UNIT TESTING TOOLS
JUNIT & MOCKITO

Team Crystals

Team Members

Anuja Ajay
Aysha Fathima
Nimisha R S
Sankari Suresh

JUnit and Mockito

JUnit and Mockito are two of the most popular open-source testing frameworks for Java applications. These frameworks provide developers with the tools to write and execute automated tests for their applications, ensuring that the code is robust and free from errors. JUnit provides the framework for writing and executing unit tests, while Mockito provides the framework for creating and managing mock objects.

Unit testing is an essential part of software development. It involves the creation of test cases that validate the behavior of individual units or components of the code. Unit tests are typically written by developers and are run frequently during the development process to ensure that code changes do not break existing functionality.

Mocking is a technique used in unit testing to simulate the behavior of objects that the code under test depends on. By creating mock objects, developers can isolate the code under test from its dependencies and test it in isolation. This technique helps to ensure that the tests are focused on the behavior of the code under test and not on the behavior of its dependencies.

JUnit

Java provides a framework called JUnit to perform the unit testing of our Java code. In the development of **test-driven** development, JUnit is especially important. The JUnit is one of the frameworks available in the unit testing frameworks. The **xUnit** is the unit testing framework family, and JUnit is the part of the **xUnit**.

JUnit promotes the idea of "first testing then coding", which emphasizes setting test data for a piece of code that can be tested first and then implemented. Junit increases the stability of the code. It also increases the productivity of the programmer.

These are the following features of JUnit:

1. An open-source framework used to write and run tests.
2. For testing the expected result, the JUnit provides assertions.
3. To identify the test methods, it provides annotation.
4. We can write the code faster for increasing quality using JUnit.
5. For running tests, it provides test runners.
6. It is quite simple, not so complex and requires less time.

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 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.

The common methods of Assert class are as follows:

7. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
8. **void assertTrue(boolean condition)**: checks that a condition is true.
9. **void assertFalse(boolean condition)**: checks that a condition is false.
10. **void assertNull(Object obj)**: checks that object is null.
11. **void assertNotNull(Object obj)**: checks that object is not null.

JUnit follows a simple and straightforward workflow for unit testing. Here are the basic steps involved in using JUnit:

Define a test case: A test case is a set of conditions or inputs that are used to test a specific behavior or function of a unit of code.

Write test methods: A test method is a method that performs a specific test case. It typically includes assertions that compare the actual output of the unit of code with the expected output.

Run the tests: JUnit provides a test runner that executes the test methods and reports the results.

Here's an example of how to write a JUnit test case:

java code

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

import org.junit.Test;

public class MyTest {
```

```
@Test
public void testAddition() {
    int result = Calculator.add(2, 3);
    assertEquals(5, result);
} }
```

In this example, the @Test annotation is used to mark the testAddition() method as a test method. The assertEquals() method is used to compare the actual output of the Calculator.add() method with the expected output.

Mockito

Mockito is a Java-based library or mocking framework that is used to perform unit testing of Java applications. Mockito allows us to add mock data or dummy functionality to the mock interface to perform unit testing.

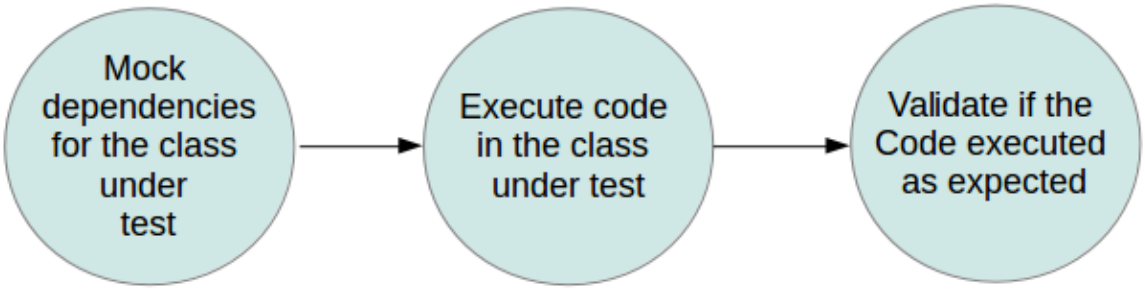
To create a dummy object for a given interface, Mockito uses Java reflection. The mock objects are the proxy of the actual implementations. Testing the functionality of a class without requiring a database connection is referred to as **Mocking**. For performing the Mocking of the real service, mock objects are used.

These are the following benefits of using the Mockito for testing:

1. There is no need to write the dummy data on your own.
2. It supports the return values.
3. It supports annotation for creating mocks.
4. It supports exceptions.

5. Changing the interface name or re-ordering the parameters does not affect the test code because mocks are created at runtime.

   Mockito is a popular open source framework for mocking objects in software test. Using Mockito greatly simplifies the development of tests for classes with external dependencies.
   A *mock object* is a dummy implementation for an interface or a class. It allows to define the output of certain method calls. They typically record the interaction with the system and tests can validate that.
   This allows you to simplify the test setup.



Recent versions of Mockito can also mock static methods and final classes. Also private methods are not visible for tests, they can also not be mocked.

Mockito records the interaction with mock and allows you to check if the mock object was used correct, e.g. if a certain method has been called on the mock. This allows you to implement behaviour testing instead of only testing the result of method calls.

Using the Mockito libraries should be done with a modern dependency system like Maven or Gradle. All modern IDEs (Eclipse, Visual Studio Code, IntelliJ) support both Maven and Gradle.

Mockito provides several methods to create mock objects:

- Using the `@ExtendWith(MockitoExtension.class)` extension for JUnit 5 in combination with the `@Mock` annotation on fields
- Using the static `mock()` method.
- Using the `@Mock` annotation.

If you use the `@Mock` annotation, you must trigger the initialization of the annotated fields. The `MockitoExtension` does this by calling the static method `MockitoAnnotations.initMocks(this)`.

Here are the basic steps involved in using Mockito:

Create a mock object: Use the mock() method of the Mockito class to create a mock object that simulates the behavior of a real object.

Define the behavior of the mock object: Use the when() method of the Mockito class to specify the behavior of the mock object when a specific method is called.

Use the mock object in your test: Use the mock object in your test code to simulate the behavior of the real object.

Here's an example of how to use Mockito to create a mock object:

java code

