Introduction to JUnit

CS 3250
Software Testing

[Ammann and Offutt, "Introduction to Software Testing," Ch. 3]
[https://junit.org/junit5/docs/current/user-guide/]

Today’s Objectives

- Understand JUnit test classes
- Understand structure of basic JUnit test methods
- Get started with Junit – some commonly used JUnit assertions and other features

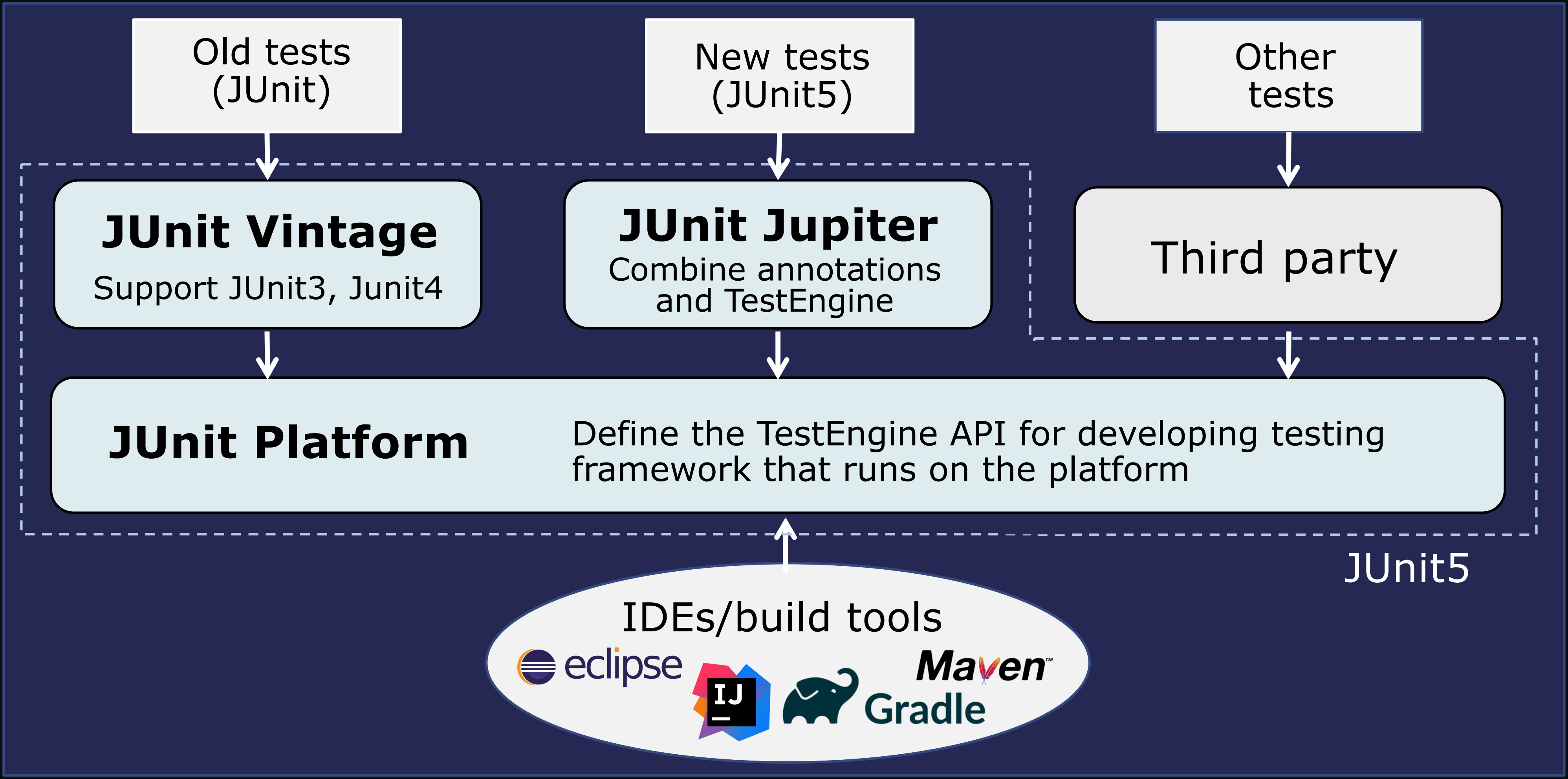
What is JUnit?

