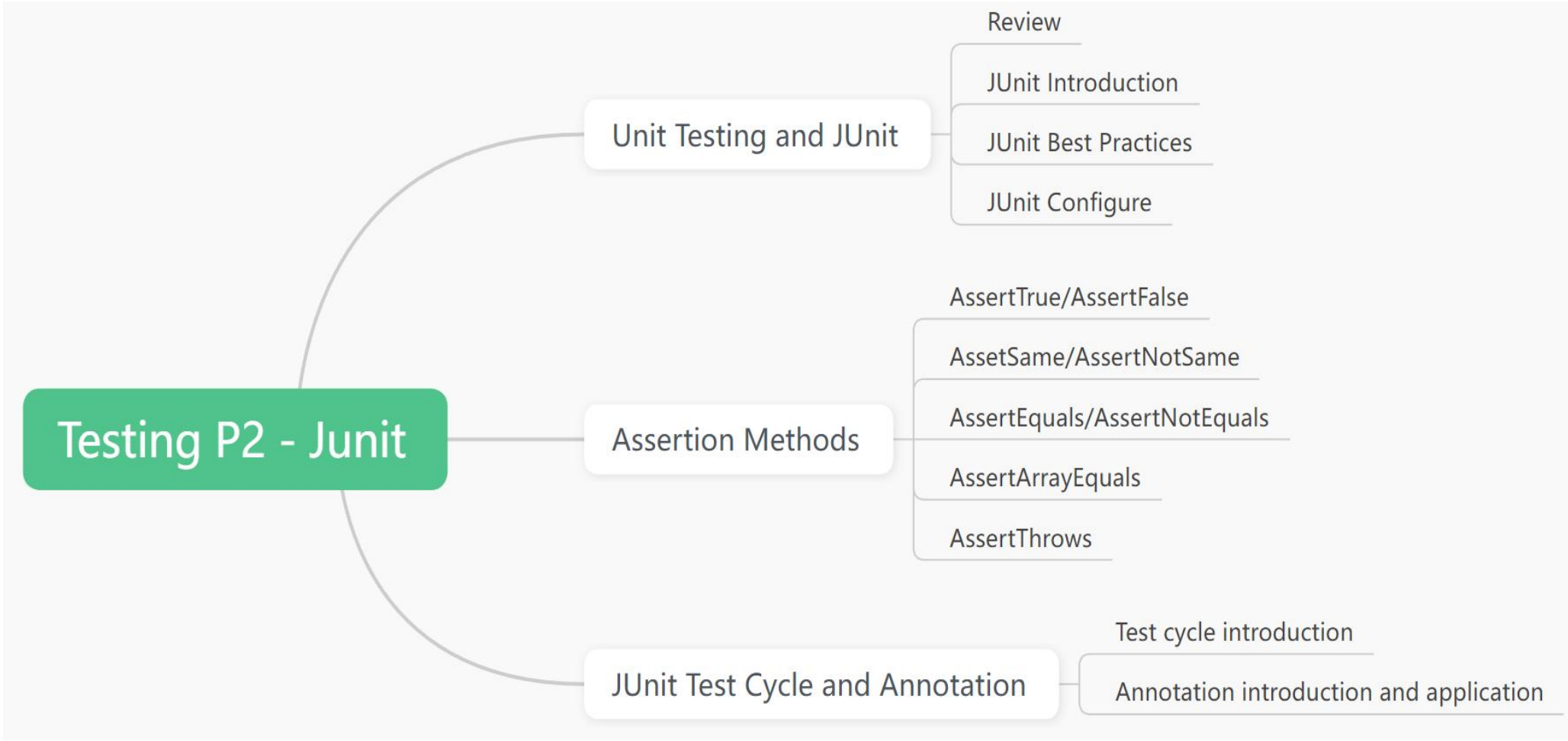
Software Testing Part 2
Junit Testing

AY25/26

Week 11

Outline


Testing P2 - Junit
Unit Testing and JUnit
Review
JUnit Introduction
JUnit Best Practices
JUnit Configure
Assertion Methods
AssertTrue/AssertFalse
AssetSame/AssertNotSame
AssertEquals/AssertNotEquals
AssertArrayEquals
AssertThrows
JUnit Test Cycle and Annotation
Test cycle introduction
Annotation introduction and application

1. Unit Testing and JUnit

1.1 Review - Unit Testing

- Testing of an individual software unit
  - usually an object class
- Focus on the functions of the unit
  - functionality, correctness, accuracy
- Usually carried out by the developers of the unit

1.2 Review - Automated Framwork

- A **setup part,** where you initialize the system with the test case (e.g., initialize the object under test)
- A **call part,** where you call the object or method to be tested.
- An **assertion part** where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

1.3 JUnit

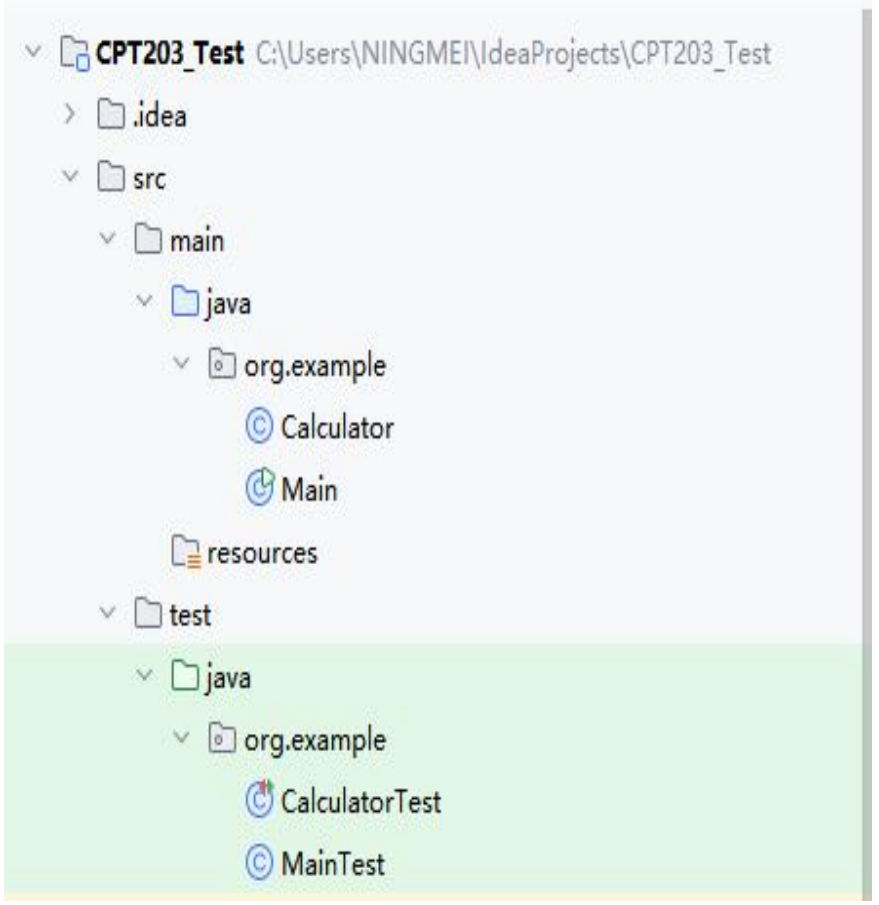
- JUnit is a framework for writing unit tests
  - Designed for the purpose of writing and running tests on Java code
  - A Junit test examines one class, and each test focuses on checking one method within that class.
  - Ensures individual test cases are executed in isolation, promoting more accurate results
- Why Junit
  - Enhanced Code Quality
  - Java-based
  - Integrates seamlessly with IDEs like Eclipse and build tools like Maven and Gradle
  - Free

```
public class CalculatorTest {

    @Test
    void testAdd() {
        Calculator c = new
Calculator();
        int result = c.add(2, 3);
        assertEquals(5, result);
    }

    @Test
    void testSubtract() {
        Calculator c = new
Calculator();
        int result =
c.subtract(10, 4);
        assertEquals(6, result);
    }
}
```

