# Testing with JUnit in Java

*Fsoft Academy*

# Lesson Objectives

- **Understand** the basics of JUnit with the fundamental concepts of JUnit, such as test classes, test methods, assertions, and annotations.
- **Able to write** effective and comprehensive test cases to verify the behavior of your code.
- **Able to use** assertions provided by JUnit, such as **assertEquals**(), **assertTrue**(), **assertFalse**(), and more.
- Understand various annotations provided by JUnit, such as **@Test**, **@Before**, **@After**, **@BeforeEach**, **@AfterEach**, and **@Ignore**.
- **Understand** how to verify that specific exceptions are thrown when expected.
- **Understand** how test suites can help organize and manage your tests.

# Agenda

1. What is Unit Testing?
2. Setting up Junit
3. JUnit Test framework
4. JUnit Assert
5. JUnit TestSuite

Section 1

# What is Unit Testing?

# Unit testing

**Unit testing** is a software testing method where individual units or modules of code are tested to validate their correctness and functionality.

- The goal is to test components in isolation to catch issues early before integration.
- This allows developers to identify and fix defects efficiently in their own code.

# Unit Test – What and Who ?


- **Unit Testing Conductor**: Development team


- **Unit Testing Key Points:**
  - ✓ Validates individual units of code work properly
  - ✓ Unit is smallest testable part method, function, procedure, etc.
  - ✓ Tests units in isolation to catch issues before integration
  - ✓ Allows developers to test their own code modules
  - ✓ Enables early detection of defects


- **Unit Testing Deliverables:**
  - ✓ Tested software units
  - ✓ Related documents (Unit Test case, Unit Test Report)

# Unit Test – Why ?


- Ensure **quality of software unit**.


- Detect **defects** and **issues** early.

- Reduce the Quality Effort & Correction Cost.

Section 2

# Setting up JUnit

# What is JUnit?

- **Key points about JUnit:**
  - ✓ Open source unit testing framework for Java
  - ✓ Enables writing repeatable automated tests
  - ✓ Primarily used for unit testing to isolate code pieces
  - ✓ Integrates with IDEs and build tools like Ant
  - ✓ Easy to learn and use
  - ✓ Major versions are JUnit 3, 4, and 5

# Why you need JUnit testing ?

- It finds bugs early in the code, which makes our code more reliable.
- JUnit is useful for developers, who work in a test-driven environment.
- Unit testing forces a developer to read code more than writing.
- You develop more readable, reliable and bug-free code which builds confidence during development.

# Features and Advantages of JUnit4

- All the old assert statements are same as before.
- JUnit 4 can be used with Java 5 or higher version.
- While using JUnit4, you are not required to extend `junit.framework.TestCase`.
- You can just create a simple Java class.
- You need to use annotations in spite of special method name as before.
- Most of the things are easier in JUnit4 as:
  - ✓ With JUnit 4 you are more capable of identifying exception.
  - ✓ Parameterized test is introduced, which enables us to use parameters.
  - ✓ JUnit4 still can execute JUnit3 tests.