- An open source Java testing framework (junit.org) used to write and run repeatable automated tests
- JUnit is widely used in industry
- A structure for writing test drivers
- JUnit features include
  - Assertions to evaluate expected results
  - The ability to share common test data among tests
  - Test sets to easily organize and run tests
  - The ability to run tests from either a command line or a GUI

Junit 5

- JUnit 4 – single jar file
- JUnit 5 – modular, flexible, robust, extensible

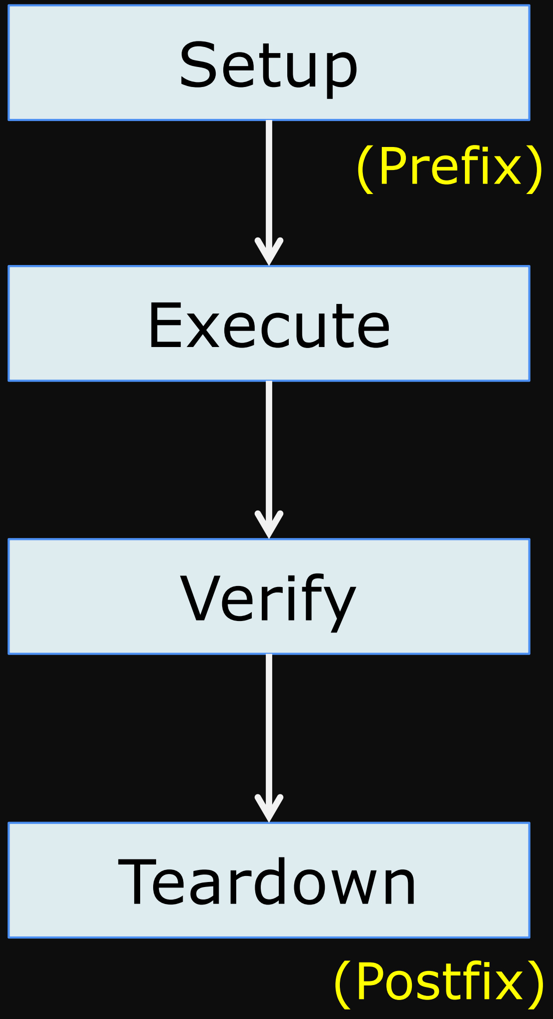
(Not much changed between Junit 4 and Junit 5 in test writing styles)



JUnit Tests

- For unit and integration testing
  - Entire object, part of an object (a method or some interacting methods), and interaction between several objects
- One test case in one test method
- A test class contains one or more test methods
- Test classes include
  - A collection of test methods
  - Method to set up the state before running each test (prefix)
  - Method to update the state after each test (postfix)
  - [Optional] Method to set up and update before and after all tests

Test Lifecycle



Initialize the test fixture

Interact with the subject under test

Compare the actual (observed) result of running the test with the expected result – using assertion(s)

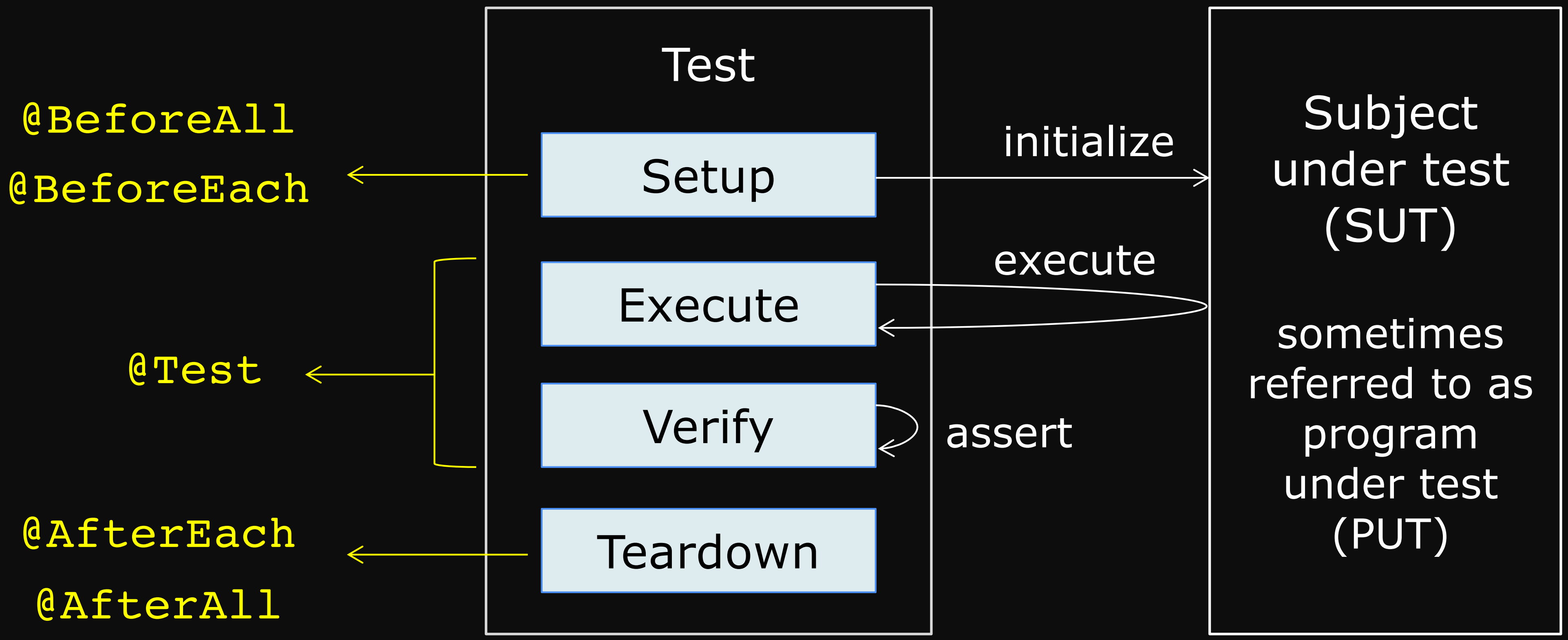
Release the test fixture to put the subject under test back into the initial state