1.3.1 Junit Test Structure

Formal structure (e.g., in Maven Project)

Class that is being tested and its test class (where the test is implemented) are **seperated in different folders.**



1.3.1 Junit Test Structure

Simplified structure (e.g., in a regular IntelliJ Project)

In a basic IntelliJ-created Java project, the class under test and its corresponding test class are placed **in the same src folder,** without a formal separation between production code and test code.

1.3.1 Junit Test Structure

Non-standard (same-file) test placement

This layout may appear in exams/class demonstration for quick question/problem identification. However, it is **not recommended** in real projects (e.g., CW2), as production code and test code should not be mixed in the same source folder.

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

class CalculatorTest {

    @Test
    void testAdd() {
        Calculator c = new Calculator();
        assertEquals(5, c.add(2, 3));
    }
}
```

1.3.2 Junit Test Verdicts

A *verdict* is the result of executing a single test case.

Pass

- The test case execution was completed
- The function being tested performed as expected

Fail

- The test case execution was completed
- The function being tested did *not* perform as expected

Error

- The test case execution was not completed, due to
  - an unexpected event, exceptions, or
  - improper set up of the test case, etc.

1.3.2 Junit Test Verdicts (cont.)

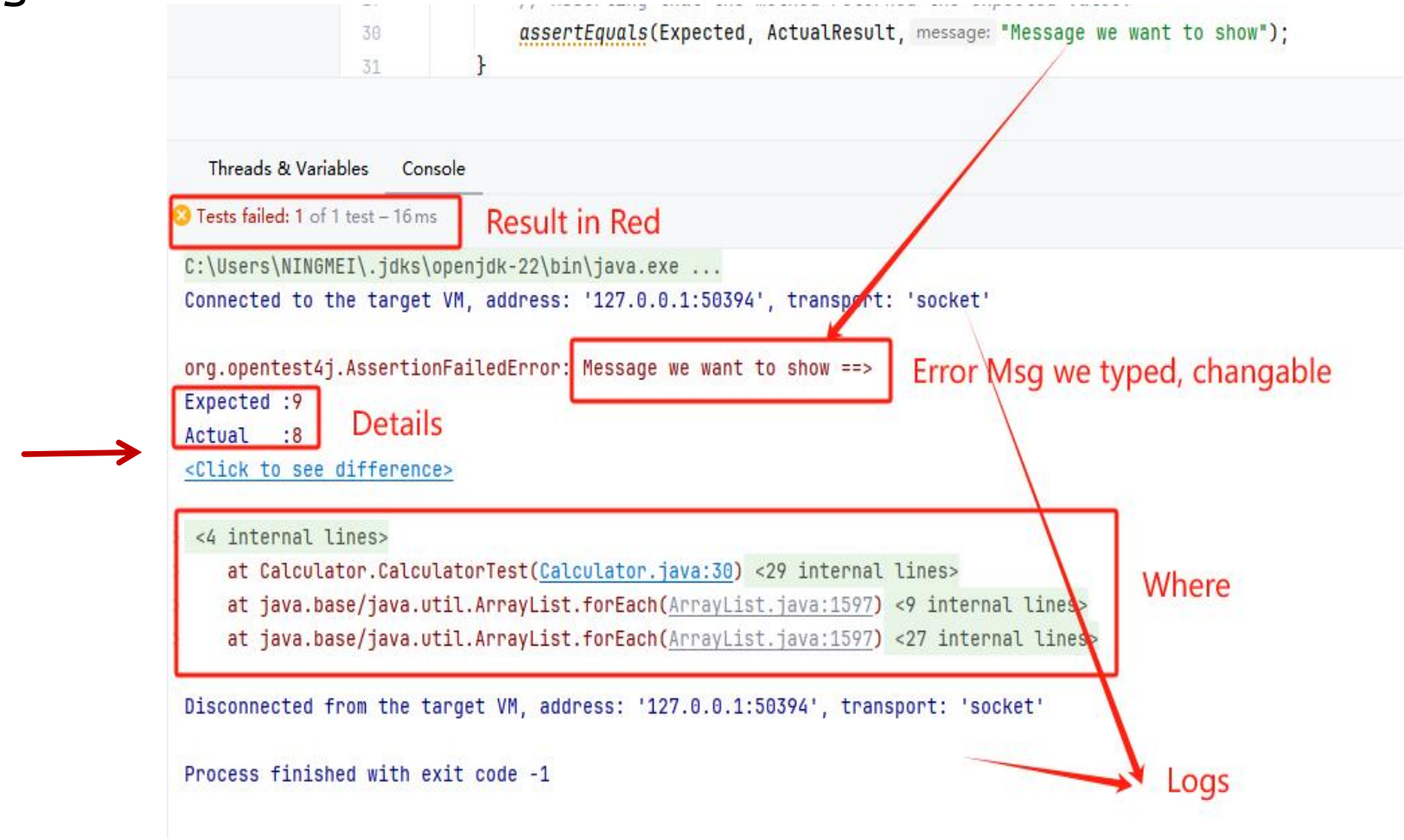
- If the tests run correctly, a test method does nothing but shows the results in Green

```
int input1 =5;
int input2 = 3;
int Expected = 8;
```

1.3.2 Junit Test Verdicts (cont.)

- If a test fails

```
int input1 =5;
int input2 = 3;
int Expected = 9;
```



1.3.2 Junit Test Verdicts (cont.)

- If an error happens

