JUnit | Testing Framework for Java

Junit provide the concepts of **unit testing in 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.

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 expectedava Program for Beginners

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 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.

JUnit Rules

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.of India (1947-2020)

**@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.**

# 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**.

## 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.

Basics of Mockito:

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**

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.

The ArgumentMatchers class contains a variety of methods; some of them are listed below:

|  |  |
| --- | --- |
| **Method type and method name** | **Description** |
| <T> any() | It matches all values (anything), including null values and varargs. |
| boolean anyBoolean() | It matches to any boolean or not-null boolean values. |
| byte anyByte() | It matches to any byte or not-null byte values. |
| char anyChar() | It matches to any char or not-null character values. |
| Collection <T> anyCollection | It matches any not-null collection in the application. |
| double anyDouble() | It matches to any double or not-null double values. |
| float anyFloat() | It matches to any float or not-null float values. |
| int anyInt() | It matches to any int or not-null integer values. |
| Iterable<T> anyIterable() | It matches to any int or not-null integer values. |
| Iterable<T> anyIterable() | It matches to any not-null iterable values. |
| List<T> anyList() | It matches any not-null list. |
| long anyLong() | It matches to any long or not-null long values. |
| Set<T> anySet() | It matches any not-null set. |
| short anyShort() | It matches to any short or not-null short values. |
| String anyString() | It matches any not-null String. |
| <T> argThat(ArgumentMatcher<T> matcher) | It allows the creation of custom argument matchers. |
| boolean booleanThat(ArgumentMatcher<Boolean> matcher) | It allows the creation of the custom boolean argument matchers. |
| byte byteThat(ArgumentMatcher<Byte> matcher) | It allows the creation of custom byte argument matchers. |
| char charThat(ArgumentMatcher<Character> matcher) | It allows the creation of custom char argument matchers. |
| String contains(String substring) | It matches the String argument that contains the substring. |
| double doubleThat(ArgumentMatcher<Double> matcher) | It allows the creation of custom double argument matchers. |
| String endsWith(String suffix) | It matches the String argument that ends with the given suffix. |
| boolean eq(boolean value) | It matches with the boolean argument that is equal to the given value. |
| double eq(double value) | It matches with the double argument that is equal to the given value. |
| long eq(long value) | It matches the long argument that is equal to the given value. |
| <T> isNotNull() | It matches with the not null argument. |
| <T> is Null() | It matches the null argument. |
| <T> same(T value) | It checks whether the object argument is the same as the given value. |

# 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.

# Mockito Annotations

**@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.

**@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.

**@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.

**@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.