# JUnit Framework & Automation Testing – Study Notes

## 1. Introduction to JUnit Framework

* **JUnit** is a popular **unit testing framework for Java**.
* It helps developers write and run automated tests to ensure correctness of code.
* JUnit supports **Test-Driven Development (TDD)** and plays a key role in **automation testing**.

## 2. Key Features of JUnit

1. Open-source and lightweight.
2. Annotation-driven (@Test, @BeforeEach, etc.).
3. Provides **assertions** to validate test results.
4. Supports **test suites** to group tests.
5. Exception testing using assertThrows().
6. Timeout testing with @Timeout.
7. Parameterized tests for multiple inputs.
8. Integrates with IDEs (Eclipse, IntelliJ) and build tools (Maven, Gradle).
9. Each test runs independently.
10. Works with **mocking frameworks** (e.g., Mockito).
11. Compatible with CI/CD tools (Jenkins, GitHub Actions).
12. Supports **Hamcrest matchers** for readable assertions.
13. Cross-platform (runs anywhere Java runs).

## 3. Why JUnit?

* Ensures **code quality** by identifying bugs early.
* Encourages **TDD** → write tests before implementation.
* Provides **automation** of tests → reduces manual effort.
* Supports **regression testing** → prevents breaking old functionality.
* Provides **fast feedback** on code changes.
* Integrates with tools and pipelines.
* Structured and **readable test cases**.
* **Cost-effective** → fixing bugs early is cheaper.
* Open-source, widely used, strong community support.

## 4. Limitations (Cons) of JUnit

1. Restricted to **unit testing**, not full system/UI testing.
2. Works only for **Java applications**.
3. Requires **coding knowledge** → not tester-friendly.
4. Complex scenarios (multi-threading, GUIs) are harder to test.
5. Beginners may find setup confusing.
6. Provides only **basic reporting**, needs external tools for detailed reports.
7. Limited support for **parallel test execution**.
8. Maintenance becomes difficult in **large projects**.
9. Older versions had weak support for **data-driven testing**.

## 5. JUnit Basics

### 5.1 Unit Test Case in JUnit

* A **test case** is a method that tests a single functionality of code.
* Example:

@Test

void testAdd() {

assertEquals(5, new Calculator().add(2, 3));

}

### 5.2 Test Methods

* Declared with @Test.
* Should test **one functionality at a time**.

### 5.3 Assertions

* Validate expected results:
  + assertEquals(expected, actual)
  + assertTrue(condition)
  + assertFalse(condition)
  + assertThrows(Exception.class, () -> { ... })
  + assertNotNull(object)

### 5.4 Annotations

* @Test → Marks a test method.
* @BeforeEach → Runs before each test.
* @AfterEach → Runs after each test.
* @BeforeAll → Runs once before all tests.
* @AfterAll → Runs once after all tests.
* @Disabled → Skips a test.

## 6. Hands-On with Eclipse

1. Create a **Java Project**.
2. Add **JUnit 5 library**.
3. Create the class to test (Calculator).
4. Create a **test class** (CalculatorTest).
5. Write @Test methods.
6. Run → Right-click → **Run As → JUnit Test**.

## 7. Advanced JUnit Concepts

### 7.1 Test Suite

* Group multiple test classes:

@RunWith(Suite.class)

@Suite.SuiteClasses({

CalculatorTest.class,

AnotherTest.class

})

public class TestSuiteExample {}

### 7.2 Exception Testing

@Test

void testDivideByZero() {

assertThrows(ArithmeticException.class, () -> calculator.divide(10, 0));

}

### 7.3 Test Timeouts

@Test

@Timeout(1) // seconds

void testTimeout() throws InterruptedException {

Thread.sleep(2000); // fails

}

### 7.4 Parameterized Tests

@ParameterizedTest

@ValueSource(ints = {1, 2, 3, 4})

void testIsPositive(int number) {

assertTrue(number > 0);

}

### 7.5 Hamcrest Library

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.Matchers.\*;

@Test

void testHamcrest() {

assertThat("JUnit", containsString("Unit"));

assertThat(5, greaterThan(3));

}

### 7.6 Mocking (Mockito)

@Mock

List<String> mockList;

@Test

void testMocking() {

mockList.add("Hello");

verify(mockList).add("Hello");

}

## 8. Automation Testing Overview

### 8.1 What is Automation Testing?

* Using **tools and scripts** to execute tests automatically.

### 8.2 Why Automation Testing?

* Faster, accurate, cost-effective, supports regression and CI/CD.

### 8.3 Manual vs Automated Testing

* **Manual**: Tester executes steps manually.
* **Automated**: Scripts/tools execute steps automatically.

