**Junit 5 Testing / Jupiter Testing**

Testing in Software Development refers to something which is done post/after the development, and which is done by a separate dedicated tester team/QA Engineers/Tester in the industry.

* **Unit Testing:**
  + During the development of the application, there are so many classes and in those classes there will be so many methods implemented. One Unit refers to one method of those classes.
  + In method, we implement some behavior/operation to perform that method consider as an one unit of an application. In small applications, it could be one method considered as one unit and in the bigger application, there could be group of methods or classes considered as an one Unit of an application.
  + Dividing the application into the Unit based on the behavior or features trying to develop.
  + Generally, we refer Unit means one method, in which some behavior will be there, some functionality will be there and some task/operation to perform written. And testing this one method is considered as a Unit Testing.
  + In the Unit testing, first we write a Test, and based on the test we develop specific Unit then will run the test to validate is it passing or not. If it pass successfully then we have developed Unit perfectly and if it does not pass, we need to make modifications into the method to pass the test successfully.
  + While we write an Unit test, it has to work during that implementation time as well, also in future when new features will add, at that particular time also it has to work.
* **Pre-requisites for Testing:** Java(from beginner to intermediate)
* **What will you get:** Junit-5 Framework (Theory + Practical)
* **Outcome**: Junit5 or Junit Jupiter Testing Framework Expert for Unit Testing
* **Normal Testing vs Unit Testing:**
  + Testing in Software Development, it refer to something which is done post/after the development, and which is done by a separate dedicated tester team/QA Engineers/Tester in the industry.
  + Unit testing also tests the application and testing the functionality of the application which is done by Software Developer itself. The one who write or develop the code, he will only perform the Unit testing.
* **Flow of completing the development of an application:**
  + **Follow SDLC (Software Development Lifecycle):** 
    - **Designing:** Before we start the development or implementation, first we design the application or make blueprint of it.
    - **Development:** Once designing is done, then in Development stage, the actual code writing and implementation will done in Development stage.
    - **Testing:** Once implementation has been done, then will perform Testing of an application. In this stage, will validate the application whether it is working fine and producing expected outputs or not. This type of testing will done post the development.

**(This testing stage is different from the Unit Testing. In this stage, testing will done by Testing team/QA Team)**

**(While Unit test done during we write a code and it is done by Software Developer Team/Software Developer who are responsible to develop the application)**