```
Calculator calculator = new Calculatorsss();
int input1 =5;
int input2 = 3;
int Expected = 8;
```

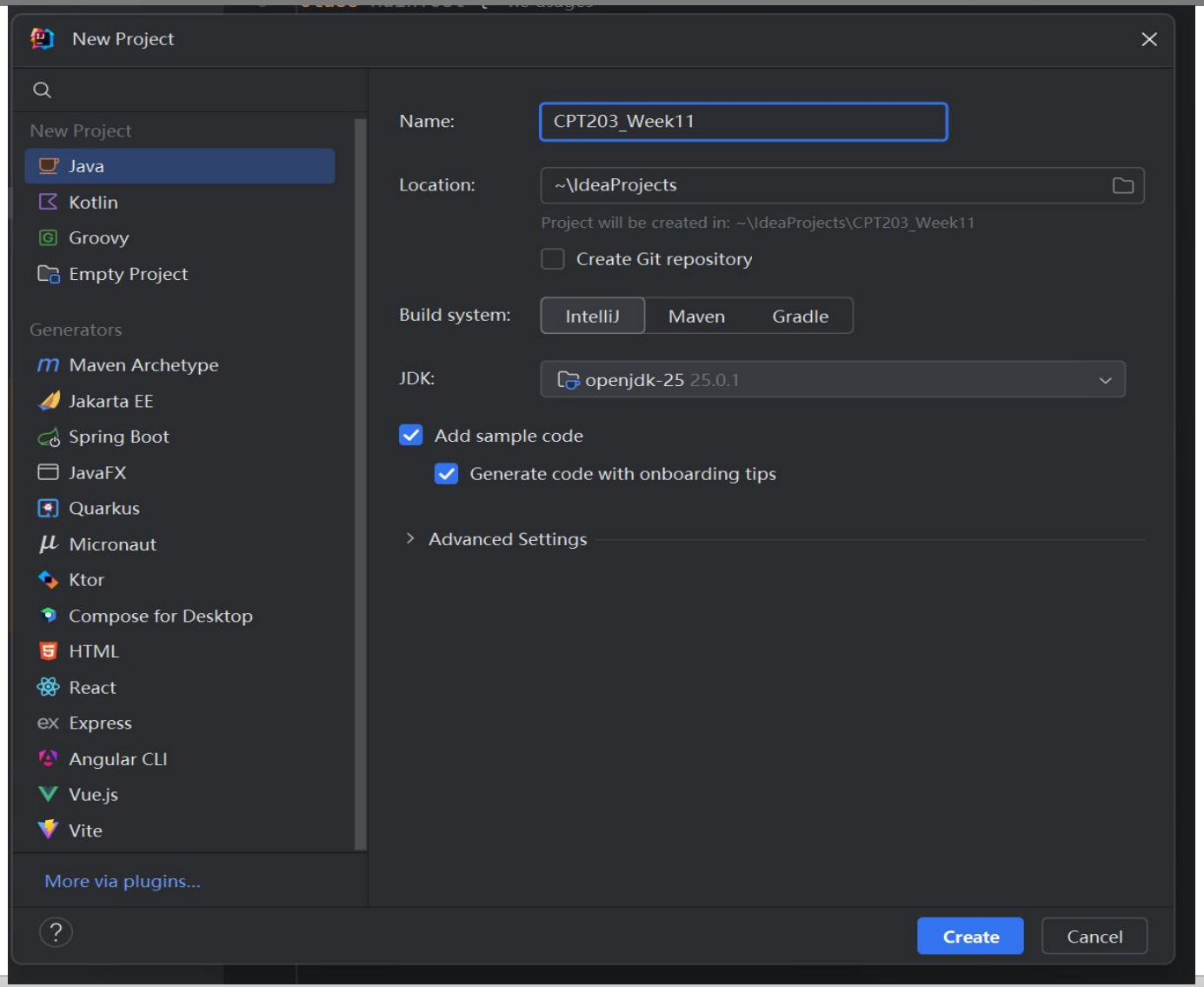
```
C:\Users\NINGMEI\IdeaProjects\123123\src\Calculator.java:19:37
java: 找不到符号
  符号:   类 Calculatorsss
  位置: 类 Calculator
```

1.4 JUnit Best Practices

- Tests need *failure atomically* (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
    - Each test should have roughly just 1 assertion at its end.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always `assertTrue`.