JUNIT with Spring:

- What is Unit Testing ?
- Why is Unit Testing ?
- What is Junit ?
- Junit Introduction
- Junit Architecture
- Unit Testing Examples
- Mocking
- Rest API Unit Testing
- Code Coverage using Jacocoo

1. What is Unit Testing ?

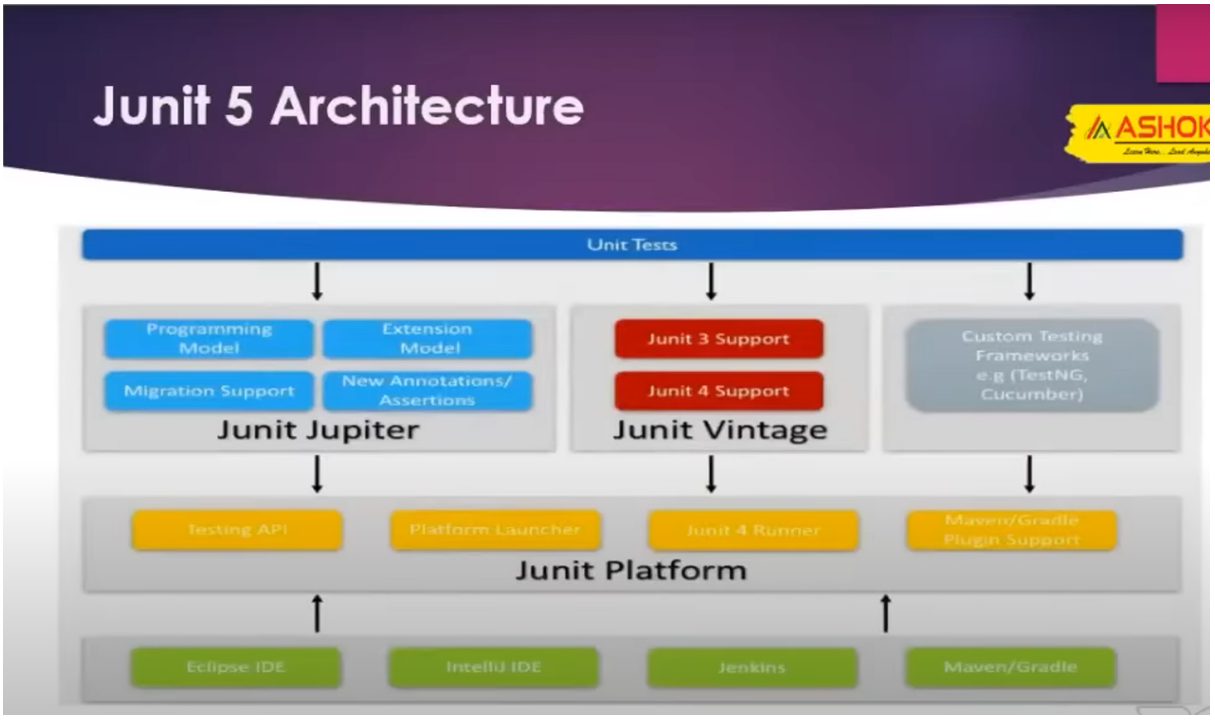
- Unit Testing is a type of software testing where individual units or components of software/application are tested.
- The purpose is to validate that each unit of the software code performs as expected.
- Unit Testing is done during the development (coding phase) of an application by the developers
- Unit Tests isolate a section of code and verify its correctness.
- A unit may be an individual function, method, procedure, module or object.

▼ What is JUnit ?

- JUnit is a free and open source Unit Testing framework for Java application
- It is developed by Kent Black and Erich Gamma
- Its first version was released in 1997

- It became one of the most popular testing frameworks in Java Community due its ease of use.
- It is lightweight testing framework which allowed Java Developers to write unit test-cases in Java language.
- The current version is Junit 5.

JUnit Architecture :



JUnit Annotations:

- @Test : This annotation marks a method as a test case. The test runner will execute this method to verify the functionality of your code
- @BeforeEach: This annotation indicates a method that should be executed before each test case. This is useful for setting up common test data or

objects

- @AfterEach: Similar to `@BeforeEach`, this annotation marks a method that should be run after each test case. It's commonly used for cleaning up resources or resetting state.
- @BeforeAll: This annotation is for a method that should be executed only once before all test cases in a class are run. It's helpful for setting up global test fixtures or resources.
- @AfterAll: The opposite of `@BeforeAll`, this annotation marks a method that should be executed only once after all tests in a class have finished. It's used for cleaning up global resources.
- @ParameterziedTest: This annotation is used with JUnit Jupiter (version 5) for tests that take data from a source and run the test multiple times with different data sets.