# What is JUnit 5?

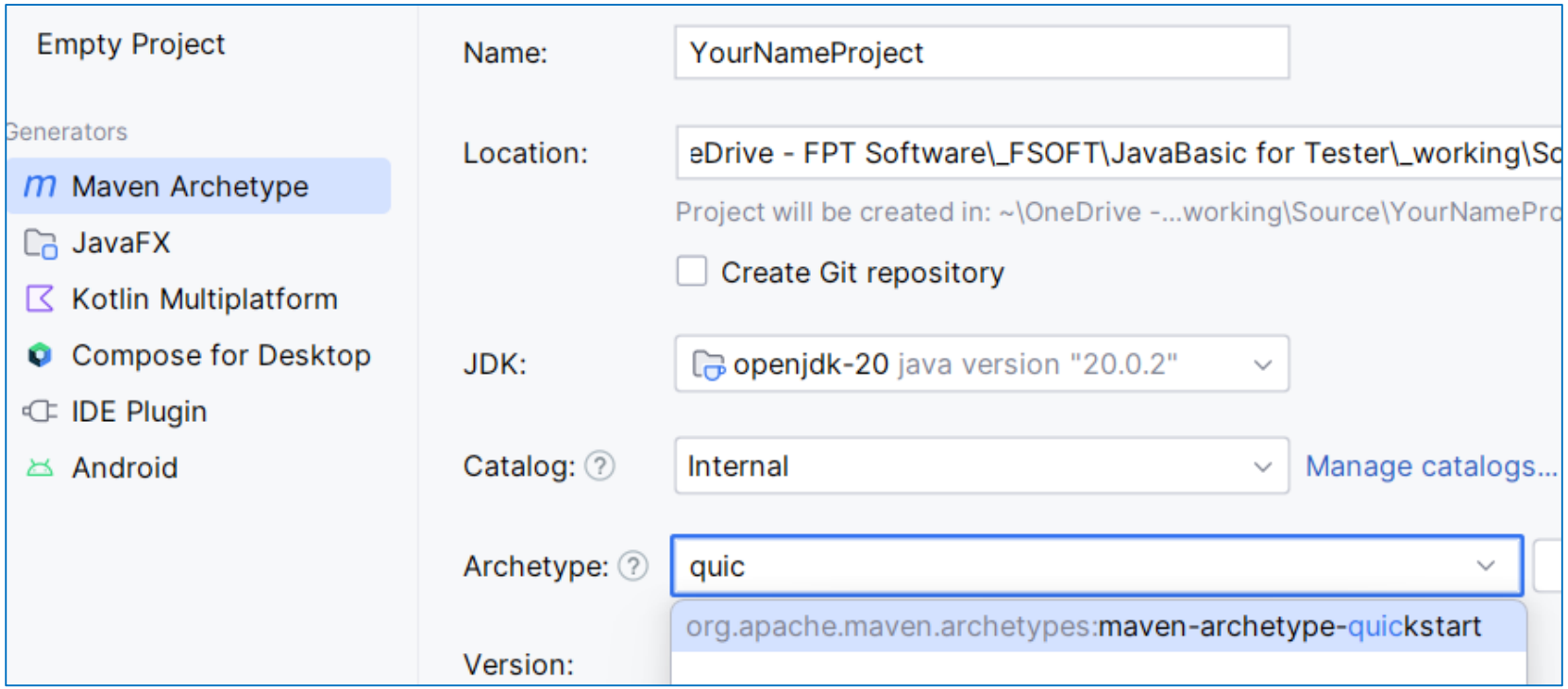
- **JUnit 5 Key Points:**
  - ✓ **Composed of JUnit Platform, Jupiter, and Vintage**
    - JUnit Platform foundation to launch testing frameworks
    - JUnit Jupiter new programming model and extensions
    - JUnit Vintage runs JUnit 3 and 4 tests
  - ✓ **Jupiter** has new annotations like @**BeforeEach**, @**AfterEach**
  - ✓ **Supports parameterized tests**
  - ✓ **Better exception handling**
  - ✓ **Runs natively on JUnit Platform**
  - ✓ **Integrated with IDEs like IntelliJ, Eclipse**
  - ✓ **JUnit 5 = Next generation of JUnit testing framework**

# Installation JUnit 5

- **Here are key points on setting up JUnit 5:**
  - ✓ Requires Java 8 or higher at runtime
  - ✓ Can test code compiled with older JDKs
  - ✓ Add JUnit 5 library to project build path
  - ✓ Annotate tests with JUnit 5 annotations
  - ✓ Run/debug JUnit tests through IDE

# Installation JUnit 5

- **Create Maven project**



# Installation JUnit 5

- **Update pom.xml**

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.10.0</version>
  <scope>compile</scope>
</dependency>