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.

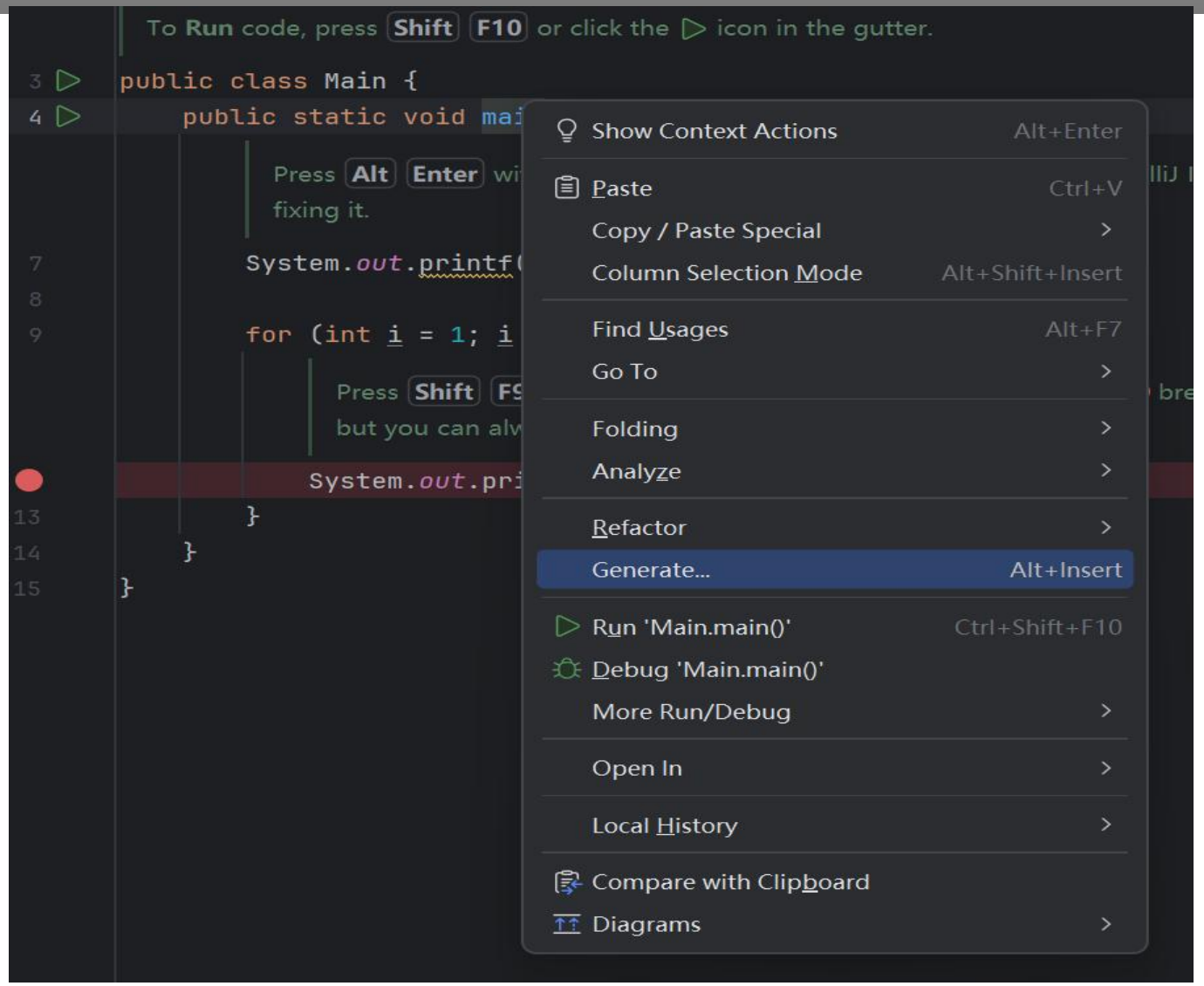
1.5 Configure JUnit in IntelliJ

- Create a New Poject (e.g., IntelliJ)



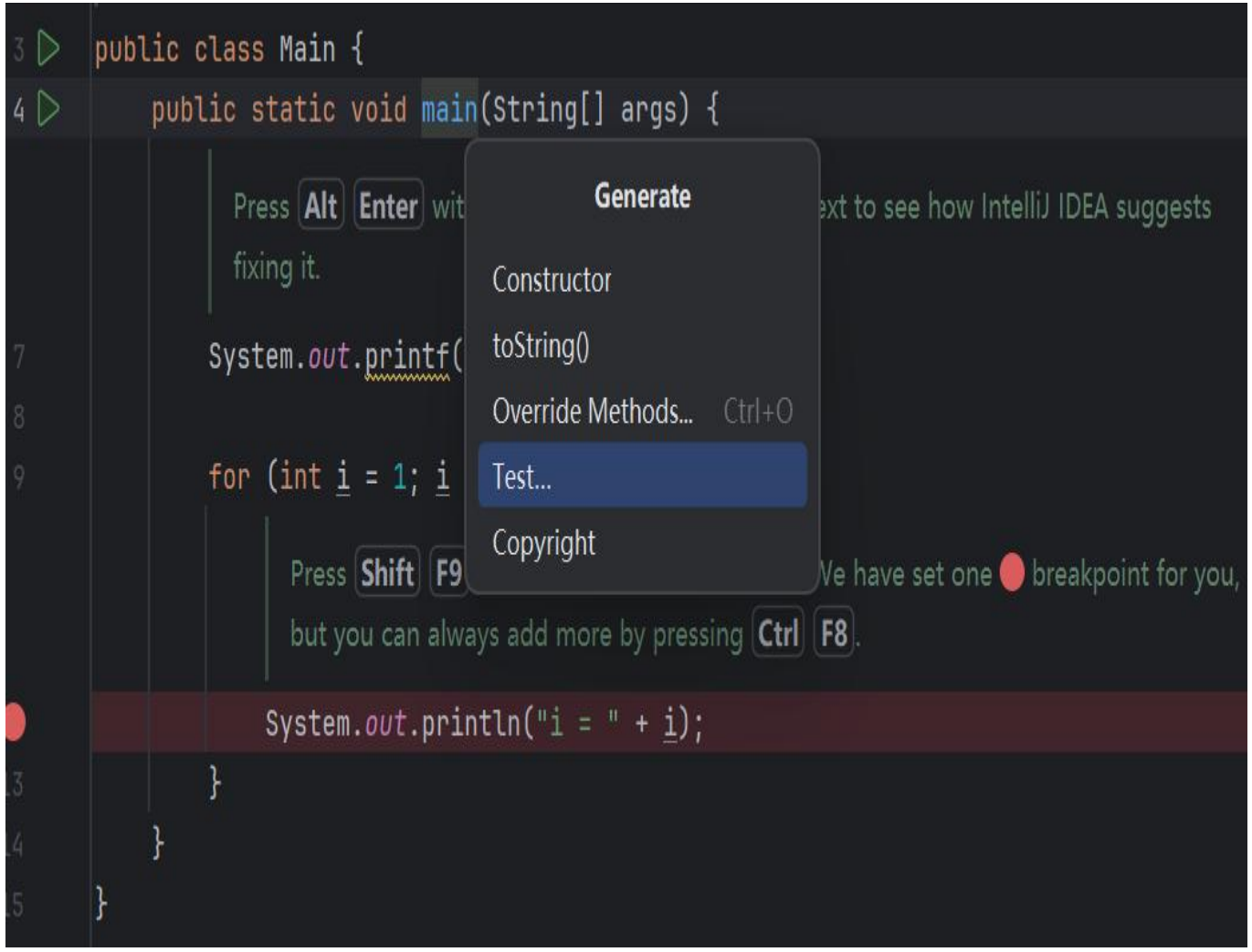
1.5 Configure JUnit in IntelliJ

- Right click the default **main method** and select **Generate**



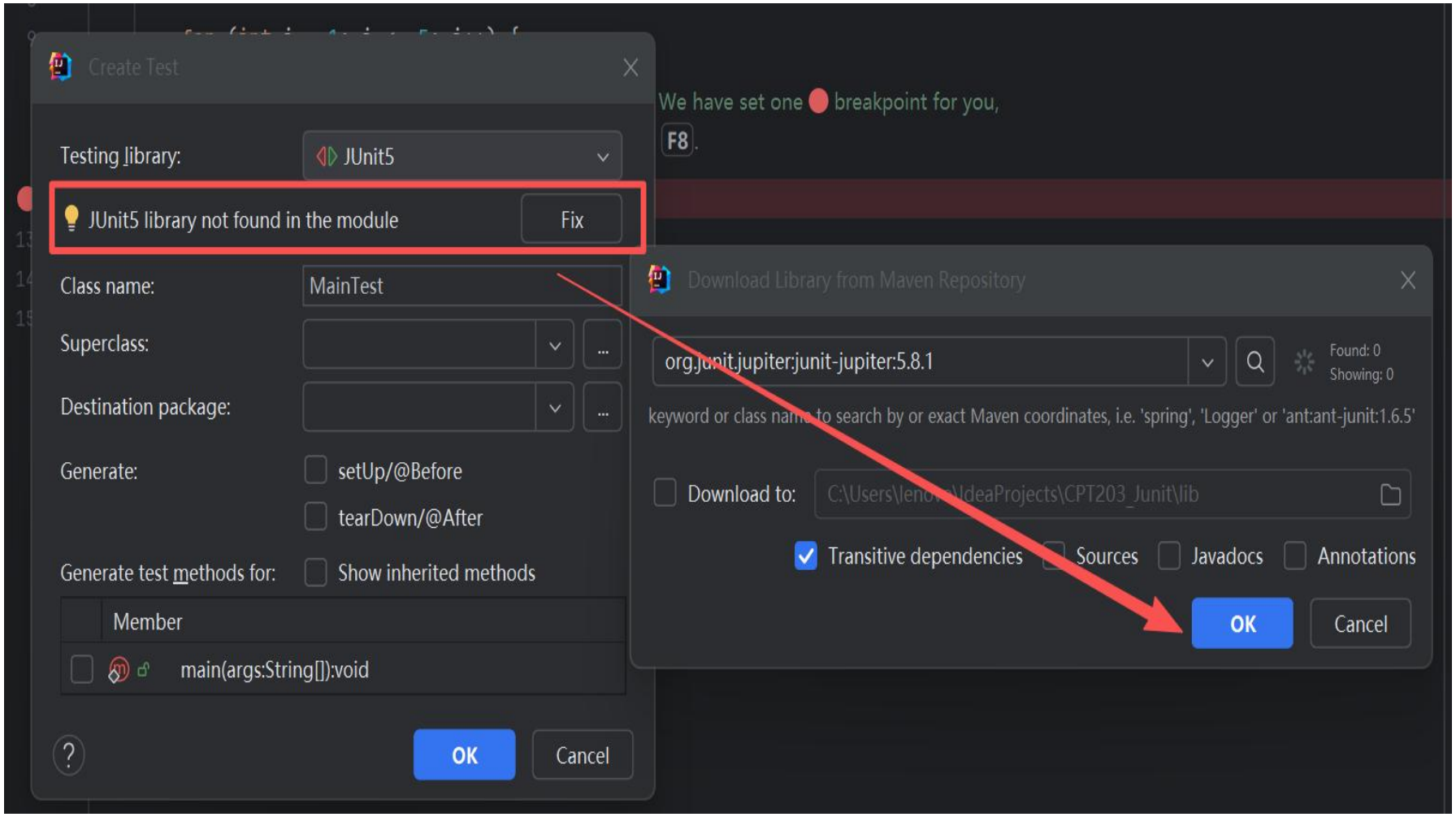
1.5 Configure JUnit in IntelliJ

- Select **Test** in the popup window



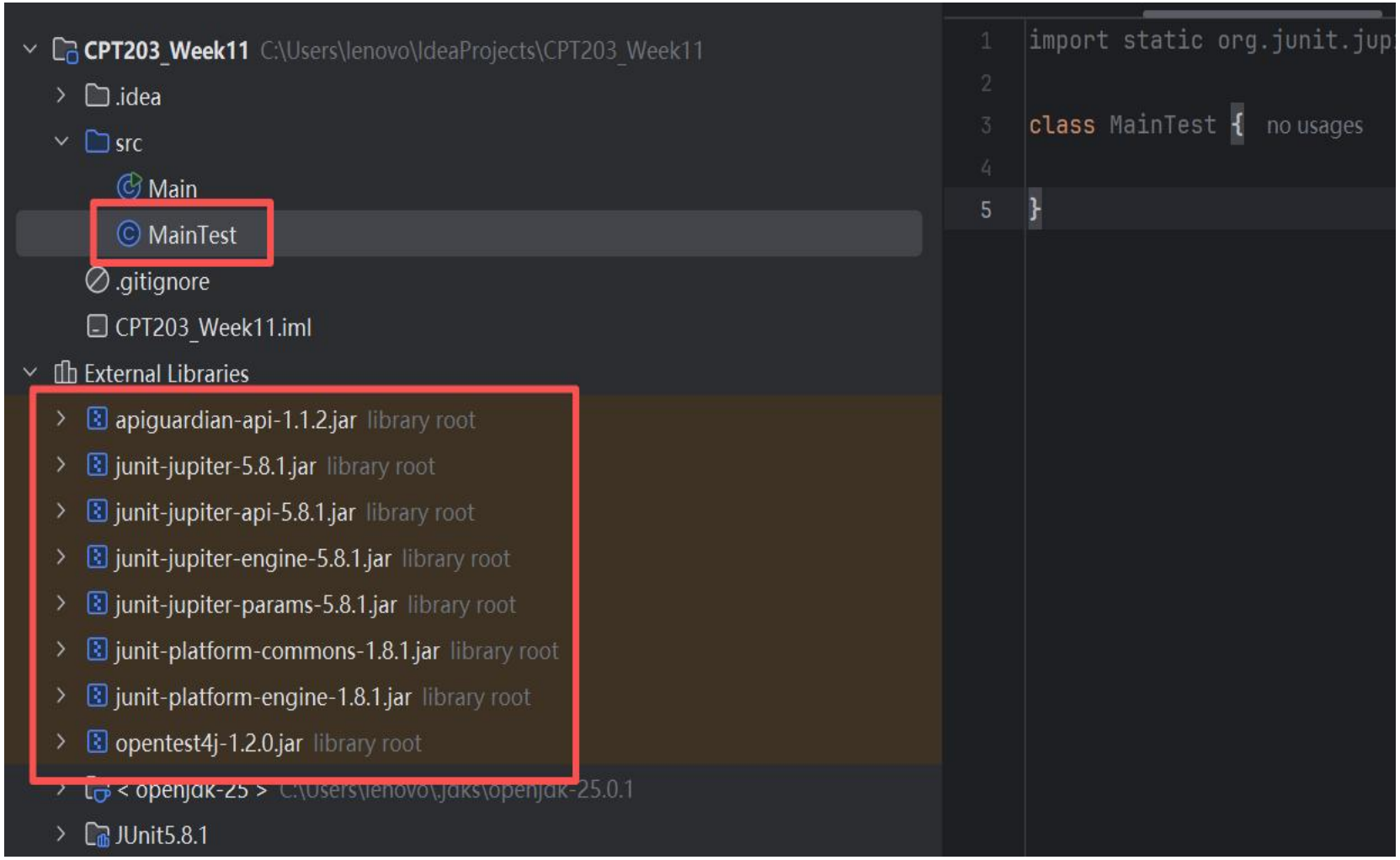
1.5 Configure JUnit in IntelliJ

- Click the **Fix** button
- And download the Junit by clicking **OK**



1.5 Configure JUnit in IntelliJ

- The MainTest class is created
- The Junit has been added to the library
- Done



2. Assertion Methods

2.1 AssertTrue/AssertFlase

- Assert a Boolean condition is true or false

  assertTrue(*condition*)
  assertFalse(*condition*)

- Optionally, include a failure message

  assertTrue(*condition, message*)
  assertFalse(*condition, message*)