* **Performing Unit test without having framework:**
  + If we want to perform a Unit test without having framework, we need to write individual test cases in the main method of the application and need to write code logic for all individual scenarios.
* **Steps need to follow in the Unit Testing:**
  + **Prepare:** 
    - First we need to set up a test environment that will includes importing Junit, Creating test classes and writing test methods.
    - Set up environment & test class.
    - Add annotations @Test, @BeforeEach, @AfterEach, @BeforeAll, @AfterAll, @Disabled
  + **Provide Testing Input:**
    - Pass inputs to the method you want to test.
    - Calling method with predefined input parameters.
    - Simulating user input, API parameters, or mock data.
  + **Run the Test:**
    - **To execute the test methods using JUnit runner.**
    - **Annotating test methods with @Test.**
    - Execute the test case using the test runner (either through your IDE or Maven/Gradle).
    - JUnit automatically runs the methods marked with @Test.
    - Using build tools (e.g., Jenkins, GitHub Actions) for CI.
    - Use @Test annotation, run via IDE or CLI
  + **Provide Expected Output:**
    - Define what correct output should be.
    - Define what the correct result should be for the given input.
    - Use hardcoded, clear values as expectations
  + **Perform Assertion / Verify the Result:**
    - Compare actual vs. expected results.
    - To compare actual output against expected and determine pass/failure.
    - Compare the actual output from the test to the expected output using assertion methods.
    - There are lot of APIs which will perform the assertion with Junit-5.
    - Use ***assertEquals***, ***assertTrue***, ***assertThrows***, include helpful messages
  + **Report Test Result:**
    - Output pass/fail for feedback.
    - JUnit will report whether the test passed or failed.
    - Integrate with CI/CD, generate readable reports, use test coverage tools
    - Generating reports with tools like **Surefire**, **JUnit HTML Reports**, or **Allure**.
  + In the place of manually testing unit, while we use Junit-5 testing framework, we just have to do Prepare step, have to provide Input to the test method, and just have to provide an Expected output of the method. After that Junit will take care of running the test cases, perform the assertion and verify that is that actual test matching with actual expected results and Junit will use own way to alert the developers whether test case is pass or fail.
* **Junit 5 Architecture: (Junit 5 is not Junit 4 + new features)**
  + **Junit Platform:**
    - It is core of component of Junit 5 architecture.
    - It is a foundation for running test engines (e.g., Jupiter or third-party engines).
    - Used by IDEs, build tools (Maven/Gradle), and CI tools.
    - It provides an environment to run tests which consist of test runners.
    - Developers do not directly works with platform even though it is core of Junit 5 architecture.
    - One of the key components of Junit Platform is “***Launcher API***” that allows test frameworks and IDEs to launch and execute tests.
    - Platform is responsible for run your test cases.
  + **JUnit Jupiter:**
    - JUnit Jupiter API provides the API for writing tests and test cycle control and the engine for running those tests.
    - JUnit Jupiter API contains all the methods to perform an Assertion / verify test results, all the annotations, which we will going to use in order to write a test and platform will be able to run the test which we have written.
  + **JUnit Vintage:**
    - If we already written Test cases in JUnit 4 and now want to migrate to JUnit 5, JUnit 5 does not provide backward compatibility as 5 is the whole new thing.
    - So, for backward compatibility, JUnit5 provides JUnit Vintage API which we can use to run existing test cases which we have written in old JUnit (3 or 4) and want to run in Junit 5.
    - JUnit 5 does not have direct backward compatibility, but we need to use JUnit Vintage API for it.
    - Run existing JUnit 3/4 test code in the JUnit 5 ecosystem.
    - Migrate to JUnit 5 incrementally, not all at once.
    - Use modern test tools (IDE, CI/CD, reporting) with older tests.
  + **Extension:**
    - Extension is the enhance the capability of JUnit 5 means we can have our own API and we make platform works for it without having to use Jupiter.
    - Usually we mainly use Jupiter libraries / API whenever we are working with JUnit 5 and all the annotations and methods for assertions which are associated in order to write a test all of them are part of Jupiter. If we want to use those libraries and method, we need to add the Jupiter dependencies into our project using Maven and we can use those libraries, annotations and methods within that Jupiter API.
    - But if we want to use third party API and don’t want to use Jupiter API then we need to use Extension.
    - When we want to run a Test, we can use any of the IDEs such as Eclipse, VS Code, IntelliJ etc.., all are integrated with JUnit 5 framework and we can run a test by just right-clicking on the test case.