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.10.0</version>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-suite-engine</artifactId>
  <version>1.8.1</version>
</dependency>
```

# JUnit Annotations

- **Common JUnit 5 Annotations:**
  - ✓ **@Test**: Denotes test method
  - ✓ **@DisplayName**: Custom display name for test
  - ✓ **@BeforeEach**: Run before each test
  - ✓ **@AfterEach**: Run after each test
  - ✓ **@BeforeAll**: Run once before all tests
  - ✓ **@AfterAll**: Run once after all tests
  - ✓ **@Disabled**: Disable a test
  - ✓ **@Nested**: Nested non static test class
  - ✓ **@Tag**: Tag tests for filtering
  - ✓ **@ExtendWith**: Register custom extensions
- **JUnit 5 annotations help write and organize tests.**

Section 3

# JUnit Assertion

# Assertion Overview

- JUnit assertions validate expected vs actual results.

## Common JUnit 5 Assertions

- **assertEquals**()/assertNotEquals(): equality checks
- **assertArrayEquals**(): check array equality
- **assertIterableEquals**(): check Iterable equality
- **assertLinesMatch**(): check line by line content
- **assertNotNull**()/assertNull(): check null
- **assertNotSame**()/assertSame(): check object references

## Common JUnit 5 Assertions

- **assertTimeout**()/assertTimeoutPreemptively(): check execution time
- **assertTrue**()/assertFalse(): boolean checks
- **assertThrows**(): check if exception is thrown
- **fail**(): always fail

# JUnit Assertion methods

- **Boolean:**
  - ✓ **assertTrue**(condition) asserts condition is true
  - ✓ **assertFalse**(condition) asserts condition is false
- **Null object:**
  - ✓ **assertNull**(object) asserts object is null
  - ✓ **assertNotNull**(object) asserts object is not null
- **Identity:**
  - ✓ **assertSame**(expected, actual) objects refer to same instance
  - ✓ **assertNotSame**(expected, actual) objects refer to different instances
- **Equality:**
  - ✓ **assertEquals**(expected, actual) values are equal via equals() method

# JUnit Assertion methods

- **Arrays:**
  - ✓ **assertArrayEquals**(expected, actual) arrays contain same elements
- **Failures:**
  - ✓ **fail**(message): always fails with given message
  - ✓ **fail**(): always fails with default message
- **Custom failure messages:**
  - ✓ Most assertions support an optional first parameter with a custom failure message, e.g:
    - **assertEquals**(message, expected, actual)

# How assertEquals() works?

- **assertEquals** method is used to compare two values or objects to check if they are equal.

```
public class MyJUnitTest {
    @Test
    public void testEqual1() {
        String obj1 = "Junit", obj2 = "Junit";
        assertEquals(obj1, obj2); // true
    }
    @Test
    public void testEqual2() {
        Integer a = 5;
        Integer b = 5;
        assertEquals(a, b); // true
            }
    public void testEqual3() {
        int a = 5;
        int b = 5;
        assertEquals(a, b); // true
    }
}
```

# Floating Point Assertions

- How **assertEquals**() works for floats/doubles in JUnit:
  - ✓ For floating point types like float and double, **assertEquals**() takes an additional delta parameter.
  - ✓ This is because of round-off errors in floating point math.
- **The assertion checks:**
  - ✓ Math.**abs**(expected  actual) <= delta
- **Example:**
  - ✓ double a = 1.23456, b = 1.23459;
  - ✓ assertEquals(a, b, 0.001); // would pass as difference is less than 0.001 delta
  - ✓ assertEquals(a, b, 0.00001); // would fail as difference is more than 0.00001 delta
- So for floats/doubles, **assertEquals**() takes a delta tolerance to avoid round-off errors.

# Example

```java
public class MathOperations {
    public static double add(double num1, double num2) {
        return num1 + num2;
    }

    public static double multiply(double num1, double num2, double num3) {
        return num1 * num2 * num3;
    }
}
```