```
public class NumberChecker {  2 usages
    public boolean isPositive(int number) {  1 usage
        return number > 0;
    }
}
```

```
package org.example;

import org.junit.jupiter.api.Test;

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

class NumberCheckerTest {

    @Test
    public void testIsPositive() {
        NumberChecker checker = new NumberChecker();
        boolean isPositive = checker.isPositive( number: 5);
        assertTrue(isPositive);
        assertFalse(isPositive,  message: "This failure message is optional");
    }
}
```

2.2 AssertSame/AssertNotSame

- Assert two object references are identical

  assertSame(*expected, actual*)

  - True if: expected == actual

  assertNotSame(*expected, actual*)

  - True if: expected != actual

- With a failure message

  assertSame(*expected, actual, (optional) message*)
  assertNotSame(*expected, actual, (optional) message*)

- Note: Compare if two objects are exactly the same one, NOT the value

2.2 AssertSame/AssertNotSame (cont.)

```java
package org.example;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class assertSame {
    @Test
    public void assertSameShowcase() {
        String s1 = new String( original: "Hello");
        String s2 = new String( original: "Hello");
        Assertions.assertSame(s1, s2);
    }
}
```

```
Tests failed: 1 of 1 test – 15 ms

C:\Users\NINGMEI\.jdks\openjdk-22\bin\java.exe ...

org.opentest4j.AssertionFailedError: expected: java.lang.String@491666ad<Hello> but was: java.lang.String@176d53b2<Hello>
Expected :Hello
Actual   :Hello
<Click to see difference>

<5 internal lines>
    at org.example.assertSame.assertSameShowcase(assertSame.java:11) <29 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1597) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1597) <27 internal lines>


Process finished with exit code -1
```

2 object references (in hash code) are different, indicating they are stored in different memory location in Java

2.3 AssertEquals/AssertNotEquals

- Assert two objects are equal to each regarding value/content
- It doesn't matter if expected and actual are the same object or different object; as long as their content is equal, the test will pass.

  assertEquals(expected, actual, *(optional) message*)
- Not only for int type, but also for other values (e.g., string, float, ...)

```java
package org.example;

public class StringCase {  2 usages
    public String combining(String one, String two) {  1 usage
        return one + two;
    }
}
```

```java
package org.example;

import org.junit.jupiter.api.Test;

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

public class assertEquals {

@Test
public void testCombining() {
    StringCase SC = new StringCase();

    String expected = "HelloWorld";
    String actual = SC.combining( one: "Hello",  two: "World");

    assertEquals(expected, actual);
  }
}
```

2.4 AssertArrayEquals

- Assert two arrays are equal:

  assertArrayEquals(expected, actual, *(optinal) message*)

  - arrays must have same length
  - Check for each valid index i, comparing values at index 0, 1, 2, ...

```
public class Array {  1 usage

    public static int[] generateArray() {  1 usage
        return new int[] {1,2,3};
    }
}
```

```
package org.example;

import org.junit.jupiter.api.Test;

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

class ArrayTest {
@Test
    public void testArray() {
    int[] expectedArray = new int[] {3, 2, 1};
    int[] actualArray = Array.generateArray();
    assertArrayEquals(expectedArray, actualArray);}
}
```

2.5 AssertThrows

- Used to test that a specific type of exception is thrown during the execution of a block of code
- assertThrows(expectedExceptionClass, executable)
  - expectedExceptionClass is the type of exception you expect
  - executable is a *lambda expression* or a method reference that executes the code under test
- Particularly useful for negative test cases where you want to ensure that your code fails under certain conditions

Examples:

- ArithmeticException: Thrown when a number divided by zero.
- ArrayIndexOutOfBoundsException: Thrown when accessing an invalid index in an array.
- FileNotFoundException: Thrown when attempting to access a file that does not exist.

2.5 AssertThrows (cont.)

```java
package org.example;


public class AssertThrowsCalCase { no usages
    public double divide(int numerator, int denominator) { no usages
        if (denominator == 0) {
            throw new ArithmeticException("Division by zero is not allowed");
        }
        return (double) numerator / denominator;
    }

}
```

```java
class AssertThrowsCalCaseTest {
    @Test
    public void testDivisionByZeroThrowsException() {
        // Create an instance of the AssertThrowsCalCase class.
        // This class contains the 'divide' method we want to test.
        AssertThrowsCalCase calculator = new AssertThrowsCalCase();

