**Name – Nabanita Sarkar**

**Emp ID – 815944**

**Junit Notes:**

JUnit is a unit testing framework for Java programming language. It plays a crucial role test-driven development, and is a family of unit testing frameworks collectively known as xUnit.

JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented. This approach is like "test a little, code a little, test a little, code a little." It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

**Features of JUnit**

* JUnit is an open source framework, which is used for writing and running tests.
* Provides annotations to identify test methods.
* Provides assertions for testing expected results.
* Provides test runners for running tests.
* JUnit tests allow you to write codes faster, which increases quality.
* JUnit is elegantly simple. It is less complex and takes less time.
* JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.

**Unit Test Case**

A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected. To achieve the desired results quickly, a test framework is required. JUnit is a perfect unit test framework for Java programming language.

A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.

**Unit Testing**

Unit testing involves the testing of each unit or an individual component of the software application. It is the first level of functional testing. The aim behind unit testing is to validate unit components with its performance. A unit is a single testable part of a software system and tested during the development phase of the application software. The purpose of unit testing is to test the correctness of isolated code. A unit component is an individual function or code of the application. White box testing approach used for unit testing and usually done by the developers. Whenever the application is ready and given to the Test engineer, he/she will start checking every component of the module or module of the application independently or one by one, and this process is known as Unit testing or components testing.

**Assert Methods**

* [assertArrayEquals()](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertArrayEquals)
* [assertEquals()](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertEquals)
* [assertTrue() + assertFalse()](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertTrue)
* [assertNull() + assertNotNull()](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertNull)
* [assertSame() and assertNotSame()](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertSame)
* [assertThat()](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertThat)

**assertArrayEquals()**

The assertArrayEquals() method will test whether two arrays are equal to each other. In other words, if the two arrays contain the same number of elements, and if all the elements in the array are equal to each other.

**assertEquals()**

The assertEquals() method compares two objects for equality, using their equals() method.

**assertTrue() + assertFalse()**

The assertTrue() and assertFalse() methods tests a single variable to see if its value is either true, or false.

**assertNull() + assertNotNull()**

The assertNull() and assertNotNull() methods test a single variable to see if it is null or not null.

**assertSame() and assertNotSame()**

The assertSame() and assertNotSame() methods tests if two object references point to the same object or not. It is not enough that the two objects pointed to are equals according to their equals() methods.

**assertThat()**

The assertThat() method compares an object to an org.hamcrest.Matcher to see if the given object matches whatever the Matcher requires it to match.

**Mockito**

Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.

Consider a case of Stock Service which returns the price details of a stock. During development, the actual stock service cannot be used to get real-time data. So we need a dummy implementation of the stock service. Mockito can do the same very easily, as its name suggests.

**Benefits of Mockito:**

* **No Handwriting** − No need to write mock objects on your own.
* **Refactoring Safe** − Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
* **Return value support** − Supports return values.
* **Exception support** − Supports exceptions.
* **Order check support** − Supports check on order of method calls.
* **Annotation support** − Supports creating mocks using annotation.

**BDD Mockito**

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.

**ArgumentCaptor**

The AgrumentCaptor is a class that is defined in the org.mockito package. It is used to capture argument values for further assertions. We use argument captor with the methods like verify() or then() to get the values passed when a specific method is invoked.

**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.
* **@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.
* **@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.