```java
public class MathOperationsTest {
    @Test
    public void testAdd() {
        double result = MathOperations.add(5.1, 7.3);
        assertEquals(12.4, result, 0.0001);
         // Here, 0.0001 is the delta for comparison
    }
    @Test
    public void testMultiply() {
        double result = MathOperations.multiply(2.0, 3.0, 4.0);
        assertEquals(24.0, result, 0.0001);
         // Delta is 0.0001 for floating-point comparison
    }
}
```

# How assertIterableEquals() works?

- **assertIterableEquals** checks that two Iterables are deeply equal
- **Deep equality means:**
  - ✓ The number and order of elements must match
  - ✓ Each element must be equal according to equals()
- **Overloaded methods:**
  - ✓ public static void **assertIterableEquals**(Iterable<?> expected, Iterable> actual)
  - ✓ public static void **assertIterableEquals**(Iterable<?> expected, Iterable> actual, String message)
  - ✓ public static void **assertIterableEquals**(Iterable<?> expected, Iterable> actual, Supplier<String> messageSupplier)
- *Useful for comparing Lists, Sets, Queues, and other collections*
- *Fails if number of elements or any element differs*

# Example

```
@Test
public void testListEquality2() {
    List<Integer> list1 = Arrays.asList(1, 2, 3);
    List<Integer> list2 = Arrays.asList(1, 2, 3);
    assertIterableEquals(list1, list2);
}

@Test
public void testCase() {
    Iterable<Integer> listOne = new ArrayList<>(Arrays.asList(1, 2, 3, 4));
    Iterable<Integer> listTwo = new ArrayList<>(Arrays.asList(1, 2, 3, 4));
    Iterable<Integer> listThree = new ArrayList<>(Arrays.asList(1, 2, 3));
    Iterable<Integer> listFour = new ArrayList<>(Arrays.asList(1, 2, 4, 3));

    // Test will pass
    Assertions.assertIterableEquals(listOne, listTwo);

    // Test will fail
    Assertions.assertIterableEquals(listOne, listThree);

    // Test will fail
    Assertions.assertIterableEquals(listOne, listFour);
}
```

# Practical Time

- Create a simple test class named **AssertionTest.java.**
- Create few variables and important assert statements in JUnit.

```java
// Test integers
int expectedInt = 42;
int actualInt = 42;
assertEquals(expectedInt, actualInt);

// Test strings
String expectedStr = "Hello";
String actualStr = "Hello";
assertEquals(expectedStr, actualStr);

// Test arrays
int[] expectedArray = {1, 2, 3};
int[] actualArray = {1, 2, 3};
assertArrayEquals(expectedArray, actualArray);
```

# Practical Time

```
// Test boolean
boolean condition = true;
assertTrue(condition);

// Test null
Object nullObject = null;
assertNull(nullObject);

// Test not null
Object notNullObject = new Object();
assertNotNull(notNullObject);

// Test equality
assertSame(expectedInt, actualInt);

// Test inequality
assertNotSame(expectedInt, actual: 24);
```

# Practical Time

```java
// Test true condition
assertTrue(condition);

// Test false condition
assertFalse(!condition);

// Test for exception
assertThrows(ArithmeticException.class, new Executable() {
    @Override
    public void execute() throws Throwable {
        int result = 1 / 0;
    }
});
```

Section 4

# JUnit 5 Test LifeCycle

# How JUnit 5 test lifecycle works?

- **Four main annotations drive test lifecycle:**
  - ✓ @**BeforeAll**: Run once before all tests
  - ✓ @**BeforeEach**: Run before each test
  - ✓ @**AfterEach**: Run after each test
  - ✓ @**AfterAll**: Run once after all tests
- @Test: denotes a test method
- @BeforeAll / @AfterAll: must be static as run only once
- @BeforeEach / @After: Each run per test instance

# Writing Test

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

```java
public class AppTest {

    @BeforeAll
    static void setup() {
        System.out.println("@BeforeAll executed");
    }

    @BeforeEach
    void setupThis() {
        System.out.println("@BeforeEach executed");
    }

