**JUNIT**

**1. Introduction to JUnit**

JUnit is a widely used testing framework for that helps developers write repeatable tests to ensure the correctness of their code. The latest version, JUnit 5, is divided into three main modules:

1. **JUnit Platform** – Provides the foundation for launching tests.
2. **JUnit Jupiter** – Contains the new programming model and APIs.
3. **JUnit Vintage** – Provides backward compatibility for running older JUnit 3 and 4 tests.

**2. Setup and Dependencies**

To use JUnit 5, you need to add the appropriate dependencies to your project. If you're using **Maven** or **Gradle**, you can add the following:

**Maven**

xml

<dependency>

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

<artifactId>junit-jupiter-api</artifactId>

<version>5.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

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

<artifactId>junit-jupiter-engine</artifactId>

<version>5.8.1</version>

<scope>test</scope>

</dependency>

**Gradle**

groovy

testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'

testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'

**3. Writing Basic Tests**

JUnit tests are written in , and test methods are annotated with @Test. Here’s an example of a basic test:

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class CalculatorTest {

@Test

void testAddition() {

int result = 2 + 3;

assertEquals(5, result);

}

}

**Important Assertions:**

* assertEquals(expected, actual) – Verifies that two values are equal.
* assertNotEquals(expected, actual) – Verifies that two values are not equal.
* assertTrue(condition) – Verifies that the condition is true.
* assertFalse(condition) – Verifies that the condition is false.
* assertNull(object) – Verifies that the object is null.
* assertNotNull(object) – Verifies that the object is not null.

**4. Lifecycle Annotations**

JUnit 5 introduces several lifecycle annotations that can be used to set up or tear down tests before and after execution:

* **@BeforeEach** – Runs before each test method.
* **@AfterEach** – Runs after each test method.
* **@BeforeAll** – Runs once before all test methods (static method).
* **@AfterAll** – Runs once after all test methods (static method).

**Example:**

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.Test;

public class CalculatorTest {

@BeforeEach

void setUp() {

// Runs before each test method

System.out.println("Setting up before test.");

}

@AfterEach

void tearDown() {

// Runs after each test method

System.out.println("Tearing down after test.");

}

@Test

void testAddition() {

assertEquals(5, 2 + 3);

}

@Test

void testSubtraction() {

assertEquals(1, 3 - 2);

}

}

**5. Parameterized Tests**

JUnit 5 supports parameterized tests, which allow you to run the same test with different input data. You can use @ParameterizedTest with various sources like @ValueSource, @CsvSource, or custom arguments.

**Example:**

import org.junit.jupiter.api.ParameterizedTest;

import org.junit.jupiter.api.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class PalindromeTest {

@ParameterizedTest

@ValueSource(strings = {"madam", "racecar", "level"})

void testIsPalindrome(String word) {

assertTrue(isPalindrome(word));

}

private boolean isPalindrome(String word) {

String reversed = new StringBuilder(word).reverse().toString();

return word.equals(reversed);

}

}

**6. Exception Testing**

JUnit 5 provides a way to test for exceptions using assertThrows. This allows you to ensure that a certain exception is thrown under specific conditions.

**Example:**

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertThrows;

public class CalculatorTest {

@Test

void testDivisionByZero() {

assertThrows(ArithmeticException.class, () -> {

int result = 10 / 0;

});

}

}

**7. Test Interfaces and Inheritance**

JUnit 5 supports inheritance for test classes, allowing you to reuse setup/teardown logic. You can also create tests with shared behavior using test interfaces.

**Example of Test Inheritance:**

import org.junit.jupiter.api.Test;

public class CalculatorTest extends BaseCalculatorTest {

@Test

void testAddition() {

int result = 2 + 3;

assertEquals(5, result);

}

}

**8. Conditional Test Execution**

JUnit 5 introduces annotations to conditionally execute tests:

* **@EnabledIf** – Enables the test based on a condition.
* **@DisabledIf** – Disables the test based on a condition.
* **@EnabledOnOs** / **@DisabledOnOs** – Enable or disable tests based on the operating system.
* **@EnabledIfSystemProperty** / **@DisabledIfSystemProperty** – Based on a system property.

**Example:**

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.condition.EnabledIfSystemProperty;

public class ConditionalTest {

@Test

@EnabledIfSystemProperty(named = "os.arch", matches = "x86\_64")

void testFor64BitSystem() {

System.out.println("Test runs on 64-bit system");

}

}

**9. Grouping Tests with Tags**

JUnit 5 allows you to group tests by tagging them, which can be helpful for categorizing and filtering tests.

import org.junit.jupiter.api.Tag;

import org.junit.jupiter.api.Test;

public class TaggedTests {

@Test

@Tag("critical")

void testCriticalFeature() {

// Critical test

}

@Test

@Tag("non-critical")

void testNonCriticalFeature() {

// Non-critical test

}

}

You can filter tests by tags using build tools like Maven or Gradle.

**10. Running Tests with JUnit Platform**

JUnit 5 can be run using the command line, IDEs, or build tools. For example:

**Maven:**

bash

mvn test

**Gradle:**

bash

gradle test

**IDE:**

Most IDEs, such as IntelliJ IDEA and Eclipse, support running JUnit 5 tests directly through their interface.

**Summary of Key JUnit 5 Annotations**

| **Annotation** | **Description** |
| --- | --- |
| @Test | Marks a method as a test method. |
| @BeforeEach | Runs before each test method. |
| @AfterEach | Runs after each test method. |
| @BeforeAll | Runs once before all test methods (static). |
| @AfterAll | Runs once after all test methods (static). |
| @ParameterizedTest | Marks a method as a parameterized test. |
| @ValueSource | Provides a list of values for parameterized tests. |
| @Tag | Assigns a tag to a test, used for grouping or filtering tests. |
| @EnabledIf | Enables tests conditionally. |
| @DisabledIf | Disables tests conditionally. |
| @TestInstance | Controls the lifecycle of test instances. |