JUnit Testing

Testing vs Unit Testing

Testing -> Test Engineers

Unit Testing -> Developers

SDLC (Software Development Life Cycle)

Step 1: Design

Step 2: Developing application (writing code)

Step 3: 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.

Defn:

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**.

- Unit testing helps testers and developers to understand the base of code that makes them able to change defect causing code quickly.
- Unit testing helps in the documentation.
- Unit testing fixes defects very early in the development phase, that's why there is a possibility to a smaller number of defects occurring in upcoming testing levels.
- It helps with code reusability by migrating code and test cases.

We have various types of unit testing tools available in the market, which are as follows: NUnit , JUnit, PHPunit etc.

Setup:

We can add Junit dependency through Maven project or by adding external libraries of jar file to the project

JUNIT5

Unit Testing Framework (JUNIT5) ?

=> prepare (setup a Test Environment, write TestMethods ...)

=> Provide testing input

=> Run the test

=> Provide expected output

=> Perform Assertion (verify the result)

=> Report Test Results (Alert Developer if test is failed (or) passed)

@Test

↳ Applied over methods to mark method as test

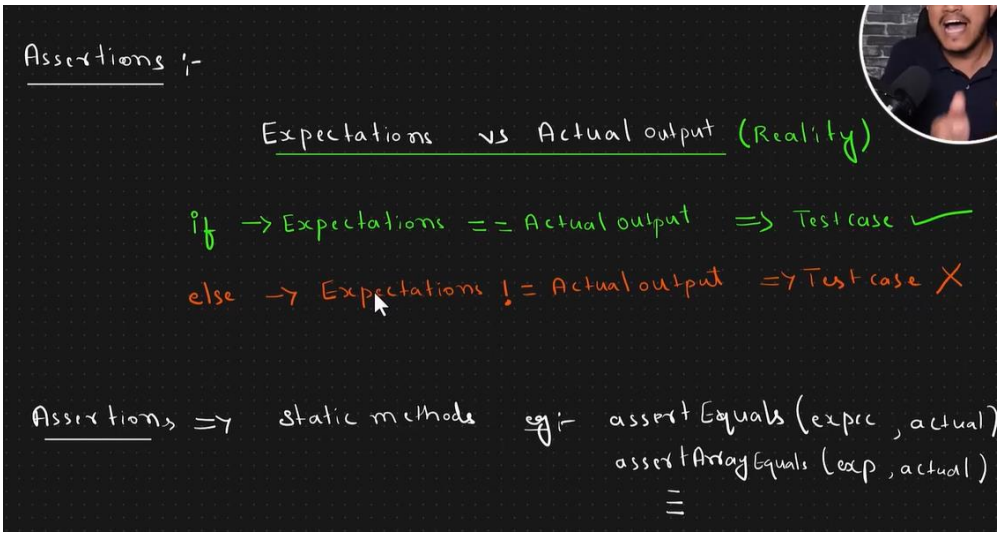
↳ → org.junit.jupiter.api ←

↳ Visibility of @Test Annotated method can be public, protected default.

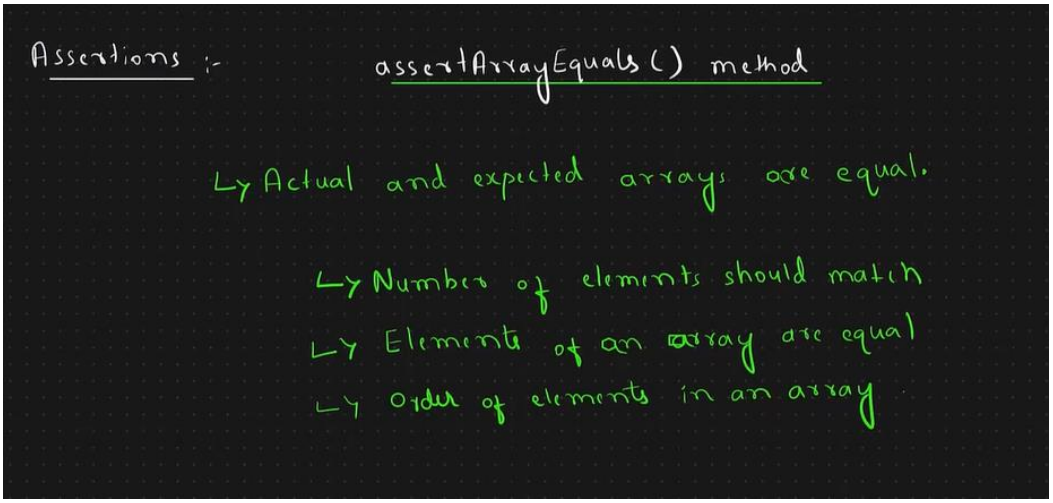
↳ Also, informs testengine what method needs to run

```
@Test
void testcomputeCircleArea()
{
    assertEquals(76.5, shape.computeCircleArea(5), "Area of circle calculation is wrong");
}

@Test
void testcomputeCircleArea_Supplier()
{
    assertEquals(76.5, shape.computeCircleArea(5), ()->"Area of circle calculation is wrong");
}
```