Annotations

Use the methods of the *org.junit.jupiter.api* class
(Refer to Javadoc for a complete API)

JUnit 5 annotation	**Description**	**JUnit 4's equivalence**
`@BeforeEach`	Method executed before each `@Test` in the current class	`@Before`
`@AfterEach`	Method executed after each `@Test` in the current class	`@After`
`@BeforeAll`	Method executed before all `@Test` in the current class	`@BeforeClass`
`@AfterAll`	Method executed after all `@Test` in the current class	`@AfterClass`
`@Test`	Define a test method	`@Test`

Lifecycle and Annotations



Writing JUnit Tests (JUnit5)

- Download necessary jar files at *junit.org*
- Use the methods of the following classes

  org.junit.jupiter.api.AfterAll
  org.junit.jupiter.api.AfterEach
  org.junit.jupiter.api.BeforeAll
  org.junit.jupiter.api.BeforeEach
  org.junit.jupiter.api.Test
  org.junit.jupiter.api.Assertions

- Each test method
  - Checks a condition (assertion)
  - Reports to the test runner whether the test failed or succeeded
- The test runner uses the result to report to the user
- All of the methods return void

Test Class

```java
package test;

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

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class LifecycleTest
{
    @BeforeAll
    static void setUpBeforeClass() throws Exception
    {
        System.out.println("Setup all tests in the class");
    }

    @BeforeEach
    void setUp() throws Exception
    {
        System.out.println("Setup each test in the class");
    }

    @Test
    void testOne()
    {
        System.out.println("Test 1 -- be sure to use meaningful method name");
    }

    @Test
    void testTwo()
    {
        fail("Not yet implemented");
    }

    @AfterEach
    void tearDown() throws Exception
    {
        System.out.println("Teardown each test in the class");
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception
    {
        System.out.println("Teardown all tests in the class");
    }
}
```

Imports

Test class

Test method

Test method

JUnit / xUnit - Conventions

- Group related test methods in a single test class

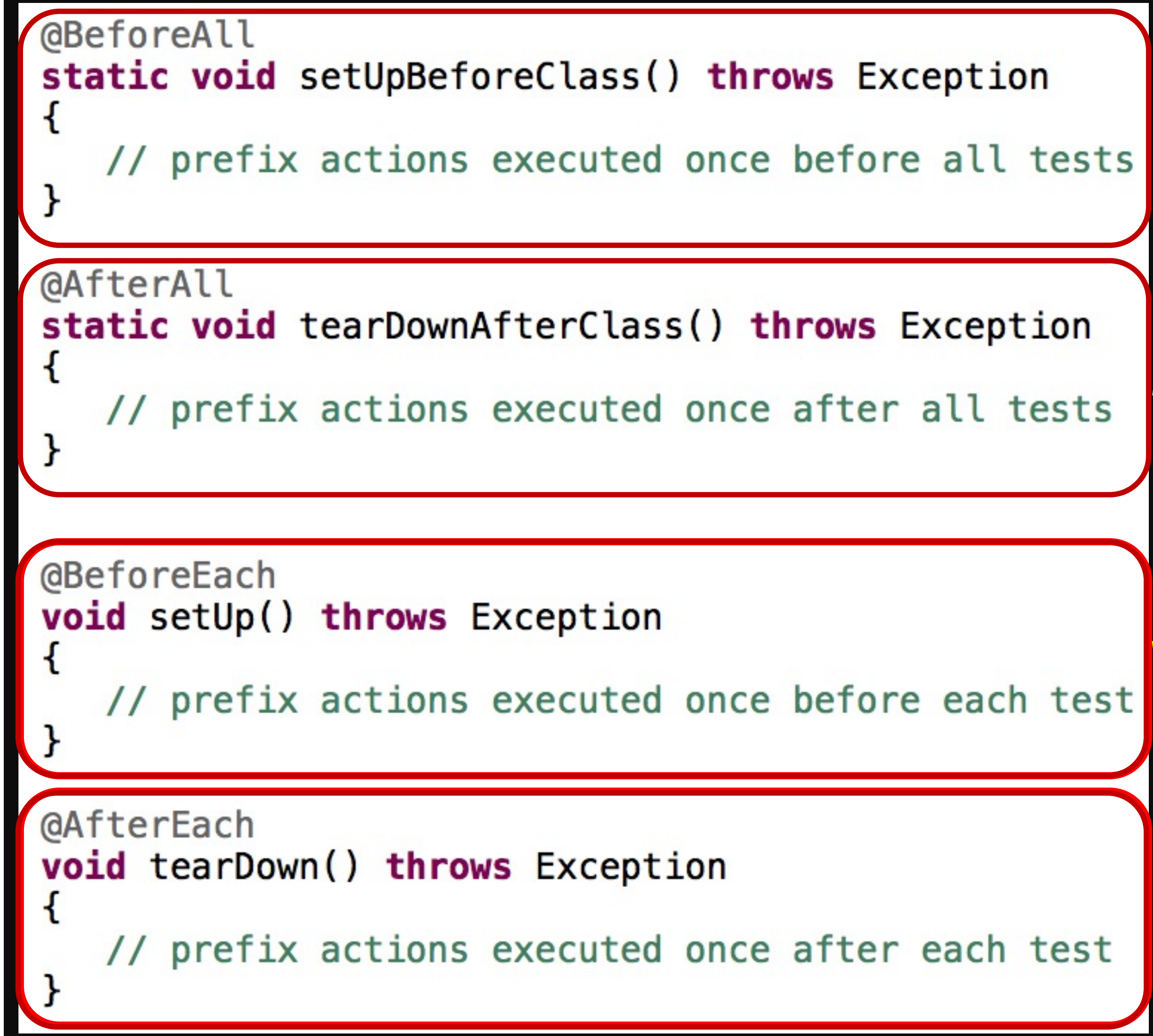
- The name of test packages/classes/methods should at least transmit:
  - The name of the subject under test (SUT) class
    - `TestArrayOperationsNumZero` or `ArrayOperationsNumZeroTest`
  - The name of the method or feature being tested
    - `TestArrayOperationsNumZero` or `ArrayOperationsNumZeroTest`
  - The purpose of the test case
    - `testNumZeroEmptyArray`