* **Implementation of Unit Testing code for application:**
  + To start with writing a code for unit testing, we will need an IDE. We can use any IDEs such as Eclipse, VS Code, IntelliJ, etc. but make sure it can create a Maven project.
  + **Why Maven Project? Maven is a build tool:**
    - Because Maven provide a specific structure in which we can easily write test by using JUnit.
    - To write JUnit test, it expected us to follow specific structure and Maven will provide that structure automatically and also in the real-world industry we use a build tool Maven.
  + **Steps to create a Maven Project for JUnit test in Eclipse:** 
    - **Step 1: File > New > Maven Project**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Or File > New > Other and In the Dialog Box Select Maven > Maven Project**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 2: Select checkbox “Create a simple project (skip archetype selection)” and “Use default Workspace location” > click on “Next”**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 3: Provide the details below and click on “Finish” button:**
      * **Group Id:** It is whatever we give convention for the package / folder name. We create a packages and inside that package we create multiple class.
        + **Ex.: com.JavaJUnitTest.****Learning**
        + While we pass package name with “.”, each “.” Consider as go inside package and create another package in it. For above example, we have written com.JavaJUnitTest.Learning, so it will create “com” package/folder inside that will create “JavaJUnitTest” package and inside that it will create “Learning” package
        + **com > JavaJUnitTest > Learning**
      * **Artifact Id:** It is a name of the project. Whatever the name we will provide, it will be the project build output.
        + **Ex.: employment-management**
        + For the Artifact ID: employment-management, it will create a JAR or WAR file with the name employment-management.jar file.
      * **Version:** It is a version of the project. We can keep its default value.
        + **Ex.: 0.0.1 SNAPSHOT**
      * **Packaging:** Select JAR, WAR or POM based upon our application type.
        + **jar:** Default. Creates a Java Archive file. Used for libraries and standalone Java apps.
        + **war:** Web Application Archive. Used for Java web applications (runs in servlet containers like Tomcat).
        + **pom:** Used for parent or aggregator projects that manage dependencies but don’t generate code/artifacts.
        + **ear:** Enterprise Archive. Used for Java EE applications (contains JARs + WARs).
        + **maven-plugin:** Used to build Maven plugins.
      * **Name:** It’s mainly used for display purposes (e.g., in logs, documentation, or Maven UIs like Jenkins, IntelliJ, or Maven Central). It does not affect the artifact's filename or Maven’s build process.
        + **We can leave it empty as well.**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Once you provide all the required details, it will create below empty Java-Maven project with below file structure**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + **File Structure of Java-Maven Project:**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **src/main/java:** On this location, we actually write code for our application. Means for our application, whatever classes are needed, whatever files are needed, will be located to this location.
    - **src/main/resources:** On this location, we store resources such as configuration files (application.properties file, .yaml, .xml files) , data files (.CSV, JSON, XML), database scripts, Images, fonts ect.
    - **src/test/java:** On this location, we will write the test cases for our application.
    - **src/test/resources:** On this location, It holds **resources needed only during testing**, not during regular application runtime. These resources are included in the **test classpath**, not in the final JAR/WAR.
      * **Below are the Typical contents:** 
        + Test configuration files (e.g., test-config.properties)
        + Mock data files (e.g., test-data.json, sample-input.csv)
        + XML or YAML files for unit test inputs
        + Test templates or scripts
    - **pom.xml:** In this file, we actually write our dependencies requirements that will use for our application. There are several dependencies code available on the internet, we need to use as per our requirement.
      * **When we want to add more dependencies, we need to write dependencies code between <dependencies></ dependencies> tag.**
      * **We can get most of the dependencies on** [**https://mvnrepository.com/**](https://mvnrepository.com/) **website for personal projects.**
      * **For industry projects, we need to pass industries’ private environment code.**
  + **Steps to add Test Case for Application: Add Manually**
    - **Step 1:** Add Java class file in src/main/java location. For that “Right-click-on src/main/java package > Select “New” > Select “Class””

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 2:** Provide the details below and click on “**Finish**”
      * **Package:** **Ex.:** com.JavaJUnitTest.Learning
      * **Name:** Provide class name. **Ex.:** Calc

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 3:** After we implement and add the application code to the class files in “src/main/java”, we need to write a test cases for each of the units/methods. That we have to add in the “src/test/java” location.

**To add the test case first** “Right-click-on src/test/java package > Select “New” > Select “JUnit Test Case”

**A screenshot of a computer

AI-generated content may be incorrect.**

**If “JUnit Test Case” option is not available in the list then** “Right-click-on src/test/java package > Select “New” > Select “Other…”

**A screenshot of a computer program

AI-generated content may be incorrect.**

**In the dialog box, search “JUnit Test Case” or Select "Java > JUnit > JUnit Test Case and click on “Next”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 4:** In the next step, click on “**Brows**” button next to the “**Class under the test**” and Search for class name on which you want to write test case > select it > Press “**OK**” button. **Ex.** Calc.

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 5:** In the “**New JUnit Test Case**” window, provide below details > Click on “**Finish**” button:
      * **Package:** Provide package name same as “**Class under test**” and do not add class name.

**Ex:** If we have “**com.JavaJUnitTest.Learning.Calc**” then copy only “**com.JavaJUnitTest.Learning**”

* + - * **Name:** We can pass any name as per our convenience but for better convenience, write ClassName followed by Test.

**Ex.**: If we have class name “Calc” then our test case name will be “CalcTest”

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will add test file on “src/test/java” location along with the package name we provided.**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 6:** To run the Unit test, right click on test file > Select “Run As” > Select “Junit Test”

**A screenshot of a computer program

AI-generated content may be incorrect.**

**It will display result in Junit tab: for test Fail**

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will display result in Junit tab: for test Success**

**A screenshot of a computer

AI-generated content may be incorrect.**

**In the concel for above “Hello Test” SOUT**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + **Steps to add Test Case for Application: Add Automatically using IDE**
    - **Step 1:** After we implement and add the application code to the class files in “src/main/java”, right click on the class file we have created in “src/main/java (EX.: Calc.java)” > Select “New” > Select “JUnit Test Case”.

