Software Testing
Practical Work

SAFAR Fatima Ezzahra
UM6P

Contents

1 Warming Up: Your First Unit Test

1.1 Objective

This step-by-step practical guide helps you write and run your first JUnit 5 unit test in Java using IntelliJ IDEA. While following the steps you will:

- Create a simple Calculator class
- Write a unit test for the add (sum) method
- Learn key assertions and test lifecycle annotations
- Run the test in IntelliJ and interpret results

1.2 Prerequisites

- IntelliJ IDEA (Community or Ultimate) installed
- JDK 11 or newer configured in IntelliJ

1.3 Step 1 — Create Project

Open IntelliJ and create a new Java project with Maven as a build system.

Ensure you have this structure:

```
project-root/
+-- src/
    |-- main/
    |   +-- java/
    +-- test/
        +-- java/
```

1.4 Step 2 — Create the Calculator Class

Understanding the Calculator Class

The Calculator class serves as our **System Under Test (SUT)**. This simple class implements basic arithmetic operations, starting with addition. The class encapsulates the business logic that we want to verify through unit testing. By keeping the implementation straightforward, we can focus on learning the testing mechanics without getting distracted by complex logic.

1.4.1 Calculator.java

```java
package main;

public class Calculator {

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

1.5 Step 3 — Create the Test Class

Test Class Generation

IntelliJ IDEA provides powerful code generation features that automatically create test class stubs. This automated approach ensures proper project structure, correctly places test files in the `src/test/java` directory, and sets up the necessary JUnit 5 dependencies. The generated test class follows naming conventions (ClassName + Test) and includes the proper imports and annotations needed for testing.

1. Open `Calculator.java` in the editor
2. Right-click inside the editor → Generate (or press Alt+Insert) → Test...
3. In the dialog choose JUnit5 (Jupiter) and select the `add` method to generate a test stub
4. If IntelliJ asks to create a test directory, click Yes

1.6 Step 4 — Explore the Generated Test and Add Assertions

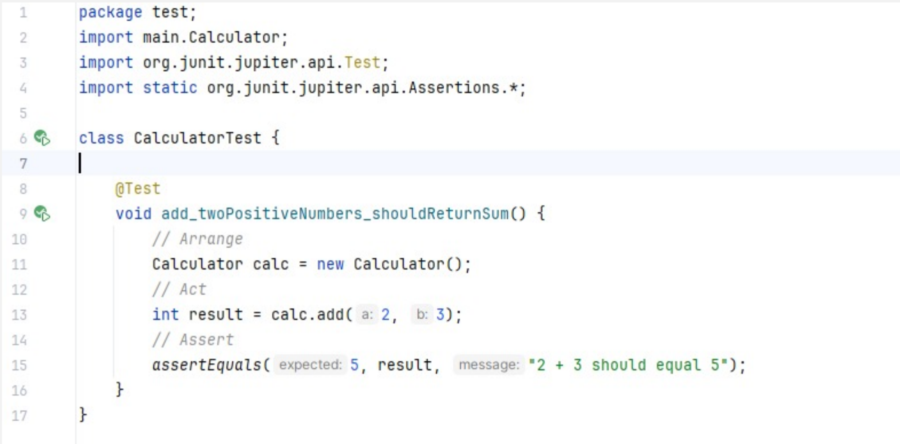
The AAA Pattern: Arrange-Act-Assert

This test follows the **Arrange-Act-Assert (AAA)** pattern, a fundamental best practice in unit testing:

- **Arrange**: Set up the test environment by creating objects and preparing input data
- **Act**: Execute the method under test with the prepared inputs
- **Assert**: Verify that the actual outcome matches the expected result

The test method name follows a descriptive convention: `methodName_scenario_expectedBehavior`, making it self-documenting and easy to understand test failures.

1.6.1 CalculatorTest.java

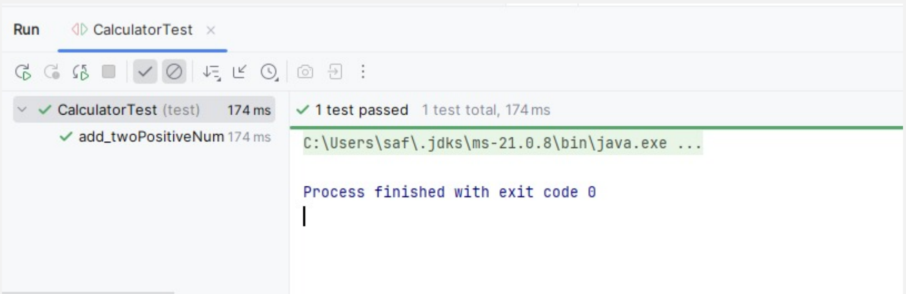


