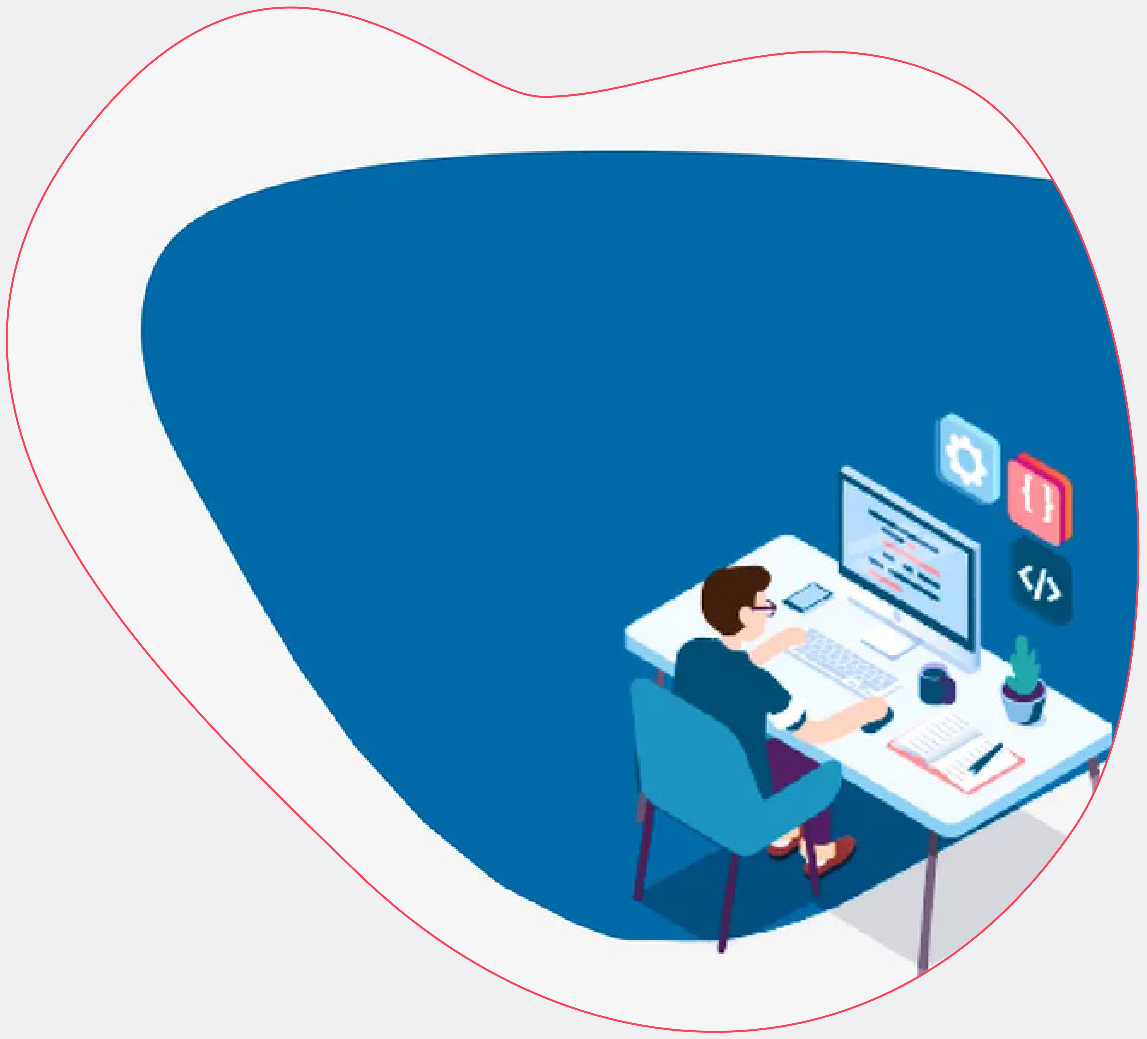
What are Test, Unit Test, and JUnit ?



musanabiyev

musanabiyev

What is a Test?
A test is a development-testing method used to verify whether a system produces the expected results according to predefined specifications.

What is a Unit Test?
Unit tests refer to tests performed piece by piece (part by part). While you can test an entire application or software, the tests should be carried out independently, without relying on other components.
Note: *In object-oriented programming (OOP) languages like Java, these parts are referred to as methods.*

What is JUnit?
JUnit is an advanced framework used for performing unit tests in the Java world.

Why is JUnit Testing Used?
JUnit is used to perform unit testing in Java, enabling faster software development. It is employed by developers to improve code quality.