    @Test
    void testCalcOne() {
        System.out.println("======TEST ONE EXECUTED=======");
        Assertions.assertEquals(4, Calculator.add(2, 2));
    }
```

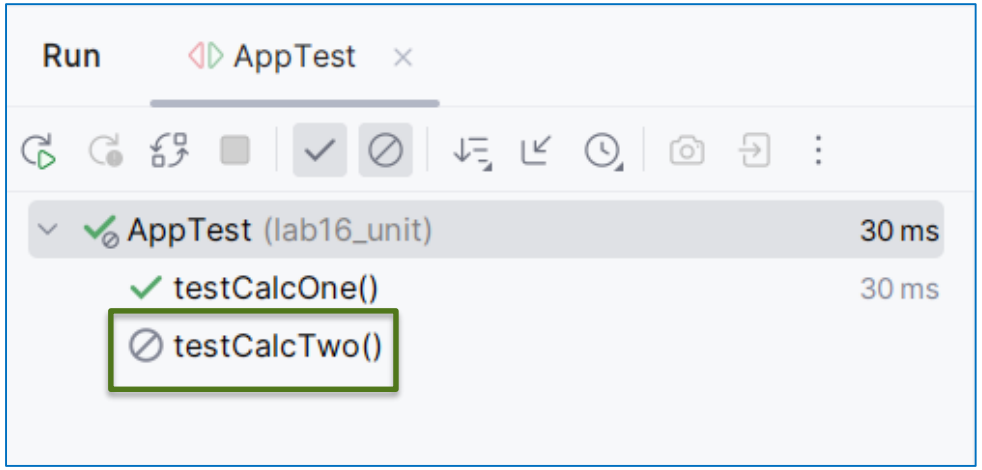
```java
    @Disabled
    @Test
    void testCalcTwo() {
        System.out.println("======TEST TWO EXECUTED=======");
        Assertions.assertEquals(6,
                Calculator.add(2, 4));
    }

    @AfterEach
    void tearThis() {
        System.out.println("@AfterEach executed");
    }

    @AfterAll
    static void tear() {
        System.out.println("@AfterAll executed");
    }
}
```

# Writing Tests in JUnit 5

- **Result Execute:**




- **Console Output:**

```
@BeforeAll executed
@BeforeEach executed
======TEST ONE EXECUTED=======
@AfterEach executed

void lab16_unit.AppTest.testCalcTwo() is @Disabled
@AfterAll executed
```

# Disabling Test

- Use @**Disabled** annotation to disable tests
- Can disable all tests in a class or disable individual methods
- Disabled tests are skipped when running tests
- So @**Disabled** allows flexibly skipping tests at both class and method level.

## ✓Disable all tests

```
@Disabled
public class DisabledTestClass {
    void testMethod1() {
        System.out.println("This test won't run.");
    }
    void testMethod2() {
        System.out.println("This test won't run.");
    }
}
```

## ✓ Disable individual methods

```
@Disabled
void testMethod1() {
    System.out.println("This test won't run.");
}

@Disabled
void testMethod2() {
    System.out.println("This test won't run.");
}
```

# @BeforeEach and @AfterEach Annotations

- **@BeforeEach** runs before each **@Test** method
- **@AfterEach** runs after each **@Test** method
- Used for repeated setup and teardown logic around tests
- Must not be static, run for each test instance
- Allows reusable setup/cleanup of the test environment

```java
@BeforeEach
void setup() {
    // initialize test data
}

@Test
void testMethod1() {
    // perform test
}

@Test
void testMethod2() {
    // perform test
}
```

```java
@AfterEach
void cleanup() {
    // clean up test data
}
```

# @BeforeEach and @AfterEach Annotations

- **Example:**

```java
@BeforeAll
public static void init() {
    System.out.println("+ BeforeAll init() method called");
}

@BeforeEach
public void initEach() {
    System.out.println("BeforeEach initEach() method called");
}

@AfterEach
public void cleanUpEach() {
    System.out.println("After Each cleanUpEach() method called");
}

@AfterAll
public static void cleanUp() {
    System.out.println("+ After All cleanUp() method called");
}
```

```java
@Test
void addNumber1() {
    System.out.println("Running test > 1");
    Assertions.assertEquals(2, Calculator.add(1, 1), "1 + 1 should equal 2");
}

@Test
void addNumber2() {
    System.out.println("Running test > 2");
    Assertions.assertEquals(2, Calculator.add(1, 1), "1 + 1 should equal 2");
}

