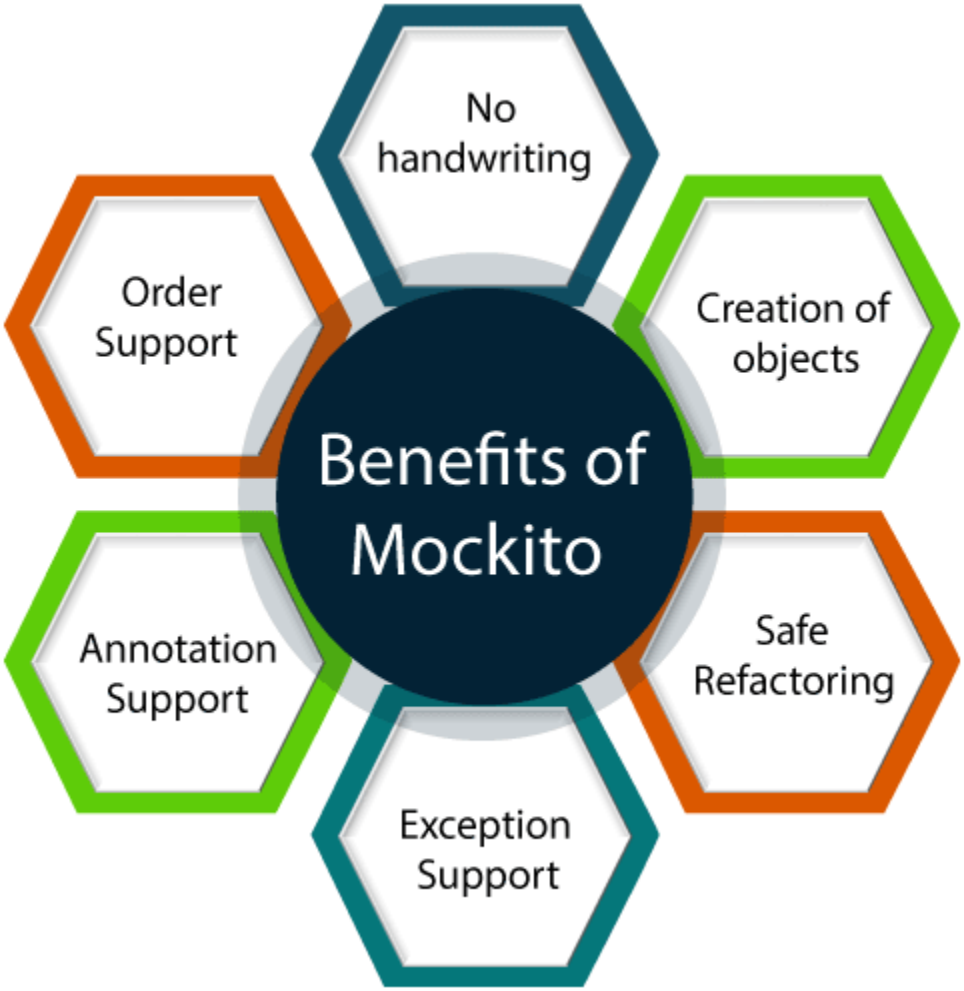
The main purpose of using the Mockito framework is to simplify the development of a test by mocking external dependencies and use them in the test code. As a result, it provides a simpler test code that is easier to read, understand, and modify. We can also use Mockito with other testing frameworks like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

The brief description of the Mockito methods are given below:

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)

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)**

Behavior-driven development (BDD)

Behavior-driven development is an Agile software development process that supports collaboration among the developers, quality analysts, and business members in a software project. It is developed from the **Test-driven development (TDD)** software.

The BDD is a combination of general techniques and principles of the TDD with the ideas originated from the Domain-driven design (DDD) and the object-oriented analysis and design (OOAD) approach.

Mockito uses the **BDDMockito** class that is available in the **org.mockito** package. It develops a test in BDD style. The BDD style of writing texts uses the **//given //when //then** comments as the primary part of the test body. It uses ***given(..)willReturn(..)*** method in place of **when(..)thenReturn(..)** method.

Example of BDD style

Here, we are going to create an example of a BDD style test. Switching to BDD style makes a minor difference only in the test syntax. It splits the test syntax into three parts: **given, when,** and **then** that makes the code more readable.

- **Given:** We can use the setup part and the given kind of syntax.
- **When:** We can do the actual invocations of the test.
- **Then:** We can use the readable **asserts** like **assertThat()** and can also check whether the post-conditions are satisfied or not.

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:

```
1. @Mock
2. ToDoService servicemock;
```

Note: The @Mock annotation is always used with the @RunWith 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:

```
1. @RunWith(MockitoJUnitRunner.class)
2. public class ToDoBusinessMock {
3. .....
4. }
```

In the above code snippet, the **MockitoJUnitRunner** class is used to check that all the mocks are created and autowired when needed.

- **@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. Following code snippet shows how to use the @InjectMocks annotation:

```
1. @InjectMocks
2. ToDoBusiness business;
```

- **@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.

- Following code snippet shows how to use the @Captor annotation:

```
1. @Captor
2. ArgumentCaptor<String> argumentCaptor;
```

- **@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. Following code snippet shows how to use the @Spy annotation:

```
1. @Spy
2. ArrayList<String> arraylistSpy;
```

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.**

```
1. @Rule
2. public MockitoRule mockitorule = MockitoJUnit.rule();
```

In the above code snippet, we have used the **MockitoRule** class. You can use any JUnit rule as per your requirement.