**A screenshot of a computer

AI-generated content may be incorrect.**

**If “JUnit Test Case” option is not available in the list then** right click on the class file we have created in “src/main/java (EX.: Calc.java)” > Select “New” > Select “Other…”

**A screenshot of a computer

AI-generated content may be incorrect.**

**In the dialog box, search “JUnit Test Case” or Select "Java > JUnit > JUnit Test Case and click on “Next”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 2:** All the required details will be populated automatically in the next dialog box and we just have to click “**Finish**”

**A screenshot of a computer

AI-generated content may be incorrect.**

* + **Steps to create a Simple Java Project for JUnit test in Eclipse and add application code and test code: without having to create Maven project**
    - **Step 1: File > New > Project**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 2: Select “Java Project” > Click on “Next”**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 3: Provide “Project Name” , keep other options as default > Click on “Finish”**

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will create a simple empty java project with “*src”* folder:**

**A close-up of a computer screen

AI-generated content may be incorrect.**

**This “*src*” folder will only contain actual java application code with multiple packages, class files. It does not contain any test code. For test code we need to create separate “test” folder.**

* + - **Step 4: Create separate folder to write test code. For that, right click on Project name > New > Select “Source Folder”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 5: Provide “Source Folder” name > Click on “Finish”**

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will create “test” folder under that project:**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 6: Add Class file or application code to “src”.**

**Right click on “src” > New > Select “Class”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 7: Provide Package name next to “Package:” field and**

**class name next to “Name:” field.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will create Java file under “src” folder**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 8: Add Test code file to “test” folder.**

**Right click on class file from “src” folder on which we want to write a test > New > Select “Other…”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 9: Search for “JUnit Test Case” > Click on “Next”.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 9: Provide Package name next to “Package:” field and**

**Test file name next to “Name:” field. >**

**Change “Source Folder” to “test” folder instead of “src” by clicking on “Browse”**

**Click on “Finish”**

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will create a test code file under test folder.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**^**

**||**

**||**

**Above**

**All Steps and Code**

**For JUnit 4**

**From Below**

**All Steps and Code**

**For JUnit 5**

**||**

**||**

**V**

* **JUnit 5:**
  + JUnit 5 testing framework also known as JUnit Jupitar.
* **Steps to create a Simple Java Project for JUnit 5 test in Eclipse:** 
  + - **Step 1:** File > New > Project

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 2:** Select “Java Project” > Click on “Next”

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 3:** Provide “Project Name” , keep other options as default > Click on “Finish”

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will create a simple empty java project with “*src”* folder:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**This “*src*” folder will only contain actual java application code with multiple packages, class files. It does not contain any test code. For test code we need to create separate “test” folder.**

* + - **Step 4: Create separate folder to write test code. For that, right click on Project name > New > Select “Source Folder”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 4: Create separate folder to write test code. For that, right click on Project name > New > Select “Source Folder”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 5: Provide “Source Folder” name > Click on “Finish”**

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will create “test” folder under that project:**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 6: Add Class file or application code to “src”.**

**Right click on “src” > New > Select “Class”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 7: Provide Package name next to “Package:” field and**

**class name next to “Name:” field. > Click on “Finish”**

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will create Java file under “src” folder**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 8: Add Test code file to “test” folder.**

**Right click on class file from “src” folder on which we want to write a test > New > Select “Other…”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 9: Search for “JUnit Test Case” > Click on “Next”.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 9: Provide Package name next to “Package:” field and**

**Test file name next to “Name:” field. >**

**Change “Source Folder” to “test” folder instead of “src” by clicking on “Browse”**

**Click on “Finish”**

**Must select “New JUnit Jupiter test” and it is referring to JUnit 5**

**A screenshot of a computer

AI-generated content may be incorrect.**

**It will create a test code file under test folder.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* **Annotations of JUnit 5 testing:**
  + **@Test Annotation:**
    - @Test annotation is mainly use over the method to mark method as a test it means @Test annotation will inform test engine for that method where we have written out test.
    - @Test annotation comes under “***org.junit.jupiter.api***” package.
    - While we use JUnit @Test annotation, in Junit 5 the test method can be default, protected or public while in previous versions, test method has to be public.
    - By adding Test annotation, it is informing test engine that it is a test method that needs to be execute for the test.
    - While we run the Test method, by default its output/test result will be success, even we do not write anything inside the test method. During the execution, it will validate the logic we have written for testing, if it is failed then it will consider as a failed test.