- It is common to prefix or suffix test classes with "Test" and prefix test methods with "test" (with or without "_")

JUnit Test Fixtures

- A test fixture is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (prefix values)
  - Reset values (postfix values)
- Different tests can use the objects without sharing the state
- Objects used in test fixtures should be declared as instance variables
- Objects should be initialized in a @BeforeEach method
- Objects can be deallocated or reset in an @AfterEach method

Prefix / Postfix Actions



```
@BeforeAll
static void setUpBeforeClass() throws Exception
{
   // prefix actions executed once before all tests
}

@AfterAll
static void tearDownAfterClass() throws Exception
{
   // prefix actions executed once after all tests
}

@BeforeEach
void setUp() throws Exception
{
   // prefix actions executed once before each test
}

@AfterEach
void tearDown() throws Exception
{
   // prefix actions executed once after each test
}
```

Initialize objects and variables that are used by more than one test

Reset objects and variables that are used by more than one test

Common Methods (JUnit 5)

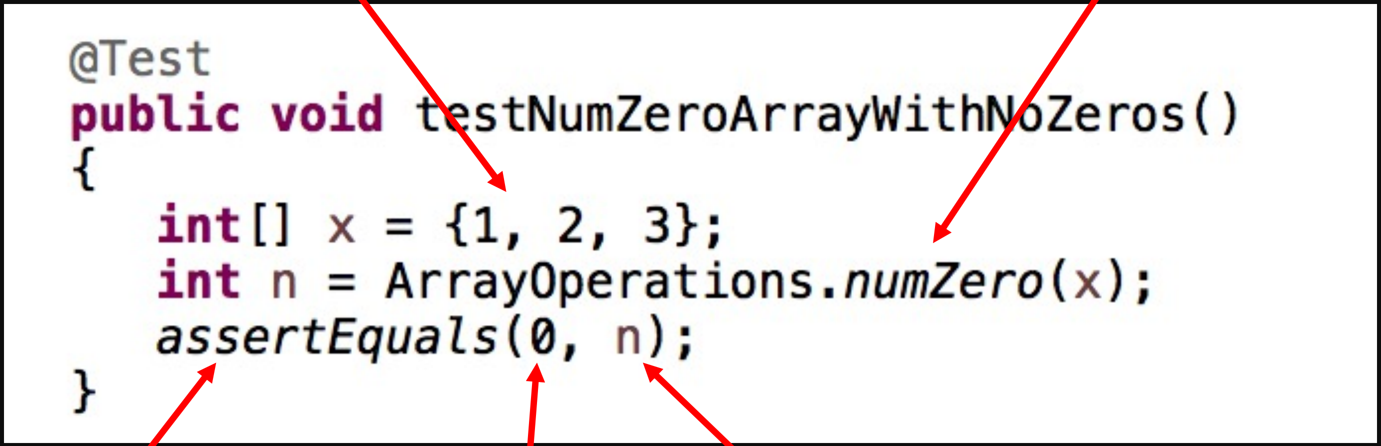
Assertions	Description
`assertTrue(boolean condition)`	Assert that a condition is true.
`assertTrue(boolean condition, String message)`	Assert that a condition is true. If the assertion is true, the string is ignored. Otherwise, the string is sent to the test engineer.
`assertEquals(Object expected, Object actual)`	Assert that two objects are equal.
`fail(String message)`	If a certain situation is expected when a certain section of code is reached, the string is sent to the test engineer. Often used to test exceptional behavior.