```java
package test;
import main.Calculator;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @Test
    void add_twoPositiveNumbers_shouldReturnSum() {
        // Arrange
        Calculator calc = new Calculator();
        // Act
        int result = calc.add(2, 3);
        // Assert
        assertEquals(5, result, "2 + 3 should equal 5");
    }
}
```

1.6.2 Output



1.7 Step 6 — Small Variations to Try

Learning from Failure

Intentionally breaking tests is an excellent learning technique. By modifying the implementation to return incorrect results, you observe how the test framework detects and reports failures. This exercise demonstrates the protective value of unit tests: they act as a safety net that catches regressions when code is modified incorrectly.

1.7.1 Calculator.java

```java
package main;

public class Calculator {

    public int add(int a, int b) {
        return a - b;
    }
}

```

1.7.2 CalculatorTest.java

```java
package test;
import main.Calculator;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @Test
    void add_twoPositiveNumbers_shouldReturnSum() {
        // Arrange
        Calculator calc = new Calculator();
        // Act
        int result = calc.add(2, 3);
        // Assert
        assertEquals(5, result, "2 + 3 should equal 5");
    }
}
```

1.7.3 Output

```
CalculatorTest (test) 232 ms
  add_twoPositiveNum 232 ms

1 test failed   1 test total, 232 ms
C:\Users\saf\.jdks\ms-21.0.8\bin\java.exe ...

org.opentest4j.AssertionFailedError: 2 + 3 should equal 5 ==>
Expected :5
Actual   :-1
<Click to see difference>
```

1.8 Step 7 — Extend the Calculator

Extended Calculator Implementation

The extended Calculator class now implements four fundamental arithmetic operations. Each method is designed to handle its specific operation, with the `divide` method including defensive programming through exception handling for division by zero. This comprehensive implementation provides multiple methods to practice writing diverse test cases, covering different scenarios including edge cases and error conditions.

1.8.1 Calculator.java

```java
package main;