@Test
void addNumber3() {
    System.out.println("Running test > 3");
    Assertions.assertEquals(2, Calculator.add(1, 1), "1 + 1 should equal 2");
}
```

# @BeforeEach and @AfterEach annotations

- **Result:**

Run: org.example in demoJUnit5

Tests passed: 3 of 3 tests – 17 ms

| | |
|---|---|
| example (org) | 17 ms |
| BeforeAndAfter | 17 ms |
| addNumber1() | 14 ms |
| addNumber2() | 1 ms |
| addNumber3() | 2 ms |

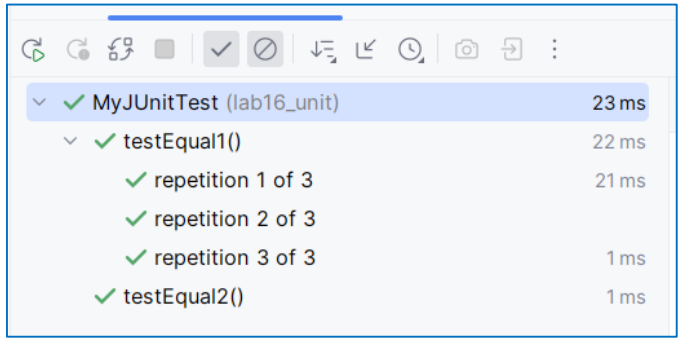
```
C:\Users\Asus\.jdks\corretto-19.0.2\bin\java.exe ...
+ BeforeAll init() method called
BeforeEach initEach() method called
Running test > 1
After Each cleanUpEach() method called
BeforeEach initEach() method called
Running test >2
After Each cleanUpEach() method called
BeforeEach initEach() method called
Running test > 3
After Each cleanUpEach() method called
+ After All cleanup() method called
```

# @RepeatedTest Annotation

- **@RepeatedTest** allows repeating test execution
- Specify repetition count: @RepeatedTest(3) // run test 3 times

```java
public class MyJUnitTest {
    @RepeatedTest(3)
    public void testEqual1() {
        String obj1 = "Junit", obj2 = "Junit";
        assertEquals(obj1, obj2); // true, as obj1.equals(ob
    }