(Refer to Javadoc for a complete API)

JUnit – Test Methods

1) Setup test case values

2) Execute program under test



```
@Test
public void testNumZeroArrayWithNoZeros()
{
    int[] x = {1, 2, 3};
    int n = ArrayOperations.numZero(x);
    assertEquals(0, n);
}
```

expected

actual output

3) Assert expected vs. actual test outputs

JUnit – Test Methods

1) Setup test case values

2) Execute program under test

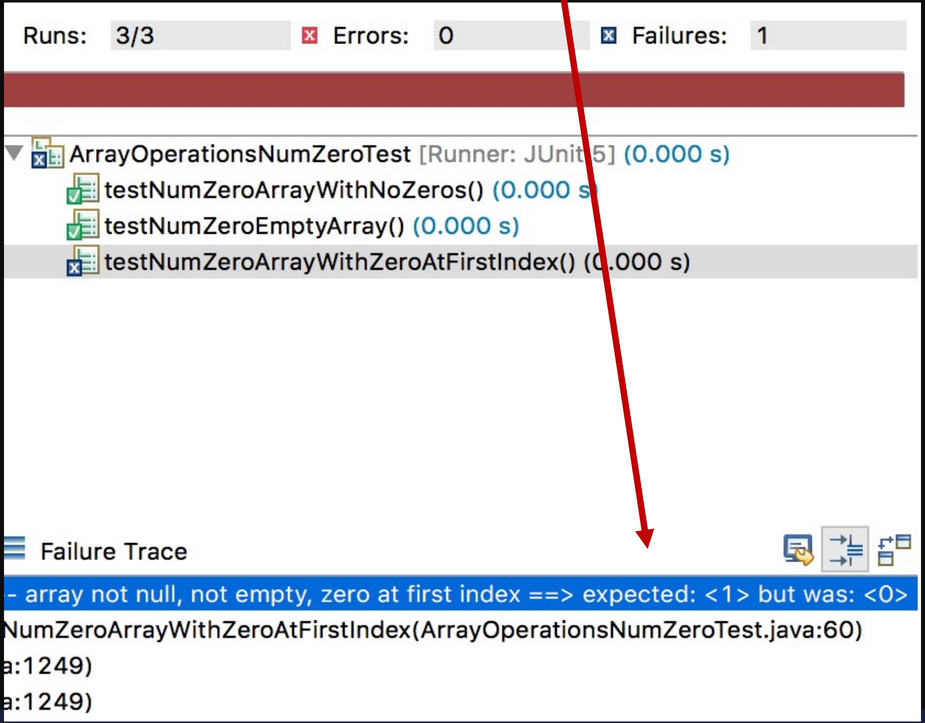
4) Printed if assert fails

```
@Test
public void testNumZeroArrayWithZeroAtFirstIndex()
{
   int[] x = {0, 2, 3};
   int n = ArrayOperations.numZero(x);
   assertEquals(1, n, "test -- array not null, not empty, zero at first index");
}
```