**JUnit 4: Test Method has to be public**

**A screenshot of a computer

AI-generated content may be incorrect.**

**JUnit 5: Test Method can be public, protected or default**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* **What is Assertion in JUnit 5 testing:**
  + Assertion is all about Expectations vs Actual output/reality.
  + To compare actual output against expected and determine pass/failure.
  + Compare the actual output from the test to the expected output using assertion methods.
  + Use ***assertEquals***, ***assertTrue***, ***assertThrows***, include helpful messages
  + **When Expected result is equal to actual output, our test case will pass and green bar but when it does not equal then our test case will fail and give red bar.**
  + There are several Assertion static methods available in the JUnit framework. See the list below: All the methods come under org.junit.jupiter.api.Assertions.\*; package.
    - **assertEquals(expected, actual, “Error Message For Failed Test Cases”)**
      * Checks if two values are equal.
    - **assertNotEquals(unexpected, actual, “Error Message For Failed Test Cases”)**
      * Checks that two values are not equal.
    - **assertTrue(condition, “Error Message For Failed Test Cases”)**
      * Verifies the condition is true.
    - **assertFalse(condition, “Error Message For Failed Test Cases”)**
      * Verifies the condition is false.
    - **assertNull(actual, “Error Message For Failed Test Cases”)**
      * Checks that the object is null.
    - **assertNotNull(actual, “Error Message For Failed Test Cases”)**
      * Checks that the object is *not* null.
    - **assertSame(expected, actual, “Error Message For Failed Test Cases”)**
      * Verifies both refer to the same object.
    - **assertNotSame(unexpected, actual, “Error Message For Failed Test Cases”)**
      * Verifies they do *not* refer to the same object.
    - **assertArrayEquals(expected, actual, “Error Message For Failed Test Cases”)**
      * Checks two arrays are equal in content and order.
    - **assertIterableEquals(expected, actual, “Error Message For Failed Test Cases”)**
      * Checks two Iterable objects are equal.
    - **assertLinesMatch(expected, actual, “Error Message For Failed Test Cases”)**
      * Compares two lists of strings (e.g., lines of text).
    - **assertThrows(expectedException, executable, “Error Message For Failed Test Cases”)**
      * Expects an exception to be thrown.
    - **assertDoesNotThrow(executable, “Error Message For Failed Test Cases”)**
      * Ensures no exception is thrown.
    - **assertTimeout(duration, executable, “Error Message For Failed Test Cases”)**
      * Fails if execution exceeds the given time.
    - **assertTimeoutPreemptively(duration, executable, “Error Message For Failed Test Cases”)**
      * Similar to assertTimeout but interrupts if timeout is exceeded.
    - **fail(“Error Message For Failed Test Cases”)**
      * Fails a test unconditionally (useful as a placeholder).
  + JUnit 5 supports all the features of Java 8 such as Lambda Expression, Stream API, Collection API, etc.
  + For writing test cases, we can give test() method to any name also we can add multiple test methods with different names. Only thing we have to do is, we need to add @Test annotation above the method declaration to tell Test engine that these are a test methods.
  + While we write multiple test methods, during execution, it will show how many test methods implemented as well as, which are pass and which are failed in the output window. See the below snapshots:

A screenshot of a computer

AI-generated content may be incorrect.

Failed Test

Passed Test

A screenshot of a computer

AI-generated content may be incorrect.