public class Calculator {  3 usages

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

    public int subtract(int a, int b) {  no usages
        return a - b;
    }

    public int multiply(int a, int b) {  no usages
        return a * b;
    }

    public double divide(int a, int b) {  no usages
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return (double) a / b;
    }
}
```

1.8.2 Extended Calculator Tests

Comprehensive Test Coverage

These tests demonstrate comprehensive coverage of the Calculator's functionality. Each test is isolated and focuses on a single method, following the **Single Responsibility Principle** of testing. The tests cover:

- **Happy path scenarios**: Normal operations with typical inputs
- **Boundary conditions**: Testing with zero, negative numbers
- **Precision handling**: Using epsilon for floating-point comparisons in division tests

Notice how the divide test uses a delta value (0.001) when comparing doubles, accounting for floating-point arithmetic imprecision.

1.8.3 CalculatorTest.java

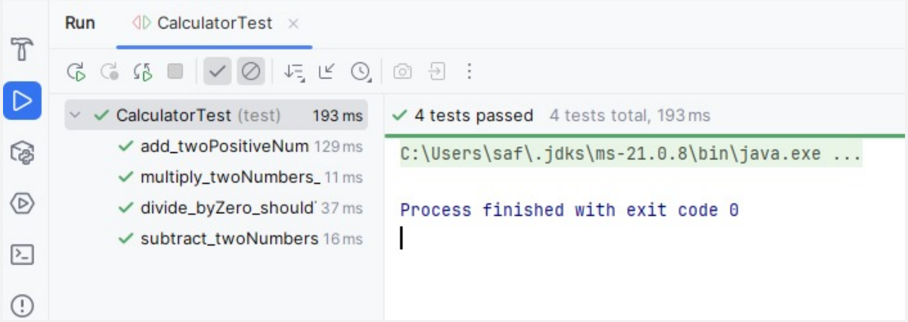
```java
package test;
import main.Calculator;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
class CalculatorTest {
    @Test
    void add_twoPositiveNumbers_shouldReturnSum() {
        // Arrange
        Calculator calc = new Calculator();
        // Act
        int result = calc.add(2, 3);
        // Assert
        assertEquals(5, result, "2 + 3 should equal 5");
    }
    @Test
    void subtract_twoNumbers_shouldReturnDifference() {
        // Arrange
        Calculator calc = new Calculator();
        // Act
        int result = calc.subtract(5, 3);
        // Assert
        assertEquals(2, result, "5 - 3 should equal 2");
    }
    @Test
    void multiply_twoNumbers_shouldReturnProduct() {
        // Arrange
        Calculator calc = new Calculator();
        // Act
        int result = calc.multiply(4, 3);
        // Assert
```

1.8.4 CalculatorTest.java

```java
    @Test
    void multiply_twoNumbers_shouldReturnProduct() {
        // Arrange
        Calculator calc = new Calculator();
        // Act
        int result = calc.multiply(4, 3);
        // Assert
        assertEquals(12, result, "4 * 3 should equal 12");
    }

    @Test
    void divide_twoNumbers_shouldReturnQuotient() {
        // Arrange
        Calculator calc = new Calculator();
        // Act
        double result = calc.divide(6, 2);
        // Assert
        assertEquals(3.0, result, 0.001, "6 / 2 should equal 3.0");
    }
}
```

1.8.5 Output



1.9 Step 8 — Exception Handling (Divide by Zero)

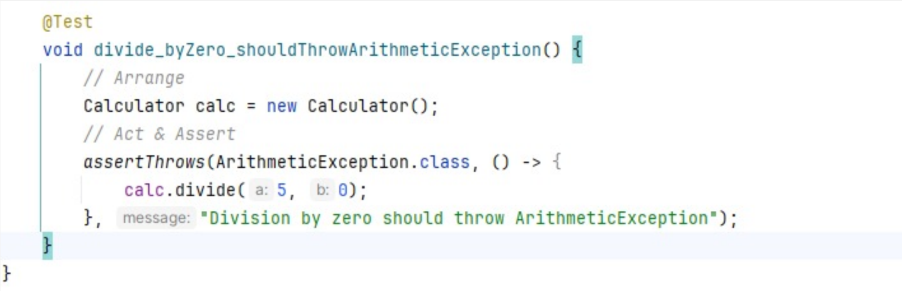
Testing Exception Behavior

Exception testing is a critical aspect of unit testing. The `assertThrows` assertion verifies that the code correctly handles error conditions by throwing appropriate exceptions. This test ensures that:

- The divide method validates its inputs
- An `ArithmeticException` is thrown when attempting division by zero
- The exception message is meaningful for debugging

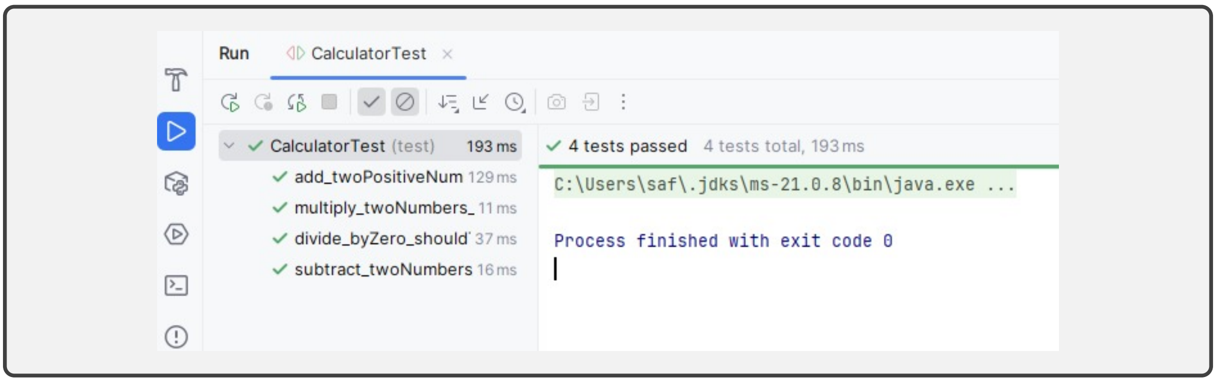
This defensive programming approach prevents undefined behavior and makes the API contract explicit: dividing by zero is not allowed and will result in an exception.

1.9.1 CalucaltorTest.java