```
import static org.mockito.Mockito.*;
List<String> mockList = mock(List.class);
when(mockList.get(0)).thenReturn("first");
when(mockList.get(1)).thenThrow(new RuntimeException());
String result = mockList.get(0);
assertEquals("first", result);
try {  mockList.get(1);
fail("Expected exception");
```

```
}
catch (RuntimeException e) {  // Exception expected }
```

In this example, the mock() method of the Mockito class is used to create a mock object of the List class. The when() method is used to specify the behavior of the mock object when the get() method is called with specific parameters. The thenReturn() method is used to specify the return value of the get() method when it is called with the parameter 0. The thenThrow() method is used to specify that an exception should be thrown when the get() method is called with the parameter 1. Finally, the mock object is used in test code to simulate the behavior of the real object.

Mockito Mocking Framework

Mockito provides developers with the ability to create mock objects that simulate the behavior of real objects. Mock objects are useful in unit testing because they allow developers to test code in isolation, without having to rely on the behavior of other components in the system.

To create a mock object with Mockito, you first need to create a mock instance of the class that you want to mock. Here's an example of how to create a mock object for the Calculator class:

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

Calculator calculatorMock = mock(Calculator.class);
```

This code creates a mock object for the Calculator class using the mock() method from the Mockito library. Once you have a mock object, you can use it in your unit tests to simulate the behavior of the real object.

Mockito Argument Matchers

Argument matchers are used in Mockito to match arguments passed to a method call on a mock object. Argument matchers are useful when you don't know the exact value that will be passed to a method call or when the value is not important to the test.

Here's an example of how to use argument matchers in Mockito:

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

 Calculator calculatorMock = mock(Calculator.class);

when(calculatorMock.add(anyInt(), anyInt())).thenReturn(4);

int result = calculatorMock.add(2, 2);

 assertEquals(4, result);
```

In this example, the add() method on the calculatorMock object is set to return the value 4 whenever it is called with any two integer arguments. The anyInt() argument matcher is used to match any integer value passed to the method call.

Mockito Verification

Mockito provides developers with the ability to verify that certain behavior has occurred on a mock object. Verification is useful in unit testing because it allows developers to ensure that their code is interacting with dependencies in the correct way.

Here's an example of how to use verification in Mockito:

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

```
Calculator calculatorMock = mock(Calculator.class);

calculatorMock.add(2, 2);

verify(calculatorMock).add(2, 2);
```

In this example, the add() method on the calculatorMock object is called with the values 2 and 2. The verify() method is then used to verify that the add() method was called with those arguments.

Mockito Exceptions

Mockito provides developers with the ability to simulate exceptions thrown by a mock object. This is useful in unit testing because it allows developers to test how their code handles exceptions in different scenarios.

Here's an example of how to simulate an exception in Mockito:

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

Calculator calculatorMock = mock(Calculator.class);

when(calculatorMock.add(anyInt(),anyInt())).thenThrow(new
RuntimeException());

 try {    calculatorMock.add(2, 2);

fail("Expected exception not thrown");

} catch (RuntimeException e) {    // Exception thrown, test passes }
```

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)

  Syntax: <T> mock(Class<T> classToMock, String name)

  FX

```
    return new MockSettingsImpl().defaultAnswer(RETURNS_DEFAULT
    S);

}
```