* + When we have written multiple test method but when we want to run only one individual test method among all the following the steps below: And it will produce output for that test case only.
    - **Step 1:** Select the test method we want to run > Right-click on it.
    - **Step 2:** Select “Run As” > Click on “JUnit Test”

**A screenshot of a computer

AI-generated content may be incorrect.**

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

* **What is TDD (Test Driven Development) in JUnit 5 testing:**
  + When we follow TDD process to develop out applications, we mainly use JUnit testing framework for it.
  + In the TDD approach, first we need to write test cases then write an actual code.
  + **Test-Driven Development (TDD) is a software development approach where tests are written before writing the actual code. The idea is to define and validate what the code will do by first writing test cases that fail, then writing the code to make them pass.**
  + Steps to follow TDD approach:
    - **Step 1:** Just declare all the methods empty in the application code class file so that we can invoke it to the test code.
    - **Step 2:** Write test cases related to that method
    - **Step 3:** Implement the application code logic according to the test case we have written.
* **Assertion Methods of JUnit 5 testing:**
  + **assertEquals** (ExpectedResultValue/Object, ActualResultMethodGeneraedValue/Object);
    - In this method either we can pass expected and actual values directly as a parameter or we can create an object of Class(on which we want to write a test case) in the test file and call the method of the class and store its result in the variable(This will be actual value) and create another variable and store expected result in it. And pass both expected and actual variables to the method parameters.
    - assertEquals(e,a) method is mainly use to compare actual and expected results from the method.
    - **Syntax:**

assertEquals(expectedResultValue, actualResultValue, “Add Message if required”);

**OR OR OR OR**

ClassName objectName = new ClassName();

dataType actualVariable = objectName.methodName();

dataType expectedVariable = expectedValue;

assertEquals(expectedVariable, actualVariable, “Add Message if required”);

**OR OR OR OR**

ClassName objectName = new ClassName();

*assertEquals*(ExpactedValue, objectName.methodName(passParametersIfRequired) , “Add Message if required”);

**A screenshot of a computer

AI-generated content may be incorrect.**

* + In the above syntax, we have written multiple lines of code by defining local variables, store values and passing those values to the assert method. Instead of that we can simply pass values to inline directly to the assert method. For that follow below steps:
    - **Step 1: Right-click on local variable from the method > Select “Refactor” > Select “Inline”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 2: In the dialog box, preview the code and click on “OK”**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Code after refactoring inline :**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Use assertEquals() method for String dataType:**

**Class File:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Test File:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **In the assertEqual() method, we can also pass the error message for better understanding which test case has failed.**
    - **This message will only display when the test case has failed.**
    - **Syntax:** Passing error message simple way OR Static way
      * **assertEqual(expectedResult, actualResult, “Error Message For Failed Test Cases”)**
    - **See the example below:** Without using Supplier Functional Interface and Lambda Expression

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Syntax:** Passing error message Using Supplier Functional Interface and Lambda Expression
      * **assertEqual(expectedResult, actualResult, () 🡪 “Error Message For Failed Test Cases”)**
    - **See the example below:** Using Supplier Functional Interface and Lambda Expression

A screenshot of a computer

AI-generated content may be incorrect.

* + **Difference Between Static and Supplier-Based Error Messages in JUnit 5:**
    - **Passing a Static Error Message in Assert Method:**
      * **Syntax**: assertEquals(expectedValue , actualValue, "Error Message.");
      * In this way, the third argument in the method will be static error message.
      * This message is always constructed and stored whether the test pass or fail and only display when the test got failed.
      * If constructing the message is expensive, it's inefficient because it's built even when the test passes.
    - **Passing a Error Message Using Supplier Functional Interface:**
      * **Syntax:** Using Lambda Expression
        + assertEquals(expectedValue, actualValue, () 🡪 "Error Message.");
        + In this way, the third argument is a Supplier<String>, a functional interface from Java 8. As it is functional interface, we can directly use Lambda expression.
        + The message only constructed if the assertion fails.
        + It is also known as lazy evaluation.
        + As it is generating message while test case fails, it helps with performance.
        + **It improves performance and efficiency in large test suites or when error messages are dynamically constructed.**
        + **In this way when the test case is pass, that error message part wont be even considered.**
        + **It is an efficient way of passing error messages for test cases.**
  + **assertNotEquals() method:**
    - This method will perform exactly opposite way of assertEquals() method.
    - In **assertNotEquals()** method, if the expected result and actual result will be same then test case will fail. But when the expected result and actual result will be different then case will pass.
    - **Syntax:** Simple method with unexpected and actual value parameter
      * **assertNotEquals (unexpectedResult, actualResult);**
    - **Syntax:** Method with passing static error message
      * **assertNotEquals (unexpectedResult, actualResult, “Error Message If required”);**
    - **Syntax:** Method with passing error message using Lambda Expression and Supplier Interface
      * **assertNotEquals (unexpectedResult, actualResult, () 🡪 “Error Message If required”);**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + **assertTrue() method:**
    - assertTrue() method will use to check whatever value we have passed whether it is true or false.
    - In this method, we need to pass the logic of the condition in the parameter.
      * Ex.: int number = 10;

assertTrue(number % 2 == 0);