```
    @Test
    void divide_byZero_shouldThrowArithmeticException() {
        // Arrange
        Calculator calc = new Calculator();
        // Act & Assert
        assertThrows(ArithmeticException.class, () -> {
            calc.divide( a: 5,  b: 0);
        },  message: "Division by zero should throw ArithmeticException");
    }
}
```

1.9.2 Output


Run
CalculatorTest
CalculatorTest (test) 193 ms
add_twoPositiveNum 129 ms
multiply_twoNumbers_ 11 ms
divide_byZero_should' 37 ms
subtract_twoNumbers 16 ms
4 tests passed 4 tests total, 193 ms
C:\Users\saf\.jdks\ms-21.0.8\bin\java.exe ...
Process finished with exit code 0

2 Exercise 2: Temperature Regulator

2.1 Context

Industrial Control System Testing

Temperature regulation systems are common in industrial settings, HVAC systems, and scientific equipment. This exercise simulates a real-world scenario where precision matters: the regulator must maintain temperature within a narrow tolerance band. Testing such systems requires careful attention to **boundary values** — the critical thresholds where behavior changes. A defect at these boundaries could cause equipment damage, energy waste, or process failures.

In this exercise, you will test the behavior of a small module used in industrial temperature control systems. Your goal is to apply Boundary Value Analysis (BVA) to detect a subtle defect in the implementation.

A regulator receives two parameters: `currentTemperature` (in °C), and `targetTemperature` (in °C). It must decide whether the system should: `HEAT`, `COOL`, or `STANDBY`.

2.2 Specification

The regulator should behave as follows:

- If current $<$ target $-$ 0.5 then the action is `HEAT`
- If current $>$ target $+$ 0.5 then the action is `COOL`
- Otherwise, the action is `STANDBY`

This defines a tolerance zone of ± 0.5 °C around the target temperature.

2.3 Implementation

Understanding the Regulator Logic

The TemperatureRegulator uses a simple threshold-based decision algorithm:

- It calculates the difference between current and target temperature
- Compares this difference against the tolerance threshold (± 0.5°C)
- Returns the appropriate action based on which zone the difference falls into

The use of an `enum` for actions ensures type safety and prevents invalid action values. The implementation uses `else-if` logic, which is crucial for handling boundary cases correctly.

2.3.1 TemperatureRegulator.java