actual output

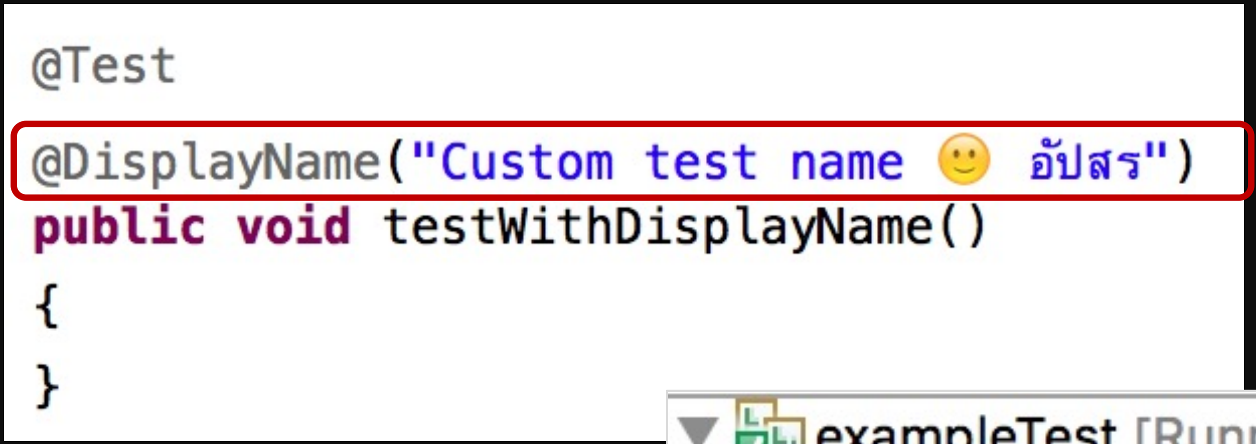
expected

3) Assert expected vs. actual test outputs

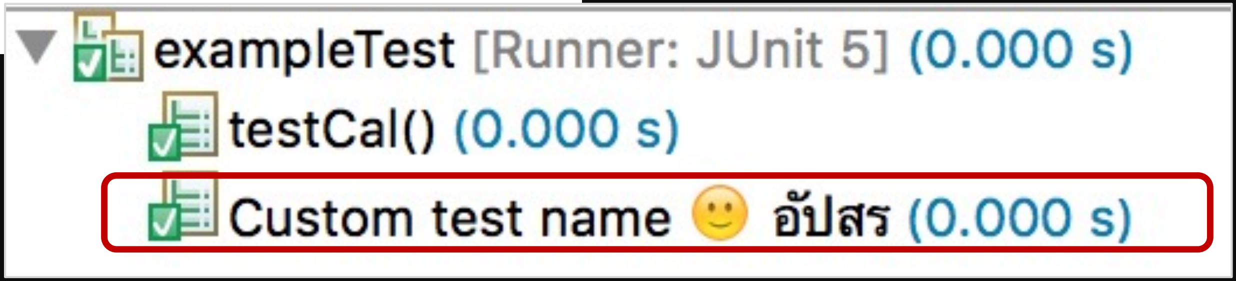


```
Runs: 3/3    Errors: 0    Failures: 1

ArrayOperationsNumZeroTest [Runner: JUnit 5] (0.000 s)
    testNumZeroArrayWithNoZeros() (0.000 s)
    testNumZeroEmptyArray() (0.000 s)
    testNumZeroArrayWithZeroAtFirstIndex() (0.000 s)

Failure Trace
- array not null, not empty, zero at first index ==> expected: <1> but was: <0>
NumZeroArrayWithZeroAtFirstIndex(ArrayOperationsNumZeroTest.java:60)
a:1249)
a:1249)
```

Display Names



```
@Test
@DisplayName("Custom test name 🙂 อัปสร")
public void testWithDisplayName()
{
}
```



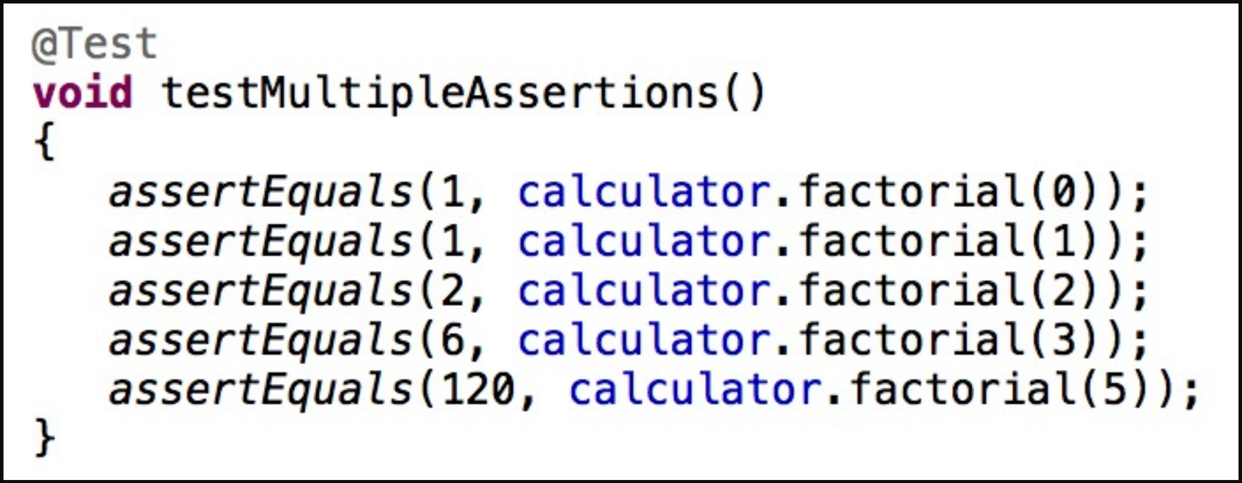
```
▼ exampleTest [Runner: JUnit 5] (0.000 s)
    testCal() (0.000 s)
    Custom test name 🙂 อัปสร (0.000 s)
```

@DisplayName annotation of the org.junit.jupiter.api.DisplayName class declares a custom display for a test class or a test method.

The name will be displayed by the test runners and reporting tools.

The name can contain spaces, special characters, and even emojis.

Multiple Assertions



```
@Test
void testMultipleAssertions()
{
   assertEquals(1, calculator.factorial(0));
   assertEquals(1, calculator.factorial(1));
   assertEquals(2, calculator.factorial(2));
   assertEquals(6, calculator.factorial(3));
   assertEquals(120, calculator.factorial(5));
}
```

In a test method with multiple assertions (written in a standard way), the first failure will be reported; the remaining assertions will not be executed and the test method is terminated.