#DataSource for @ParameterizedTest

- @ValueSource: This annotation is used to provide a simple list of literal values for a single test method parameter. It supports primitive data types like `int`, `long`, `String`, etc., and allows you to specify the values directly in the annotation.
- @CsvSource: This annotation is used to provide test data in a comma-separated value (CSV) (excel) format. You can specify multiple rows of data within curly braces, where each row represents a set of arguments for the test method.
- @CsvFileSource: Similar to `@CsvSource`, this annotation allows you to read test data from a separate CSV file. This is useful for larger datasets or when you want to keep the test data separate from the test code.
- @MethodSource: This annotation offers the most flexibility. It allows you to specify a method that dynamically generates the test data. This method can be within the test class itself or another class and can return a stream of arguments or individual arguments using the `Arguments` class.

Important

- @RepeatedTest: This annotation marks a test case to be executed multiple times. You can specify the number of repetitions within the annotation itself.
- @Disabled: This annotation marks a test case to be skipped during test execution.

Junit Assertions:

- JUnit 5 assertions helps us in validating the expected output with the actual output of a test case.
- In short, assertions are nothing but static methods that we call in our tests to verify expected behavior.
- All Junit Jupiter assertions are present in the org.junit.jupiter.Assertions class.

JUnit Assertion Methods:

- assertEquals and assertNotEquals
- assertTrue and assertFalse
- assertNull and assertNotNull
- assertSame and assertNotSame
- assertThrows

Program for Junit4-App:

💡 public class CalculatorTest {

private static Calculator c = null;

@BeforeClass
public static void init() {

c = new Calculator();
}

@AfterClass
public static void destroy() {
c = null;
}

@Test
public void testAdd() {

Integer actualResult = c.add(10,20);
Integer expectedResult = 30;

assertEquals(expectedResult,actualResult);

}

@Test

public void testMultiply() {

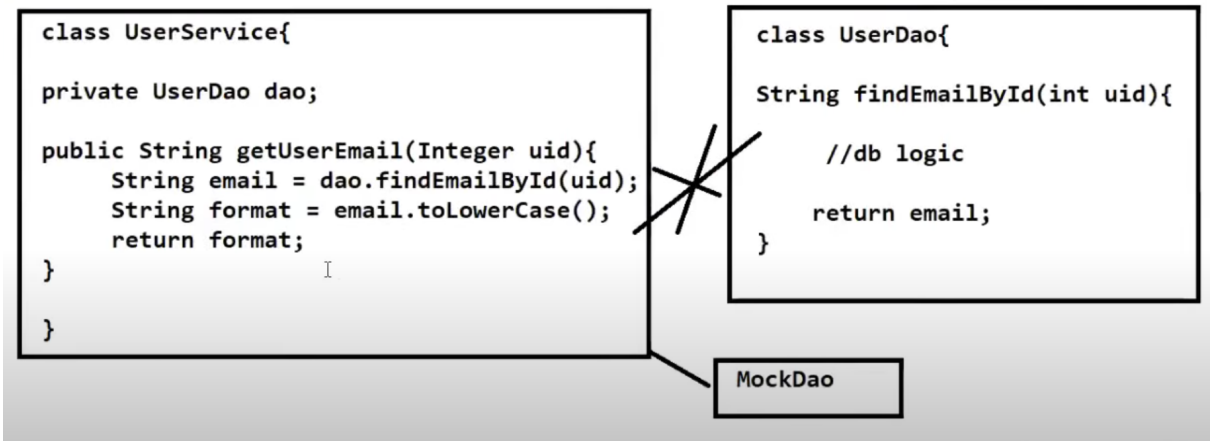
```
     Integer actualResult = c.multiply(2,3);
     Integer expectedResult = 6;
     assertEquals(expectedResult,actualResult);
 }
```

Integer actualResult = c.multiply(2,3);
Integer expectedResult = 6;
assertEquals(expectedResult,actualResult);
}

}

What is Mocking ?

- Mocking is a process used in **unit testing when the unit being tested has external dependencies**.
- The purpose of mocking is to **isolate and focus on the code being tested** and **not** on the **behavior or state of external dependencies**.
- In mocking, the **external dependencies** are replaced by **closely controlled replacements objects** that simulates the **behavior of the real ones**. (substitute objects will)
- The **process of creating substitutes objects** for **real objects** is called **mocking**.
- There are three main possible types of replacement objects - **fakes**, **stubs** and **mocks**.



Here, we want to test UserService class but it has external dependency on UserDao. So to perform Unit Testing, it has to be in Isolation and not dependent

on external factors/dependecies.

We used concept of mocking where we will create substitute objects for UserDao class.

Mock Frameworks:

- There are two types of Mocking Frameworks are available
  - Stub Based Mock Frameworks (Ex : Easy Mock)
  - ByteCode Manipulation Based Mock Frameworks (Ex : Power Mock, Mockito)