```java
package main;

public class TemperatureRegulator {  29 usages

    public enum Action { HEAT, COOL, STANDBY }  18 usages

    public Action compute(double current, double target) {  7 usages
        final double TOL = 0.5;

        double diff = current - target;

        if (diff < -TOL) {
            return Action.HEAT;
        } else if (diff > TOL) {
            return Action.COOL;
        } else {
            return Action.STANDBY;
        }
    }
}
```

2.4 Task 1: Build the Test Cases

Boundary Value Analysis (BVA)

BVA is a black-box testing technique that focuses on testing at the boundaries of input domains. Defects frequently occur at boundaries because:

- Off-by-one errors in comparisons (< vs)
- Floating-point precision issues
- Incorrect understanding of inclusive vs exclusive ranges

For this regulator, the critical boundaries are at -0.5 and +0.5 degrees from the target. We test:

- Values just below the boundary
- Values exactly at the boundary
- Values just above the boundary

2.5 Task 2: Implement the Test Using JUnit

Temperature Regulator Test Suite

This comprehensive test suite validates all seven test cases identified in the BVA analysis:

- **T1 & T5**: Test values outside the tolerance zone (should trigger HEAT/COOL)
- **T2 & T4**: Critical boundary tests at exactly ±0.5° (should be STANDBY)
- **T3**: Test at perfect equilibrium (0.0° difference)
- **T6 & T7**: Extreme values well outside tolerance zone

Each test method name clearly describes the scenario being tested, making test failures immediately understandable. The assertion messages provide context about what was expected and why.

2.5.1 TemperatureRegulatorTest.java

```java
package test;
import main.TemperatureRegulator;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class TemperatureRegulatorTest {

    @Test
    void compute_currentBelowLowerBoundary_shouldReturnHeat() {
        // Arrange
        TemperatureRegulator regulator = new TemperatureRegulator();
        // Act
        TemperatureRegulator.Action result = regulator.compute( current: 19.4,  target: 20.0);
        // Assert
        assertEquals(TemperatureRegulator.Action.HEAT, result,
                 message: "Current 19.4 with target 20.0 (diff = -0.6) should HEAT");
    }

    @Test
    void compute_currentAtLowerBoundary_shouldReturnStandby() {
        // Arrange
        TemperatureRegulator regulator = new TemperatureRegulator();
        // Act
        TemperatureRegulator.Action result = regulator.compute( current: 19.5,  target: 20.0);
        // Assert
        assertEquals(TemperatureRegulator.Action.STANDBY, result,
                 message: "Current 19.5 with target 20.0 (diff = -0.5) should STANDBY");
    }
```

Figure 1: Enter Caption

2.5.2

```java
@Test
void compute_currentEqualToTarget_shouldReturnStandby() {
    // Arrange
    TemperatureRegulator regulator = new TemperatureRegulator();
    // Act
    TemperatureRegulator.Action result = regulator.compute( current: 20.0,  target: 20.0);
    // Assert
    assertEquals(TemperatureRegulator.Action.STANDBY, result,
             message: "Current 20.0 with target 20.0 (diff = 0.0) should STANDBY");
}

@Test
void compute_currentAtUpperBoundary_shouldReturnStandby() {
    // Arrange
    TemperatureRegulator regulator = new TemperatureRegulator();
    // Act
    TemperatureRegulator.Action result = regulator.compute( current: 20.5,  target: 20.0);
    // Assert
    assertEquals(TemperatureRegulator.Action.STANDBY, result,
             message: "Current 20.5 with target 20.0 (diff = +0.5) should STANDBY");
}