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

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

Mockito when() method

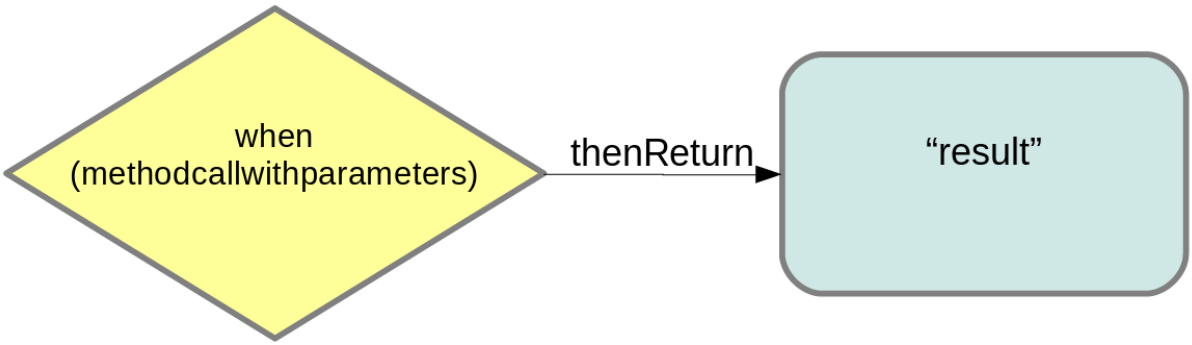
It enables stubbing methods. It should be used when we want to mock to return specific values when particular methods are called. In simple terms, "**When** the XYZ() method is called, **then** return ABC." It is mostly used when there is some condition to execute.

Syntax: <T> when(T methodCall)

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

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:

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

Mockito reset() method

The Mockito reset() method is used to reset the mocks. It is mainly used for working with the container injected mocks. Usually, the reset() method results in a lengthy code and poor tests. It's better to create new mocks rather than using reset() method. That is why the reset() method is rarely used in testing.

The signature of the reset() method is:

```
public static <T> void reset(T ... mocks) {
    MOCKITO_CORE.reset(mocks);
}
```

Mockito doAnswer() method

It is used when we want to stub a void method with a generic Answer type. The signature of the doAnswer() method is:

```
public static Stubber doAnswer(Answer answer) {
    return MOCKITO_CORE.doAnswer(answer);
    }
```

Mockito doNothing() method

It is used for setting void methods to do nothing. The doNothing() method is used in rare situations. By default, the void methods on mock instances do nothing, i.e., no task is performed.

The signature of doNothing() method is:

```
public static Stubber doNothing() {
    return MOCKITO_CORE.doAnswer(new DoesNothing());
    }
```

Mockito doReturn() method

It is used on those rare occasions when we cannot use Mockito.when(object). T he Mockito.when(object) method is always
suggested for stubbing because it is argument type-safe and more readable as compare to the doReturn() method.

The signature of doReturn() method is:

```
public static Stubber doReturn(Object toBeReturned) {
    return MOCKITO_CORE.doAnswer(new Returns(toBeReturned));
   }
```

Mockito inOrder() method

It is used to create objects that allow the verification of mocks in a specific order. Verification done in order is more flexible as we don't have to verify all interactions. We need to verify only those interactions that are interested in testing (in order). We can also use the inOrder() method to create an inOrder object passing mocks that are relevant for in-order verification.

The signature of Mockito.inOrder() method is:

```
public static InOrder inOrder(Object... mocks) {
     return MOCKITO_CORE.inOrder(mocks);
      }
```

Mockito verifyNoMoreInteractions() method

It is used to check that any of the given mocks have any unverified interactions. We can use this method after verifying all the mock, to make sure that nothing else was invoked on the mocks. It also detects the unverified invocations that occur before the test method, for example, in setup(), @Before method, or the constructor. It is an optional method, and we don't need to use it in every test.

The signature of the verifyNoMoreInteractions() method is:

```
 public static void verifyNoMoreInteractions(Object... mocks) {
    MOCKITO_CORE.verifyNoMoreInteractions(mocks);
```

}

Mockito doThrow() method

It is used when to stub a void method to throw an exception. It creates a new exception instance for each method invocation.
There are two types of doThrow() methods available in
the Mockito class with different parameters, as shown below:

doThrow() method with Throwable: This method is used when we want to stub a void method with an exception. **Syntax:** doThrow(Throwable toBeThrown)

The signature of the doThrow() method is:

```
public static Stubber doThrow(Throwable toBeThrown) {
    return MOCKITO_CORE.doAnswer(new ThrowsException(toBeThrown));
```

doThrow() method with Class: This method is used when we want to stub a void method to throw an exception of a specified class.
Syntax: doThrow(Class<? extends Throwable> toBeThrown)
The signature of doThrow() method is:

```
public static Stubber doThrow(Class<? extends Throwable> toBeThrown) {
  return MOCKITO_CORE.doAnswer(new ThrowsExceptionClass(toBeThrown));
 }
```

Mockito doCallRealMethod() method

It is used when we want to call the real implementation of a method. In other words, it is used to create partial mocks of an object.

It is used in rare situations, such as to call the real methods. It is similar to the spy() method, and the only difference is that it results in complex code.

The signature of the doCallRealMethod() method is:

```
public static Stubber doCallRealMethod() {
    return MOCKITO_CORE.doAnswer(new CallsRealMethods());
   }
```