Group of Assertions

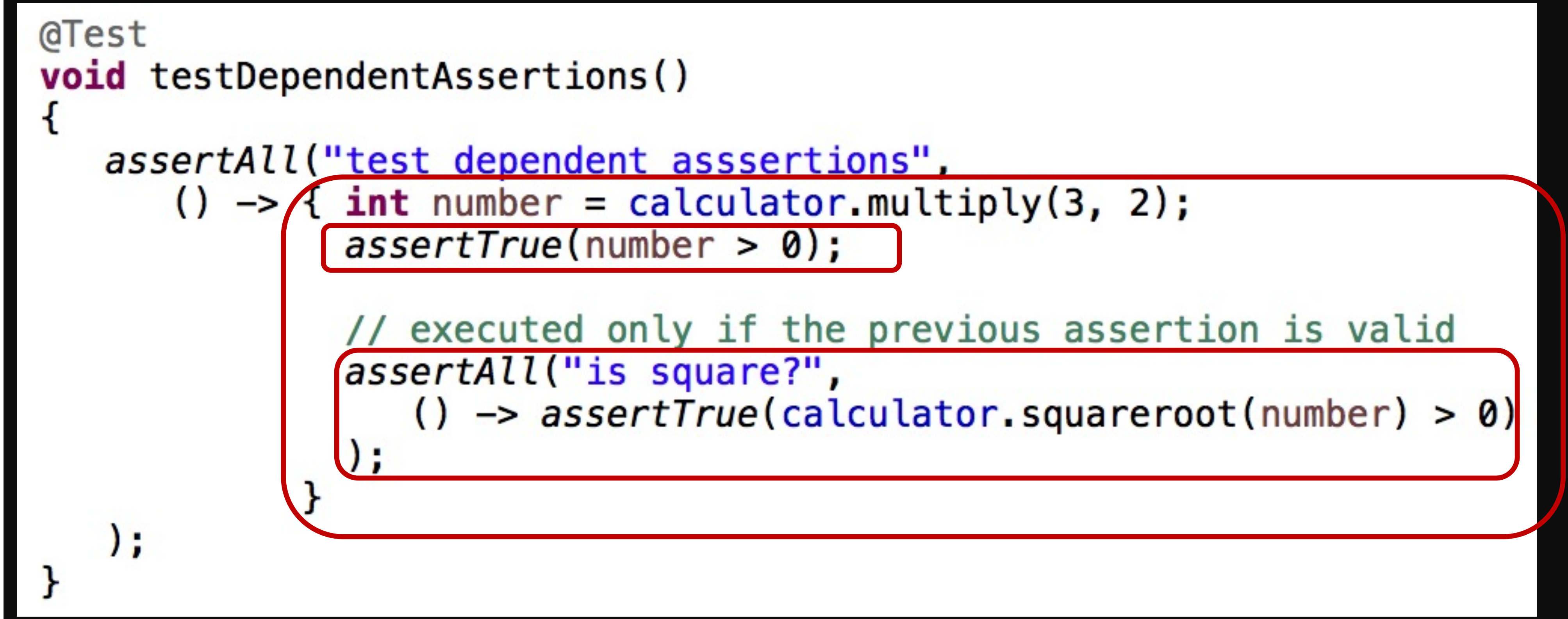
Label the assertion group

```
@Test
void testCalculatorOps()
{
   // In a test method with a grouped assertion,
   // all assertions are executed and all failures will be reported together.
   assertAll("test calculator with grouped assertions",
      () -> assertEquals(5, calculator.add(3, 2)),
      () -> assertEquals(6, calculator.multiply(3, 2))
   );
   // Note: this test method doesn't follow the general idea of "each test"
   // A more reasonable scenario to use grouped assertions may be to verify
   // things or constraints that are interrelated;
   // for example, to test a person object
   // -- verifying person.getFirstName() and person.getLastName()
}
```

assertAll method groups assertions at the same time.

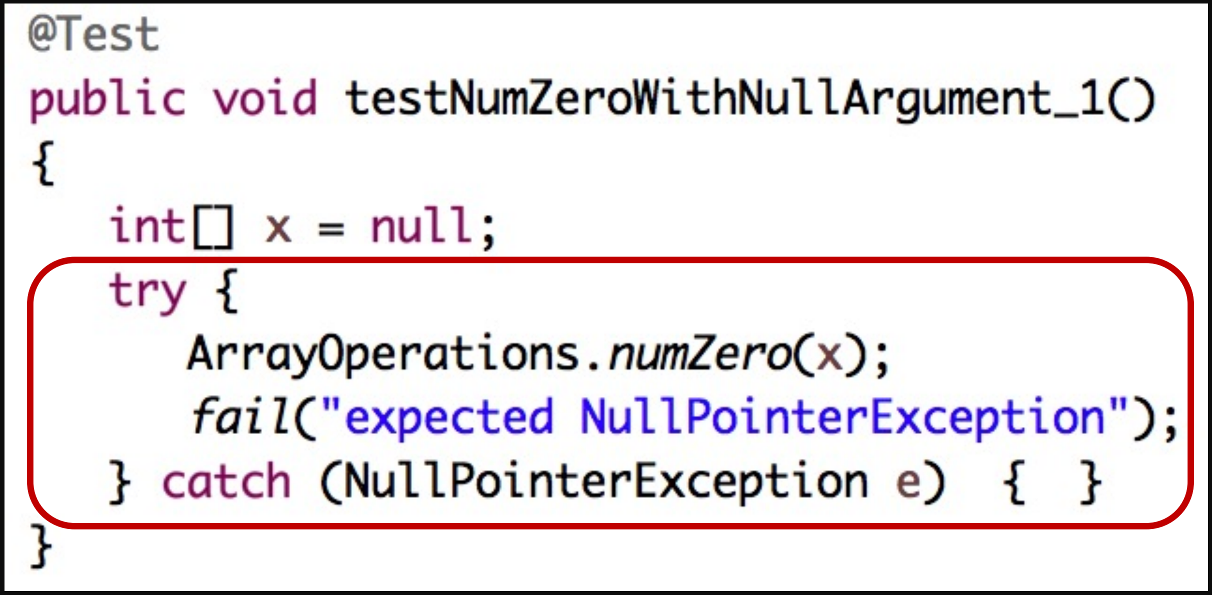
In a grouped assertion, all assertions are always executed, and any failures will be reported together.