@Test
void compute_currentAboveUpperBoundary_shouldReturnCool() {
    // Arrange
    TemperatureRegulator regulator = new TemperatureRegulator();
    // Act
    TemperatureRegulator.Action result = regulator.compute( current: 20.6,  target: 20.0);
```

2.5.3 Output

2.6 Analysis

Implementation Correctness

The test results confirm that the implementation correctly handles all boundary cases:

- Tests T2 and T4 (at boundaries ± 0.5) pass, showing that boundary values are included in the STANDBY zone
- The implementation uses strict inequalities ($<$ and $>$) for the HEAT and COOL conditions
- The `else` clause correctly captures all values in the range [-0.5, +0.5]

If the specification required exclusive boundaries (values at ± 0.5 should trigger actions), we would need to modify the comparison operators to and , and these tests would detect that discrepancy.

3 Exercise 3: Quadratic Polynomial

3.1 Mathematical Background

Quadratic Equations in Software

Quadratic equations appear frequently in computer graphics, physics simulations, optimization problems, and engineering calculations. Implementing a robust quadratic solver requires:

- Correct mathematical formula implementation
- Handling of three distinct cases (2 roots, 1 root, no real roots)
- Numerical stability considerations
- Proper validation of inputs (a 0)

Testing such implementations is crucial because small errors in the discriminant calculation or root formula can lead to completely incorrect results.

A quadratic polynomial is a function of the form:

$$P(X) = aX^2 + bX + c$$

The roots are given by the discriminant:

$$\Delta = b^2 - 4ac$$

- If $\Delta > 0$: 2 real roots: $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$
- If $\Delta = 0$: 1 real root: $x = \frac{-b}{2a}$
- If $\Delta < 0$: No real roots

3.2 Task 1: Write the Java Class

PolynomeSecondDegre Implementation

This class encapsulates a quadratic polynomial with several key design decisions:

- **Constructor validation**: Throws `IllegalArgumentException` if a=0 (not a quadratic)
- **Discriminant method**: Separates calculation for reusability and clarity
- **Root calculation**: Returns arrays of appropriate size (0, 1, or 2 elements)
- **Evaluate method**: Allows verification by substituting roots back into the equation

The implementation carefully handles the three cases for root calculation, ensuring that the returned array size matches the number of real roots.

```java
package main;

public class PolynomeSecondDegre {  24 usages

    private double a;  6 usages
    private double b;  7 usages
    private double c;  3 usages

    public PolynomeSecondDegre(double a, double b, double c) {  12 usages
        if (a == 0) {
            throw new IllegalArgumentException(
                    "Coefficient 'a' cannot be zero for a quadratic polynomial");
        }
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public double getDiscriminant() {  3 usages
        return b * b - 4 * a * c;
    }

    public int getNumberOfRoots() {  3 usages
        double delta = getDiscriminant();
        if (delta > 0) {
            return 2;
        } else if (delta == 0) {
            return 1;
        } else {
            return 0;
```

```java
        } else if (delta == 0) {
            return 1;
        } else {
            return 0;
        }
    }

    public double[] getRoots() {  5 usages
        double delta = getDiscriminant();

        if (delta < 0) {
            return new double[0]; // No real roots
        } else if (delta == 0) {
            double root = -b / (2 * a);
            return new double[]{root};
        } else {
            double sqrtDelta = Math.sqrt(delta);
            double root1 = (-b - sqrtDelta) / (2 * a);
            double root2 = (-b + sqrtDelta) / (2 * a);
            return new double[]{root1, root2};
        }
    }

    public double evaluate(double x) {  3 usages
        return a * x * x + b * x + c;
    }
}
```

3.2.1 PolynomeSecondDegre.java

3.3 Task 2: Identify Test Cases

3.4 Task 3: Write the Test Code

Polynomial Test Suite Structure

The test suite systematically validates all aspects of the polynomial solver:

- **Input validation**: Ensures constructor rejects invalid input (a=0)
- **Discriminant calculation**: Verifies the Δ formula is correct
- **Root counting**: Confirms the correct number of roots is reported
- **Root accuracy**: Tests that calculated roots are mathematically correct
- **Edge cases**: Negative leading coefficient, double roots
- **Verification**: Uses evaluate() method to confirm roots satisfy P(x)=0

Note the use of `EPSILON = 1e-9` for floating-point comparisons, accounting for numerical precision limitations.