    @Test
    public void testEqual2() {
        Integer a = 5;
        Integer b = 5;
        assertEquals(a, b); // true, int converted to Intege
    }
```



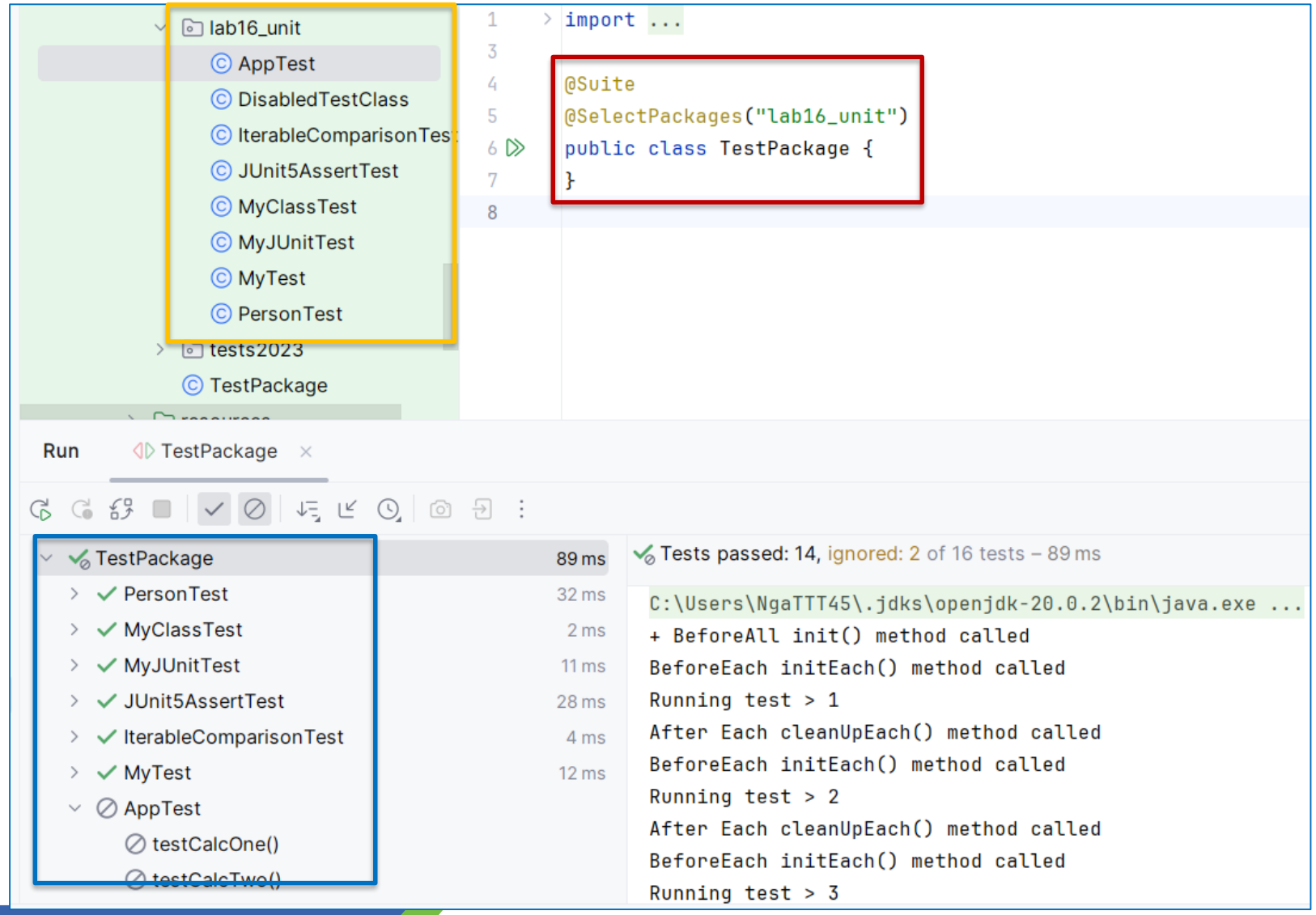
Section 5

# Junit 5 Test Suite

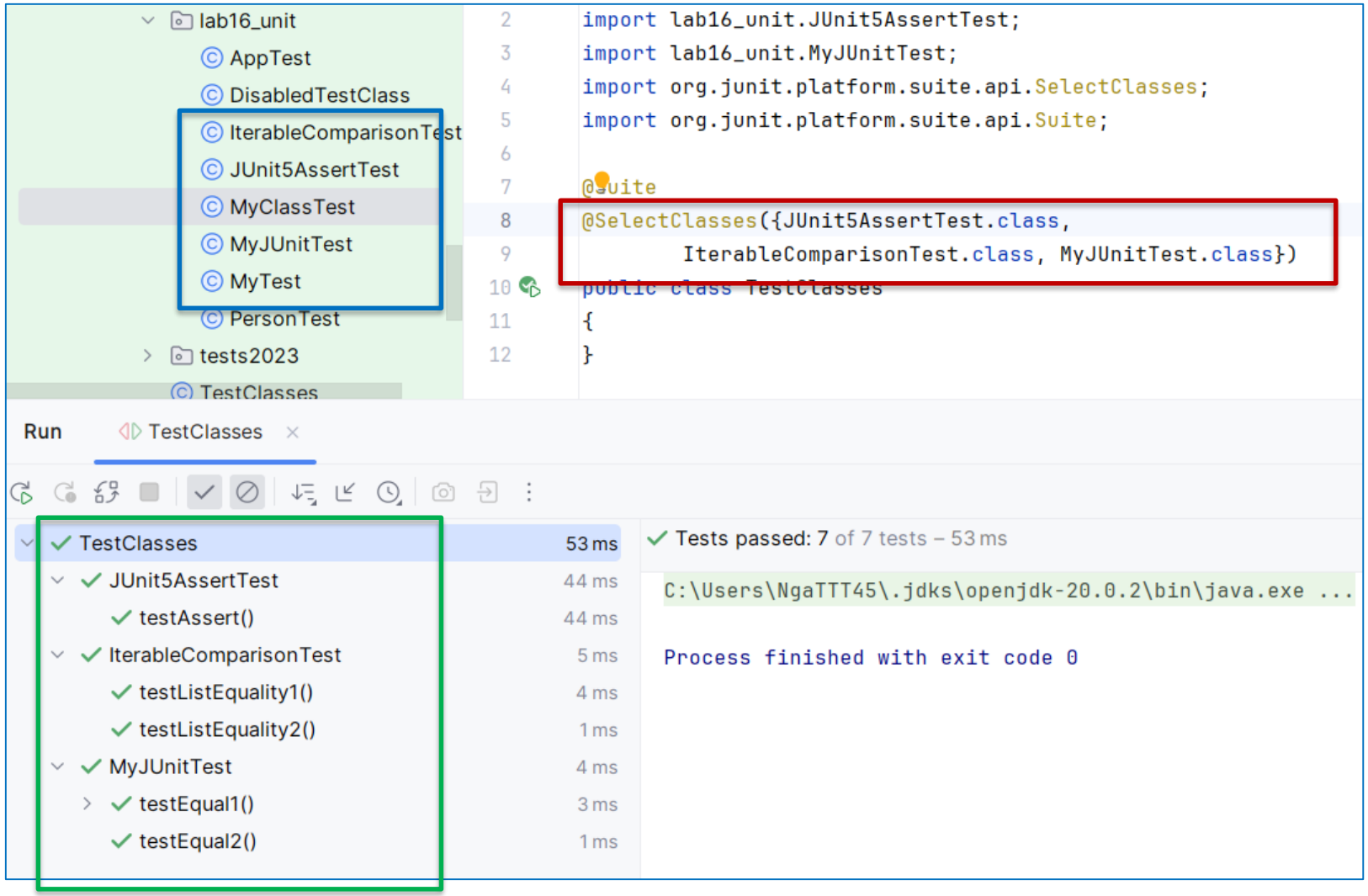
# Test Suite Overview

- **Creating test suites in JUnit 5:**
  - ✓ JUnit 5 suites run tests from multiple classes/packages
  - ✓ Annotate suite class with:
    - **@SelectPackages**("com.app.tests")
    - **@SelectClasses**({TestClass1.class, TestClass2.class})
  - ✓ **@SelectPackages** scans sub-packages too
- **Suite class runs as a normal JUnit test**
- **Useful for organizing and running groups of tests together**

# Test Suite Example



# Test Suite Example



# Test Result

- A test method in a test case can have one of three results:
  - ✓ **Pass** – all assertions matched expected values
  - ✓ **Failed** – an assertion did not match expected value
  - ✓ **Error** – an unexpected exception was thrown during execution of test method

# JUnit Exception Test

- **JUnit allows testing for exceptions in two main ways:**
  - ✓ Use @Test(expected=Exception.class) to expect an exception
  - ✓ Catch exception in try/catch and use fail() to check if unexpected
- This provides a clean and readable approach to verify expected exceptions vs unexpected ones.

# JUnit Exception Test

```java
public static int divide(int dividend, int divisor) {
    if (divisor == 0) {
        throw new ArithmeticException("Division by zero is not allowed.");
    }
    return dividend / divisor;
}
```

```java
public class TestException {
    @Test
    void testExceptionIsThrown() {
        // Assume you have a method that throws an IllegalArgumentException
        Executable executable = new Executable() {
            @Override
            public void execute() throws Throwable {
                MathOperations.divide(10, 0);
            }
        };

        assertThrows(ArithmeticException.class, executable);
    }
}
```

# Summary

- Introduction to Junit
- Setting up JUnit
- JUnit Test framework
- Junit Assert
- Junit TestSuite
- Examples

# Resources & References

- Resources
  - ✓ www.junit.org/
  - ✓ http://junit.sourceforge.net
- Recommended readings
  - ✓ Manning – Junit in Action
  - ✓ Test Driven Development: By Example. Boston: AddisonWesley, 2003

# THANK YOU!

*Any questions?*