* + - * In above example, we are checking given number is odd or even and we have written its logic inside the method.
    - **Syntax:** Simple method with passing condition only
      * **assertNotEquals (unexpectedResult, actualResult);**
    - **Syntax:** Method with passing condition and static error message
      * **assertNotEquals (unexpectedResult, actualResult, “Error Message If required”);**
    - **Syntax:** Method with passing condition and error message using Lambda Expression and Supplier Interface
      * **assertNotEquals (unexpectedResult, actualResult, () 🡪 “Error Message If required”);**
* **pom.XML file || Basic structure of pom.xml for JUnit-Maven project || Update Project Steps:**
  + The ***pom.xml*** file is the ***Project Object Model file*** used in Maven to manage a Java project's dependencies, build configuration, and plugins.
  + **Basic Predefined pom.xml file:**

A screenshot of a computer

AI-generated content may be incorrect.

* + **Model version and Project Coordinates:**

**A close-up of a computer code

AI-generated content may be incorrect.**

* + - For above block***, modelVersion*** will always be 4.0.0 for Maven project v3 or v4
    - ***groupId, artifactId and version,*** all will be same as we pass while we have created a project. All these values will populate automatically.
    - ***<groupId>, <artifactId>, <version>*** *are uniquely identifies the project.*
  + ***Properties*** section**: It Sets encoding, Java version, and JUnit version.**

***A computer code with text

AI-generated content may be incorrect.***

* + - **sourceEncoding:** Character encoding used when reading/writing source files.
      * Keep it default
    - **maven.compiler.source:** Sets Java source code compatibility (here, Java 14).
      * Set which java version is going to use, we can change its version
      * But the default version it will pick 1.5
    - **maven.compiler.target:** Sets Java bytecode compatibility (also Java 14 via property reference).
      * We can pass this value from the tag we use to set java version along with ${passTagName} sign
      * **Ex:** ${maven.compiler.source}
    - **junit.jupiter.version:** Defines the version of JUnit 5 (Jupiter engine) used in dependencies.
      * We can set JUnit version in this tag.
  + ***Dependencies*** section: **Adds JUnit 5 library for unit testing.**

A computer code with black text

AI-generated content may be incorrect.

* + - **<dependencies></dependencies> tag:**
      * In this section, we can add all the required dependencies between <dependencies></dependencies> tag.
    - **<dependency></dependency> tag:**
      * For each of the dependency, we need create separate <dependency></dependency> tag inside <dependencies></dependencies> tag.
    - **<groupId> </groupId> tag:**
      * In the **groupId** tag where the API is present.
    - **<version></version> tag:**
      * While we add the dependencies, it required to add its version as well. For easiest way, we can declare a separate tag for each of the dependencies in above Properties section and while add the version in dependencies, we just have to pass the name of that tag between ${tagName} in the <version></version> tag.
      * **Ex:** <properties>

**<junit.jupiter.version>5.10.0<junit.jupiter.version>**

</properties>

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

**<version>${junit.jupiter.version}</version>**

<scope>test</scope>

</dependency>

</dependencies>

* + - **<scope></scpoe> tag:**
      * We defined scope tag inside of dependency tag because based on the scope tag, test engine, or run engine will decide when and how that dependency will use during the build lifecycle.
      * There are predefined values only we can pass in the scope tag. Below are the values:
        + **compile (default)**

**Usage:** Needed for compiling and running the main code.

**Available in:** compile, test, runtime

**Typical for:** Core libraries your app depends on.

* + - * + **provided:**

**Usage:** Required for compilation but provided by the runtime environment (e.g., servlet API in a web container).

**Available in:** compile, test

Not packaged in your final artifact

* + - * + **runtime:**

**Usage:** Not needed for compilation but needed during execution.

**Available in:** runtime, test

**Example:** JDBC drivers, logging implementations.

* + - * + **test:**

**Usage:** Only used during test compilation and test runtime.

**Not included in production code.**

* + - * + **system (deprecated, rarely used):**

Similar to provided, but you must specify the JAR file manually with <systemPath>.

