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**.

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.

Benefits of Mockito

- **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.

Mock() methods

- **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.
  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. Mock Settings 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 answers.
- **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.

Mockito Annotations

@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.

@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.

@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.

@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.

@Spy - It allows the creation of partially mock objects. In other words, it allows shorthand wrapping of the field instances in a spy object.