        // Use assertThrows to test that an ArithmeticException is thrown.
        // assertThrows takes two main parameters:
        // 1. The class of the exception we expect to be thrown.
        // 2. A lambda expression that executes the code we're testing.
        assertThrows(
                //The expected exception type.
                //Here, we expect an ArithmeticException.
                ArithmeticException.class,
                // This is the lambda expression.
                // It is used to execute the 'divide' method of the calculator object.
                // The 'divide' method is called with arguments 10 and 0.
                () -> calculator.divide( numerator: 10,  denominator: 0));
    }
}
```

1

2

3

1. Parameter List 2. operator 3. Body: method being tested with the case (10 and 0)

2.5 AssertThrows (cont.)

Why use Lambda (Correct Approach):

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

- This approach delays the execution of divide until assertThrows can catch the exception.

Otherwise, it will be a problem, because:

- *assertThrows(ArithmeticException.class, calculator.divide(10, 0));*
- In this case, divide() is executed immediately, and if it throws an exception, it happens before assertThrows can catch it, leading to a test error.

3. JUnit Test Cycle and Annotation

3.1 Life Cycle

- Normally, a test class contains multiple test methods. JUnit manages the execution of each test method in form of a lifecycle.

- The complete lifecycle of a test case can be seen in three phases with the help of annotations.

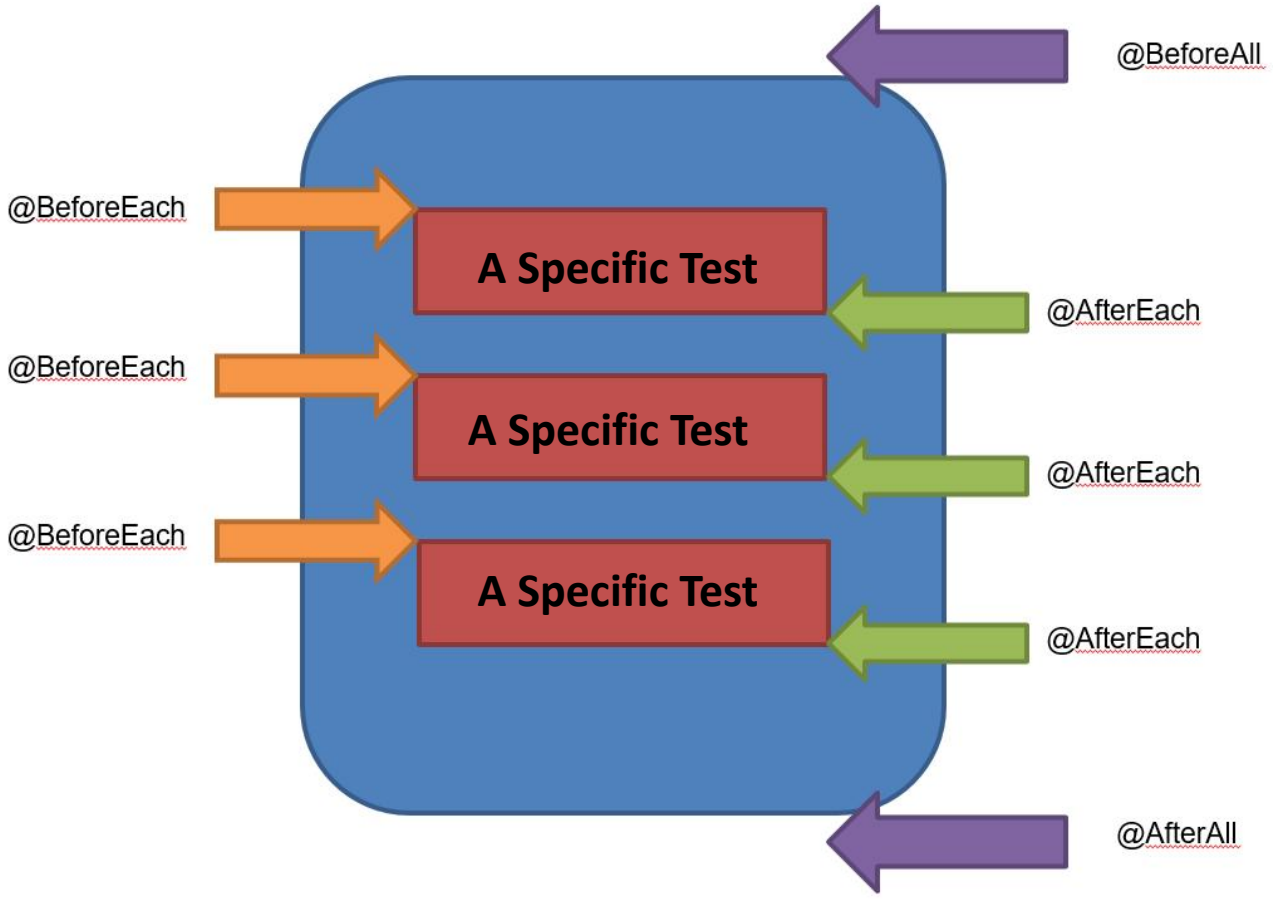
3.2 Life Cycle Phases

1. **Setup**: This phase puts the test infrastructure in place. JUnit provides class level setup *(@BeforeAll)* and method level setup *(@BeforeEach)*. Generally, heavy objects like databse comnections are created in class level setup while lightweight objects like test objects are reset in the method level setup.

2. **Test Execution**: In this phase, the test execution and assertion happen. The execution result will signify a success or failure.

3. **Cleanup**: This phase is used to cleanup the test infrastructure setup in the first phase. Just like setup, teardown also happen at class level *(@AfterAll)* and method level *(@AfterEach).*

3.2 Life Cycle Phases (cont.)



@BeforeAll and @AfterAll must be static:

- They run once for the entire test class, not for individual tests.
- Static methods do not rely on object state, so they are ideal for global setup/cleanup.
- JUnit enforces static to avoid confusion between class-level and per-test behavior.

@BeforeEach and @AfterEach must be non-static:

- They run before and after every test method.
- JUnit creates a new test object for each test, so these methods must work on that object.
- Non-static methods can access and reset the fields of the current test object, instead of the global variables, which is essential for preparing consistent test conditions.

3.2 Life Cycle Phases (cont.)

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


    public class JunitLifeCycle {

        // Class level setup - runs once before all tests
        @BeforeAll
        static void setupClass() {
            // Code to set up database connections or other heavy resources
        }

        // Method level setup - runs before each test
        @BeforeEach
        void setupTest() {
            // Code to initialize or reset test objects
        }

        // Actual test case
        @Test
        void testExample() {
            // Test execution and assertions
            assertEquals( expected: 2,  actual: 1 + 1);
        }

        // Method level cleanup - runs after each test
        @AfterEach
        void tearDownTest() {
            // Code to reset or clean up after each test
        }

        // Class level cleanup - runs once after all tests
        @AfterAll
        static void tearDownClass() {
            // Code to clean up database connections or other heavy resources
        }
}
```