```java
package test;
import main.PolynomeSecondDegre;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
class PolynomeSecondDegreTest {
    private static final double EPSILON = 1e-9;
    @Test
    void constructor_withZeroA_shouldThrowException() {

        assertThrows(IllegalArgumentException.class, () -> {
            new PolynomeSecondDegre(0, 2, 1);
        }, "Creating polynomial with a=0 should throw exception");
    }
    @Test
    void getDiscriminant_withTwoRoots_shouldReturnPositive() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre(1, -3, 2);
        double delta = poly.getDiscriminant();
        assertEquals(1.0, delta, EPSILON, "Delta should be 1");
    }
    @Test
    void getNumberOfRoots_withTwoRoots_shouldReturnTwo() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre(1, -3, 2);
        int numberOfRoots = poly.getNumberOfRoots();
        assertEquals(2, numberOfRoots, "Should have 2 roots");
    }
    @Test
    void getRoots_withTwoRoots_shouldReturnCorrectValues() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre(1, -3, 2);
        double[] roots = poly.getRoots();
        assertEquals(2, roots.length, "Should return 2 roots");
```

3.4.1 PolynomeSecondDegreTest.java

```java
        assertEquals( expected: 2, roots.length,  message: "Should return 2 roots");
        assertEquals( expected: 1.0, roots[0], EPSILON,  message: "First root should be 1");
        assertEquals( expected: 2.0, roots[1], EPSILON,  message: "Second root should be 2");
    }
    @Test
    void getNumberOfRoots_withOneRoot_shouldReturnOne() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: -2,  c: 1);
        int numberOfRoots = poly.getNumberOfRoots();
        assertEquals( expected: 1, numberOfRoots,  message: "Should have 1 root");
    }
    @Test
    void getRoots_withOneRoot_shouldReturnSingleValue() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: -2,  c: 1);
        double[] roots = poly.getRoots();
        assertEquals( expected: 1, roots.length,  message: "Should return 1 root");
        assertEquals( expected: 1.0, roots[0], EPSILON,  message: "Root should be 1");
    }
    @Test
    void getNumberOfRoots_withNoRealRoots_shouldReturnZero() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: 0,  c: 1);
        int numberOfRoots = poly.getNumberOfRoots();
        assertEquals( expected: 0, numberOfRoots,  message: "Should have 0 real roots");
    }
    @Test
    void getRoots_withNoRealRoots_shouldReturnEmptyArray() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: 0,  c: 1);
        double[] roots = poly.getRoots();
        assertEquals( expected: 0, roots.length,  message: "Should return empty array");
    }
```

```java
    }
    @Test
    void getRoots_withNegativeLeadingCoefficient_shouldWork() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: -1,  b: 0,  c: 4);
        double[] roots = poly.getRoots();
        assertEquals( expected: 2, roots.length,  message: "Should return 2 roots");
        assertEquals( expected: -2.0, roots[0], EPSILON,  message: "First root should be -2");
        assertEquals( expected: 2.0, roots[1], EPSILON,  message: "Second root should be 2");
    }
    @Test
    void evaluate_atRoot_shouldReturnZero() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: -3,  c: 2);
        double result = poly.evaluate( x: 1.0);
        assertEquals( expected: 0.0, result, EPSILON,  message: "P(1) should be 0");
    }

    @Test
    void evaluate_notAtRoot_shouldReturnNonZero() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: -3,  c: 2);
        double result = poly.evaluate( x: 0.0);
        assertEquals( expected: 2.0, result, EPSILON,  message: "P(0) should be 2");
    }
    @Test
    void getRoots_exampleFromDocument_shouldWork() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: 1,  c: -2);
        double[] roots = poly.getRoots();
        assertEquals( expected: 2, roots.length,  message: "Should return 2 roots");
```

```java
    @Test
    void evaluate_notAtRoot_shouldReturnNonZero() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: -3,  c: 2);
        double result = poly.evaluate( x: 0.0);
        assertEquals( expected: 2.0, result, EPSILON,  message: "P(0) should be 2");
    }
    @Test
    void getRoots_exampleFromDocument_shouldWork() {
        PolynomeSecondDegre poly = new PolynomeSecondDegre( a: 1,  b: 1,  c: -2);
        double[] roots = poly.getRoots();
        assertEquals( expected: 2, roots.length,  message: "Should return 2 roots");
        double eval1 = poly.evaluate( x: 1.0);
        assertEquals( expected: 0.0, eval1, EPSILON,  message: "P(1) should be 0");
    }
}
```

3.4.2 Output