JUnit 5
What is JUnit 5?
JUnit 5 is a modern testing framework for Java that allows developers to write automated unit tests to verify their code. It is the successor to JUnit 4 and introduces several improvements, including better modularity, enhanced extensibility, and support for Java 8 and above.
Layman Explanation
Imagine you're baking a cake. Before serving it, you taste a small piece to check if it's sweet enough. JUnit 5 is like that taste test—it helps developers check if their code works correctly before releasing it.

What JUnit 5 Does
JUnit 5 enables developers to:
- Write automated tests to check if their code works.
- Run tests efficiently and get instant feedback.
- Organize tests better using annotations.
- Integrate with build tools like Maven and Gradle.
Layman Explanation
Think of JUnit 5 as a quality control system for software. Just like factories test products before selling them, developers use JUnit 5 to test their code before launching an application.

Why JUnit 5 is Helpful
- Improved Modularity: JUnit 5 is divided into multiple modules, making it more flexible.
- Better Extensibility: Allows custom extensions for test execution.
- Enhanced Assertions and Assumptions: Provides more expressive assertions for better test validation.
- Support for Java 8 and Above: Uses lambda expressions and other modern Java features.
Layman Explanation
JUnit 5 is like a smart assistant that helps developers catch mistakes early. It makes testing easier, faster, and more organized.

Key Annotations in JUnit 5
JUnit 5 introduces several annotations to simplify test writing:
Annotation	Professional Definition	Layman Explanation
@Test	Marks a method as a test case.	Think of it as a checkpoint to verify if a function works.
@BeforeEach	Runs before each test method.	Like preparing ingredients before cooking.
@AfterEach	Runs after each test method.	Like cleaning up after cooking.
@BeforeAll	Runs once before all tests in a class.	Like setting up a kitchen before cooking multiple dishes.
@AfterAll	Runs once after all tests in a class.	Like closing the kitchen after all meals are cooked.
@DisplayName	Provides a custom name for test methods.	Like labeling a dish with a fancy name.
@Nested	Allows nested test classes.	Like grouping related tests together.
@Tag	Enables test filtering.	Like categorizing recipes (e.g., "Desserts" vs. "Main Course").
@ExtendWith	Registers custom extensions.	Like adding extra tools to a kitchen for better cooking.

Example Code for Each Annotation
Each example includes:
- Real Code (A simple Java class)
- Testing Code (JUnit 5 test case)
- Explanation (Step-by-step breakdown)

Example 1: @Test
Real Code

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Testing Code

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class CalculatorTest {
    @Test
    void additionTest() {
        // This test checks if 2 + 3 equals 5
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result, "Addition should be correct");
    }
}
```

Explanation
- The @Test annotation tells JUnit this is a test case.
- The test creates a Calculator object and calls add(2, 3).
- assertEquals(5, result) checks if the result is correct.
- If the test passes, it means the add() method works properly.

Example 2: @BeforeEach and @AfterEach
Real Code

```
public class Database {
    public void connect() {
        System.out.println("Connecting to database...");
    }

    public void disconnect() {
        System.out.println("Disconnecting from database...");
    }
}
```

Testing Code

```
import org.junit.jupiter.api.*;

class DatabaseTest {
    Database db;

    @BeforeEach
    void setup() {
        // Runs before each test
        db = new Database();
        db.connect();
    }

    @Test
    void sampleTest() {
        System.out.println("Executing test...");
    }

    @AfterEach
    void teardown() {
        // Runs after each test
        db.disconnect();
    }
}
```

Explanation
- @BeforeEach sets up the database connection before each test.
- @AfterEach closes the connection after each test.
- This ensures each test starts fresh without leftover data.

Example 3: @BeforeAll and @AfterAll
Real Code

```
public class Server {
    public static void start() {
        System.out.println("Starting server...");
    }

    public static void stop() {
        System.out.println("Stopping server...");
    }
}
```

Testing code

```
import org.junit.jupiter.api.*;

class ServerTest {
    @BeforeAll
    static void init() {
        // Runs once before all tests
        Server.start();
    }

    @Test
    void testMethod() {
        System.out.println("Running test...");
    }

    @AfterAll
    static void cleanup() {
        // Runs once after all tests
        Server.stop();
    }
}
```

Explanation
- @BeforeAll starts the server before running tests.
- @AfterAll stops the server after all tests are done.
- This ensures efficient resource management

Example 4: @DisplayName
Real Code

```
public class User {
    public String getName() {
        return "John Doe";
    }
}
```

Testing code

```
import org.junit.jupiter.api.*;

class UserTest {
    @Test
```

```
    @DisplayName("Custom Test Name")
    void customTest() {
        User user = new User();
        System.out.println("User name: " + user.getName());
    }
}
```

Explanation
- @DisplayName gives a custom name to the test.
- Instead of showing customTest(), the test report will display "Custom Test Name".