Note: Write test then develop -> test driven development



Eg:

```
  @Test
   void testArrays(){
       int[] expected = {2,3,4,5};
       int[] actual = {4,2,5,3};

       Arrays.sort(actual); // solve the case {2,3,4,5}

//         assertEquals(expected,actual);  // it will not check the inside elements it's
just checking the refernce of objects

       assertArrayEquals(expected,actual);
```

Throwing Exception while checking test-cases:

Note: Test cases should fail when there are no exception and pass if there is an exception

assertThrows();

How to achieve the best result via Unit testing?

Unit testing can give best results without getting confused and increase complexity by following the steps listed below:

- Test cases must be independent because if there is any change or enhancement in requirement, the test cases will not be affected.
- Naming conventions for unit test cases must be clear and consistent.
- During unit testing, the identified bugs must be fixed before jump on next phase of the SDLC.
- Only one code should be tested at one time.
- Adopt test cases with the writing of the code, if not doing so, the number of execution paths will be increased.
- If there are changes in the code of any module, ensure the corresponding unit test is available or not for that module.

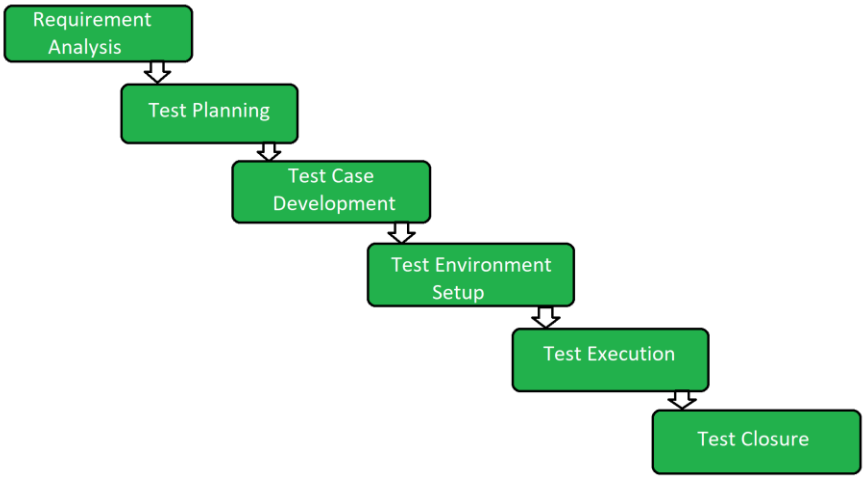
Advantages

- Unit testing uses module approach due to that any part can be tested without waiting for completion of another parts testing.
- The developing team focuses on the provided functionality of the unit and how functionality should look in unit test suits to understand the unit API.
- Unit testing allows the developer to refactor code after a number of days and ensure the module still working without any defect.

Disadvantages

- It cannot identify integration or broad level error as it works on units of the code.
- In the unit testing, evaluation of all execution paths is not possible, so unit testing is not able to catch each and every error in a program.
- It is best suitable for conjunction with other testing activities.

Lifecycle:



The **Software Testing Life Cycle (STLC)** is a structured process with six main phases: Requirement Analysis, Test Planning, Test Case **Development**, Test Environment Setup, Test Execution, and Test Closure. Testers first analyze and plan, then design and develop detailed test cases with specific conditions and outcomes. They set up the required testing environment, execute the prepared test cases, log any defects, and finally, close the testing cycle by documenting the results and finalizing reports.

The lifecycle of a test case within the **Software Development Life Cycle (SDLC)** is governed by the Software Testing Life Cycle (STLC), which encompasses phases like Requirement Analysis, Test Planning, Test Case **Development**, Test Environment Setup, Test Execution, and Test Cycle Closure. Test cases are created during the Test Case Development phase, where requirements are translated into actionable steps to verify the software's functionality, and they remain active through execution and closure to ensure quality and reliability.

@annotations Working flow:

Steps:

1. First Constructor() will call then
2. @BeforeAll annotaion will invoke
   – we will use this method for static methods()
   – common methods or units eg: JDBC conn
   -- RUN only once before invoking test cases
3. @BeforeEach annotaion will invoke
4. @Test annotaion will invoke
5. @AfterEach annotaion will invoke – just like cleaningup the previous invokes
6. @AfterAll annotaion will invoke

– we will use this method for static methods()
– common methods or units eg: close() connections
-- RUN only once after invoking test cases and annotations

7. @TestInstance(-------) --> to control the constructor execution every time when new object gets created by default constructor will automatically.
    a. TestIstance(TestInstance.Lifecycle.PER_METHOD) -- default behaviour it will invoke costuctor when new object is created
    b. TestIstance(TestInstance.Lifecycle.PER_CLASS) -- instance will create once means constructor calls only one time