**Not portable across environments.**

* + - * + **If we do not pass any values in the scope, it will take “compile” as a default state.**
    - ***<build>, <plugins>, <plugin> and <configuration>*** section: **Configures Java compilation behavior.**

**A computer code with black text

AI-generated content may be incorrect.**

* + - * **build** tag:
        + This is the parent tag for defining build-related configurations in Maven.
        + It's an optional but powerful part of Maven.
      * **plugins** tag:
        + This is a container for all plugins that you want Maven to use during the build process.
        + Plugins **extend Maven functionality**, e.g., compiling code, running tests, packaging, etc.
        + Inside plugins tag, we need to create separate <plugin></plugin> tag to add each of the plugins.
      * **plugin** tag:
        + This defines one specific plugin.
        + In above example: maven-compiler-plugin—used to compile Java code. It is most important plugin to add to compile the code.
      * **artifactId** tag:
        + Identifies the plugin by name.
        + We cannot set any name for it. **You must use a valid Maven plugin artifact ID that exists in the Maven Central Repository or another repository you're using.**
        + ***maven-compiler-plugin*** is the official Maven plugin used to compile your .java source files into .class files.
      * **version** tag:
        + Specifies the version of the compiler plugin to use.
        + Version 3.8.1 is widely used and compatible with Java 8 to Java 14+.
        + Important to set explicitly to avoid inconsistencies across environments.
        + Prevents build errors due to Maven using an outdated compiler plugin.
      * **configuration** tag:
        + Allows customization of plugin behavior.
        + For ***maven-compiler-plugin***, we configure Java language compatibility settings here.
      * **source** tag:
        + Sets the Java source version.
        + This tells Maven which Java version to expect your code to be written in.
        + It tells Maven that my source code uses Java 14 syntax/features.
        + **Ex**.**:** <maven.compiler.source>14</maven.compiler.source>
      * **target** tag:
        + Sets the Java bytecode version.
        + This is the version of Java the compiled .class files should be compatible with.
        + For the target version, it typically equals to source but we can also use lower version if need.
        + It tells maven that compile my code to run on JVM version 14.
        + **Ex.:** <maven.compiler.target>14</maven.compiler.target>
  + **After updating the pom.xml file, we need to update our project to align with the dependencies and Java version we have added. To update the project, see the steps below:**
    - **Before updating the project: Java version is v1.6**

**A screenshot of a computer program

AI-generated content may be incorrect.**

* + - **Step 1: Right-click on project > Go to “Maven” > Select “Update Project…”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **Step 2: Select project you want to update > Click “Ok”**

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - **After updating the project: Java Version change to v14**

**A screenshot of a computer

AI-generated content may be incorrect.**

* **How to run JUnit test manually or without eclipse IDE’s facility:**
  + This process is mainly use while our application is deploying in CICD pipeline. During CICD pipeline execution, build process will not install the eclipse and run test cases, it is not possible.
  + To resolve this issue, we need to execute our whole project using Maven Test. **For that we need to add plugin-sureFire plugin. This plugin, most of the companies use it for automation. To achieve it, see the steps below:**
    - **Step 1:** Add plugin code in the pom.xml file under <build> 🡪 <plugins> 🡪 <plugin> tag. Below is the <plugin> code for it:

**<plugin>**

**<artifactId>maven-surefire-plugin</artifactId>**

**<version>2.22.2</version>**

**</plugin>**

* + - **Step 2:** Right-click on project > Go to “Run As” > Select “Maven Test”

**A screenshot of a computer

AI-generated content may be incorrect.**

**OR OR OR OR OR OR OR OR OR OR OR**

* + - Right-click on project > Go to “Run As” > Select “Maven build…”

**A screenshot of a computer

AI-generated content may be incorrect.**

* + - Pass Maven Goal “test” in the Goal > Click “Run”

**A screenshot of a computer

AI-generated content may be incorrect.**

**Step 3:** After run project as a “Maven Test”, we will get below test results

A screenshot of a computer

AI-generated content may be incorrect.