**Unit Testing**

Unit testing is a software testing technique where individual units or components of a software application are tested in isolation to ensure that they function correctly. These units can be functions, methods, classes, or modules. The purpose of unit testing is to validate that each unit of the software performs as designed.

**JUnit**

JUnit is a popular open-source unit testing framework for Java. It provides a simple and effective way to write and execute unit tests for Java applications. JUnit helps developers automate the testing process and provides various features and annotations to make writing tests easier and more efficient.

**Features of JUnit**

1. Annotations
2. Assertions
3. Test Runners:  
   essentially a class that defines the mechanism for executing test cases. It's the part of the framework that is responsible for running tests and generating the results. The test runner decides how the tests will be initialized, executed, and how feedback will be provided about the test outcomes (pass, fail, or skip).
4. Test Suites  
   The concept of test suites allows you to group multiple test classes together and run them collectively.

Dependency:  
 <dependency>

<groupId>org.junit.platform</groupId>

<artifactId>junit-platform-suite</artifactId>

<version>1.8.2</version>

<scope>test</scope>

</dependency>

@Suite, @SelectClasses, @SelectPackages

1. Parameterized Tests

Parameterized tests in JUnit are useful when you need to run the same test several times with different sets of parameters. This feature helps in covering a wide range of input values and conditions with a single test method, improving code reusability and making it easier to expand test coverage

### **Different Sources for Parameters**

@ParameterizedTest

@ValueSource: Allows simple literal values (e.g., strings, ints, longs).  
@EnumSource: Provides values from an enum class.  
@MethodSource: Refers to a method that returns a stream of arguments. It's very flexible and can be used for complex parameter structures.  
@CsvFileSource: Reads values from a CSV file.

1. Exception Testing:

assertThrows(e,a)

**Annotations**

@SpringBootTest

@Test

@TestMethodOrder

@DisplayName

@Beforeeach

@AfterEach

@BeforeAll

@AfterAll

@Disabled

**Assertion**

Assertions are statements in a programming language that evaluate whether a condition is true or false. They are commonly used in testing to verify that certain conditions hold true during the execution of a program. When an assertion fails (i.e., the condition evaluates to false), it indicates a potential bug or unexpected behavior in the code.

Different Methods of assertion

assertEquals(expected, actual)

assertTrue

assertAll

**Mockito**

It allows developers to simulate the behavior of objects or dependencies that the code under test interacts with, without needing to create real instances of those dependencies

**Ways to Create Mock Objects**

1. By Using @Mock Annotation
2. By Using mock() method of mock class

**Stubbing**

Stubbing in Mockito refers to the process of defining the behavior of a mock object when its methods are called during a test. When you create a mock object, it initially has default behaviors for all its methods, such as returning null for object types or zero for numeric types.

**Spy**

spy is a type of mock object that wraps around a real object or instance of a class. Unlike regular mock objects, which are entirely stubbed and have no real behavior, spies retain the real behavior of the objects they wrap, while still allowing you to verify and stub specific methods as needed.

1. By Using the @Spy annotation
2. By using spy method of mockito class

**Argument Matchers**

Argument matchers are mainly used for performing flexible verification and stubbing in Mockito.

any(),anyInt(),anyByte() etc.

### 

### **Stubbing Methods**

* when(T methodCall): Allows stubbing of methods. Used in conjunction with methods like thenReturn(), thenThrow(), and thenAnswer() to define the behavior of the mock when the specified method is called.
* thenReturn(T value): Specifies the value to be returned when the stubbed method is called.
* thenThrow(Throwable... toBeThrown): Specifies one or more exceptions to be thrown when the stubbed method is called.
* thenAnswer(Answer<?> answer): Allows custom responses when the stubbed method is called, enabling more dynamic behavior.

### **Verification**

* verify(T mock): Used to verify that certain interactions or method calls were made on the mock object.
* verify(T mock, VerificationMode mode): Allows verification with a specific verification mode, e.g., to check if a method was called a certain number of times or never called.
* times(int wantedNumberOfInvocations): Verification mode that checks if a method was called a specific number of times.
* never(): Verification mode that checks if a method was never called.
* atLeast(int minNumberOfInvocations), atLeastOnce(), atMost(int maxNumberOfInvocations), only(): Other verification modes to check the number of invocations with more flexibility.

### **Argument Matchers**

* any(), anyInt(), anyString(), eq(T value), etc.: Argument matchers that can be used when stubbing or verifying method calls. They allow flexible argument matching, rather than exact matches.

### **. Capturing Arguments for Further Assertions**

* ArgumentCaptor<T>: A utility class to capture method arguments for further assertions. Useful when you want to perform additional tests on the parameters passed to a method call.

### **Exception Handling**

* doThrow(Throwable): An alternative approach to thenThrow() for stubbing void methods to throw an exception.

### **BDD-Style Syntax**

* given(T methodCall): An alias for when(T methodCall), used for a more Behavior-Driven Development (BDD) style.
* willReturn(T value), willThrow(Throwable...), willAnswer(Answer<?> answer): BDD-style equivalents for specifying stubbing behavior.