What are the Rules for Writing Unit Tests?

1. Each unit test method should test only one method.

2. Each unit test should have its own scenario.
If scenarios are written sequentially, a failure in the first test will prevent the subsequent ones from running. A new test method should be created for each scenario.

3. Each unit test should follow specific steps.
It is recommended to adhere to the *//Given //When //Then* standard:
Given: Create the variables that will be used and configured for the test.

When: Perform the desired operation with the values set in the Given step.

Then: Define the expected result

4. The name of the test method should reflect the tested scenario.
There are various naming conventions for this, with the following being the most commonly preferred: (examples would follow in practice).

1. testFeatureName, e.g., testSumOfPositiveNumbersReturnsPositive

2. featureName, e.g., sumReturnsPositiveWhenAddingPositiveNumbers

3. should_ExpectedBehavior_When_Condition, e.g., should_ReturnPositive_When_PositiveNumberIsProvided

4. when_Condition_Expect_ExpectedBehavior, e.g., when_PositiveNumberIsProvided_Expect_ReturnPositive

5. given_InitialCondition_When_Condition_Then_ExpectedBehavior, e.g., given_Calculator_When_PositiveNumberIsProvided_Then_ResultIsPositive

6. method_Scenario_Result, e.g., sum_PositiveNumbers_Positive

7. method_Result_Scenario, e.g., sum_Positive_PositiveNumbers

Before moving to the next page, let's get acquainted with our plugin!

Auto API Generator

Simplify your API development with our Auto API Plugin for IntelliJ IDEA! Automatically generate CRUD APIs, DTOs, services, repositories, controllers, and integrate Swagger, Liquibase, and Lombok effortlessly. Streamline your workflow and enhance productivity!

Try it now: Auto API Plugin

5. Each tested component should be independent of others.
In other words, the test should be executed without relying on other parts. For this, mock or stub can be used. For example, if a service class uses a repository class, the test for the service class would depend on the repository object. This is an undesirable situation.

Mock:
It can be thought of as a type of proxy or fake object. It behaves like the original object but is empty inside and acts according to our needs. To use it, the Mockito.mock(ClassName.class) method is used. It is preferred over stubs.

Stub:
This refers to classes that we can create by extending the original class solely for testing purposes. It is not a preferred approach because it requires additional effort to create extra classes and code.

6. Each test method must operate independently from others.
No test method should invoke another test method directly.

7. In addition to all the aforementioned points, it can be concluded that unit testing serves to simplify the development process and the subsequent stages. When a unit test is in place, it becomes crucial for addressing issues such as code complexity, misunderstandings, and inconsistent results across different scenarios during runtime.

JUnit Lifecycle

JUnit subjects each test class to specific phases. These phases are as follows:

@BeforeClass: The method annotated with this is called first after the test starts. Regardless of how many methods are in a class, the method marked with this annotation is called only once per class. After this annotation is executed, methods annotated with **@Test** become executable. The method using this annotation must be defined as static; otherwise, the test will fail.

@Before: This is executed before the **@Test** annotation. The method marked with this annotation is run once at the beginning for all test methods. Thus, regardless of which test method we are executing, the method marked with this annotation will run one time. If a class has 10 test methods and we run them all at once, this method will be called and utilized 10 times.

@Test: This annotation marks the actual method that will be tested.

@After: This is executed after the **@Test** annotation. The method marked with this annotation is run once after all test methods have been executed. It serves as the counterpart to the **@Before** annotation.

@AfterClass: After all test methods have been executed and their tasks are completed, any method you wish to mark with this annotation will run. In other words, it designates a method that needs to be executed once at the end. The method using this annotation must be defined as static; otherwise, the test will fail.

To verify the correctness of the result, the overloaded static assert* methods found in the Assertions class are used. These methods are as follows:**

assertEquals(expected, actual): This method is used to check whether two values are equal.

assertArrayEquals(array1, array2): This checks the equality of two arrays based on their values without reference checks. The test will pass if the arrays have the same values.

assertNull(object): This checks whether the given object is null. The test will pass if it is null.

assertNotNull(object): This checks whether the given object is not null. The test will pass if it is not null.

assertNotSame(object1, object2): This checks that the two object references do not point to the same object.

assertSame(object1, object2): Functionally identical to **assertEquals()**. However, this checks for reference equality of the two objects, even if the **equals()** method is overridden in the objects.

assertTrue(condition): This checks whether the given condition is true. If the condition is true, the test will pass.

assertFalse(condition): This checks whether the given condition is false. If the condition is false, the test will pass.