### 8.4 Automation Tools & Frameworks

* **JUnit, TestNG** → Unit testing.
* **Selenium** → Web UI testing.
* **Cucumber** → BDD testing.
* **Appium** → Mobile automation.
* **RestAssured** → API testing.

## 9. Test Automation Life Cycle

1. Planning
2. Design
3. Development (script writing)
4. Execution
5. Reporting
6. Maintenance

## 10. Automation Frameworks

* **Data-Driven** → External data sources (Excel, DB).
* **Keyword-Driven** → Keywords define test actions.
* **Hybrid** → Mix of data + keyword driven.
* **Page Object Model (POM)** → Used in Selenium, separates UI and logic.

## 11. Identifying Test Scenarios for Automation

* Repetitive test cases.
* Regression suites.
* High-risk functionalities.
* Data-driven scenarios.
* Multi-input test cases.

# JUnit API (JUnit 5 – Jupiter API)

## 🔹 Core Packages

* org.junit.jupiter.api → Core annotations and assertions.
* org.junit.jupiter.params → Parameterized tests.
* org.hamcrest → For Hamcrest matchers (optional).

## 🔹 Main Annotations in JUnit API

| **Annotation** | **Purpose** |
| --- | --- |
| @Test | Marks a method as a test case |
| @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) |
| @Disabled | Disables a test method/class |
| @RepeatedTest(n) | Repeats a test multiple times |
| @ParameterizedTest | Runs test with different input values |
| @Tag | Tags tests for filtering |
| @DisplayName | Gives test a custom readable name |
| @Nested | Defines nested test classes |

## 🔹 Assertions API (org.junit.jupiter.api.Assertions)

Commonly used static methods:

* assertEquals(expected, actual)
* assertNotEquals(unexpected, actual)
* assertTrue(condition)
* assertFalse(condition)
* assertNull(object)
* assertNotNull(object)
* assertArrayEquals(expectedArray, actualArray)
* assertThrows(Exception.class, () -> { ... })
* assertAll("heading", () -> { ... }, () -> { ... })

## 🔹 Assumptions API (org.junit.jupiter.api.Assumptions)

Used to skip tests when assumptions fail:

* assumeTrue(condition)
* assumeFalse(condition)
* assumingThat(condition, () -> { ... })

## 🔹 Parameterized Tests API (org.junit.jupiter.params)

* @ValueSource → Passes simple values (int, String, etc.).
* @CsvSource → Passes multiple values in CSV format.
* @CsvFileSource → Reads test data from CSV file.
* @MethodSource → Uses method as data provider.
* @EnumSource → Provides enum values.

Example:

@ParameterizedTest

@ValueSource(ints = {1, 2, 3})

void testPositiveNumbers(int number) {

assertTrue(number > 0);

}

## 🔹 Hamcrest Matchers (Optional API)

From org.hamcrest:

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.Matchers.\*;

@Test

void testHamcrestMatchers() {

assertThat("JUnit", containsString("Unit"));

assertThat(5, greaterThan(3));

}

## 🔹 JUnit Platform API

* Provides the **foundation** for running tests.
* Allows integration with IDEs and build tools (Maven, Gradle).

Key classes:

* Launcher → To launch tests.
* LauncherDiscoveryRequest → To select tests to run.
* TestExecutionListener → To listen to test execution events.

## 🔹 Example Using JUnit API

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

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

class CalculatorTest {

Calculator calculator;

@BeforeEach

void setUp() {

calculator = new Calculator();

}

@Test

@DisplayName("Test Addition")

void testAdd() {

assertEquals(5, calculator.add(2, 3));

}

@Test

void testDivideByZero() {

assertThrows(ArithmeticException.class, () -> calculator.divide(10, 0));

}

@AfterEach

void tearDown() {

calculator = null;

}

}

# Why Do We Use Assertions in JUnit?

An **assertion** is a statement that checks whether a test’s **actual result** matches the **expected result**.

If the assertion **passes**, the test continues.  
If it **fails**, JUnit marks the test as **failed**.

## Example Using Multiple Assertions

@Test

void testCalculatorAssertions() {

Calculator calc = new Calculator();

assertEquals(5, calc.add(2, 3));

assertNotEquals(10, calc.add(2, 3));

assertTrue(calc.divide(10, 2) == 5);

assertThrows(ArithmeticException.class, () -> calc.divide(10, 0));

String[] expected = {"A", "B"};

String[] actual = {"A", "B"};

assertArrayEquals(expected, actual);

assertAll("Multiple checks",

() -> assertNotNull(calc),

() -> assertFalse(2 > 5)

);

}