e.g., link to the whole database

e.g., select a specific dataset for the test below

Test implementation

e.g., clean up the spefici dataset for next round of test

e.g., Disconnect the whole databased and end test

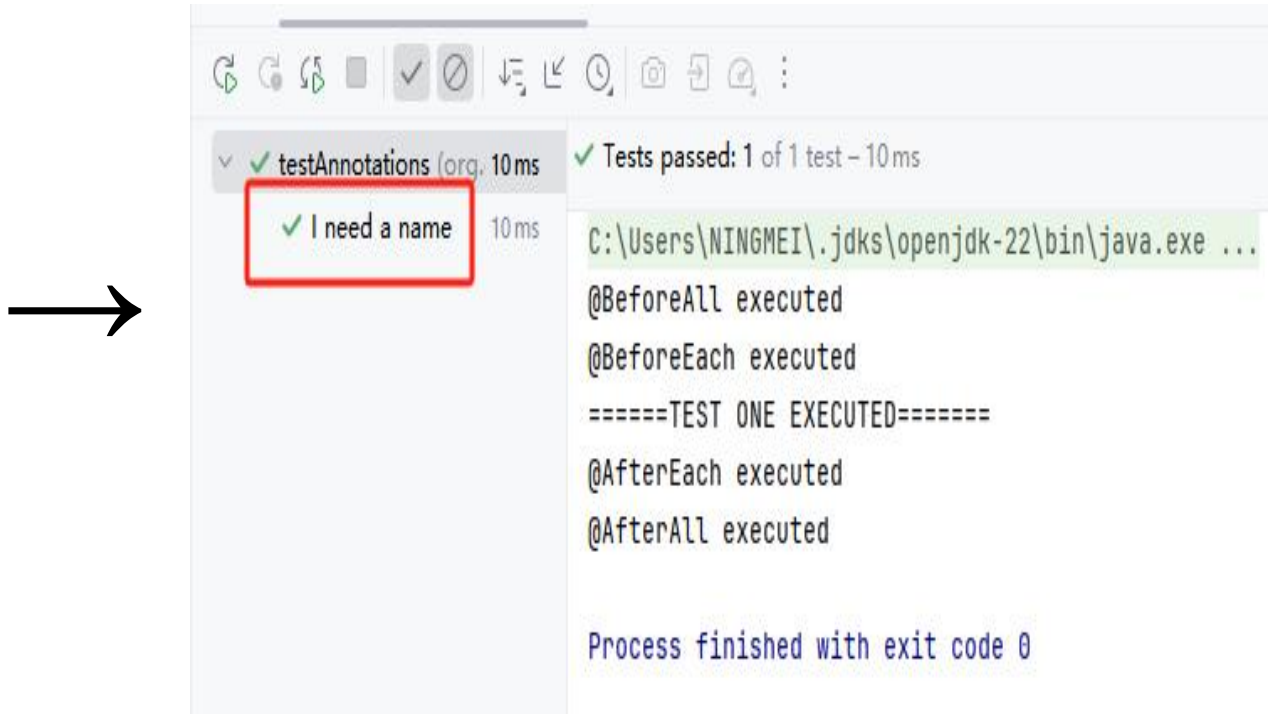
3.3 Junit 5 Annotations

Annotation	Description
`@BeforeEach`	The annotated method will be run before each test method in the test class.
`@AfterEach`	The annotated method will be run after each test method in the test class.
`@BeforeAll`	The annotated method will be run before all test methods in the test class. This method must be static.
`@AfterAll`	The annotated method will be run after all test methods in the test class. This method must be static.
`@Test`	It is used to mark a method as a junit test.
`@DisplayName`	Used to provide any custom display name for a test class or test method
`@Disable`	It is used to disable or ignore a test class or test method from the test suite.
`@Nested`	Used to create nested test classes
`@Tag`	Mark test methods or test classes with tags for test discovering and filtering
`@TestFactory`	Mark a method is a test factory for dynamic tests.

3.3.1 @DisplayName

- @DisplayName // to display meaningful name appear in the test report

```java
@DisplayName("I need a name")
//@RepeatedTest(3)
@Test
void test1()
{
    System.out.println("======TEST ONE EXECUTED=======");
    assertEquals( expected: 4 , Calculator.add( x: 2,  y: 2));
}
```



3.3.2 @Timeout

- Useful for simple performance test
  - Network communication
  - Complex computation
- The @Timeout annotation
  - Time unit defaults to seconds (@Timeout (1)) but is configurable

```
@DisplayName("I need a name")
//@RepeatedTest(3)
@Timeout(value = 1, unit= TimeUnit.MILLISECONDS)
@Test
void test1()
{
    System.out.println("======TEST ONE EXECUTED=======");
    assertEquals( expected: 4 , Calculator.add( x: 2,  y: 2));
}
```

```
Tests failed: 1 of 1 test – 16 ms
C:\Users\NINGMEI\.jdks\openjdk-22\bin\java.exe ...
@BeforeAll executed
@BeforeEach executed
======TEST ONE EXECUTED=======
@AfterEach executed

java.util.concurrent.TimeoutException: test1() timed out after 1 millisecond
> <26 internal lines>
>     at java.base/java.util.ArrayList.forEach(ArrayList.java:1597) <9 internal lines>
>     at java.base/java.util.ArrayList.forEach(ArrayList.java:1597) <27 internal lines>
```

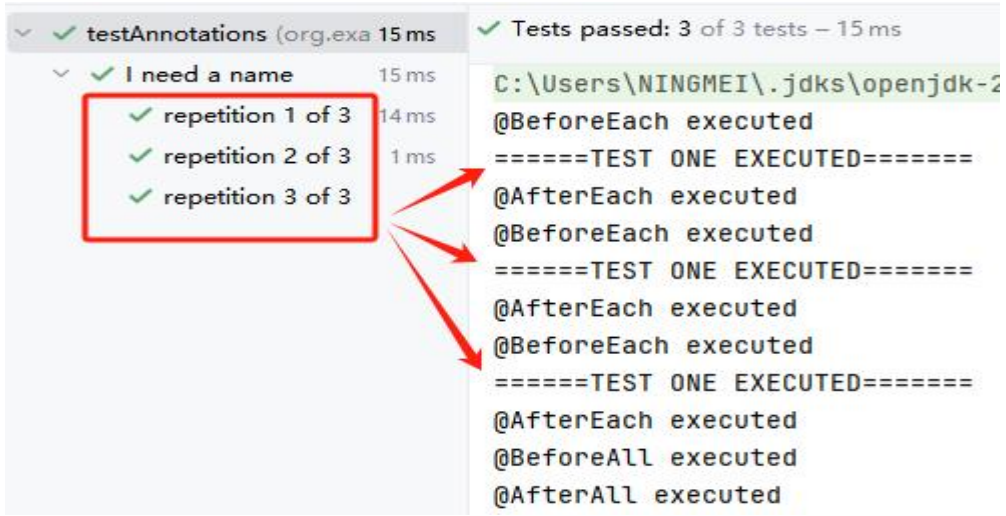
16ms > 1ms, so failed

36

3.3.3 @RepeatedTest

- *@RepeatedTest* is used to mark a test method that should repeat a specified number of times with a configurable display name.
- In the given example, the test method uses @RepeatedTest(3) annotation. It means that the test will be executed 3 times.
- *@Test* would **NOT** be needed if we are using *@RepeatedTest*

```
@DisplayName("I need a name")
@RepeatedTest(3)
//@Test
void test1()
{
    System.out.println("======TEST ONE EXECUTED=======");
    assertEquals( expected: 4 , Calculator.add( x: 2,  y: 2));
}
```



3.3.3 @RepeatedTest (cont.)

- *@RepeatedTest* is used to mark a test method that should repeat a specified number of times with a configurable display name.
- In the given example, the test method uses @RepeatedTest(3) annotation. It means that the test will be executed 3 times.
- *@Test* would **NOT** be needed if we are using *@RepeatedTest*

```
@DisplayName("I need a name")
@RepeatedTest(3)
//@Test
void test1()
{
    System.out.println("======TEST ONE EXECUTED=======");
    assertEquals( expected: 4 , Calculator.add( x: 2,  y: 2));
}
```

Otherwise, output:

...[A Warning Message (not an error)]... +
This is typically the result of annotating a method with multiple competing annotations such as @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, etc