Dependent Tests



```
@Test
void testDependentAssertions()
{
    assertAll("test dependent asssertions",
        () -> { int number = calculator.multiply(3, 2);
                assertTrue(number > 0);

                // executed only if the previous assertion is valid
                assertAll("is square?",
                    () -> assertTrue(calculator.squareroot(number) > 0)
                );
              }
    );
}
```

Exceptions as Expected Results



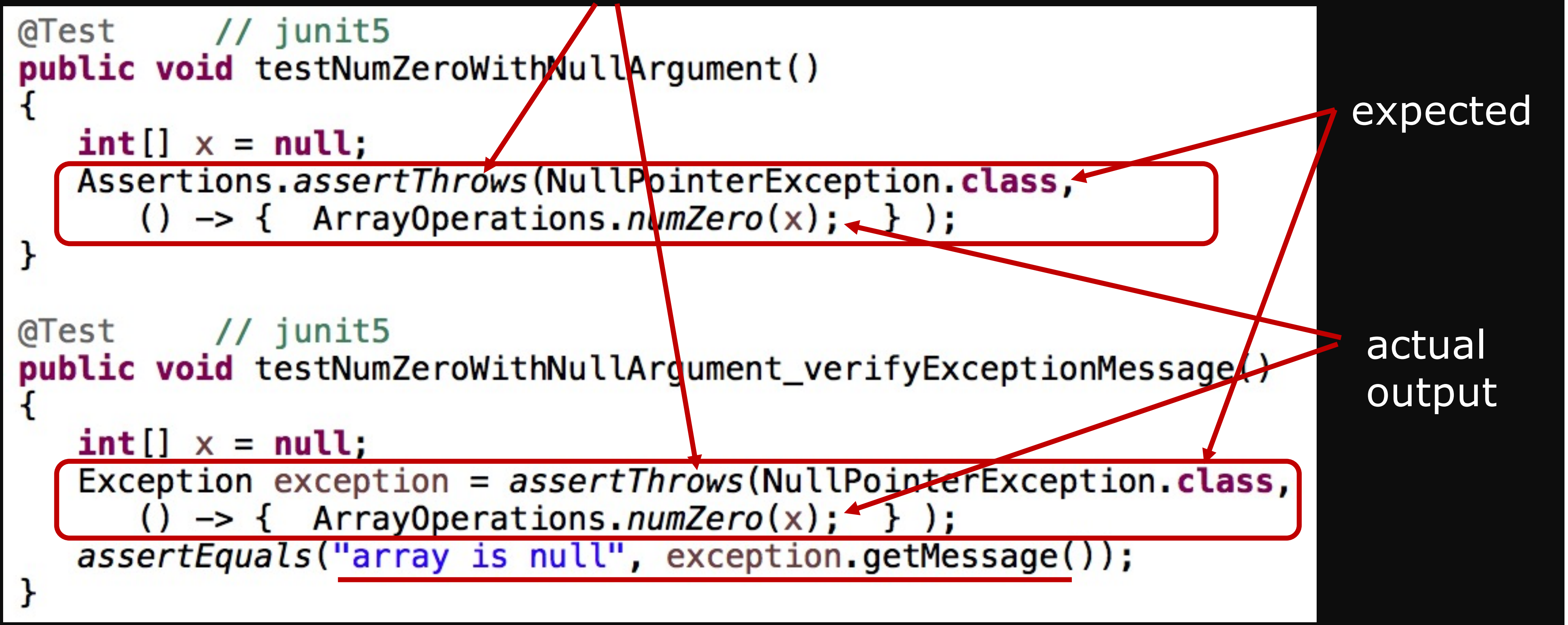
```
@Test
public void testNumZeroWithNullArgument_1()
{
   int[] x = null;
   try {
      ArrayOperations.numZero(x);
      fail("expected NullPointerException");
   } catch (NullPointerException e)  {  }
}
```

This pattern is more verbose and unnecessary in this case.

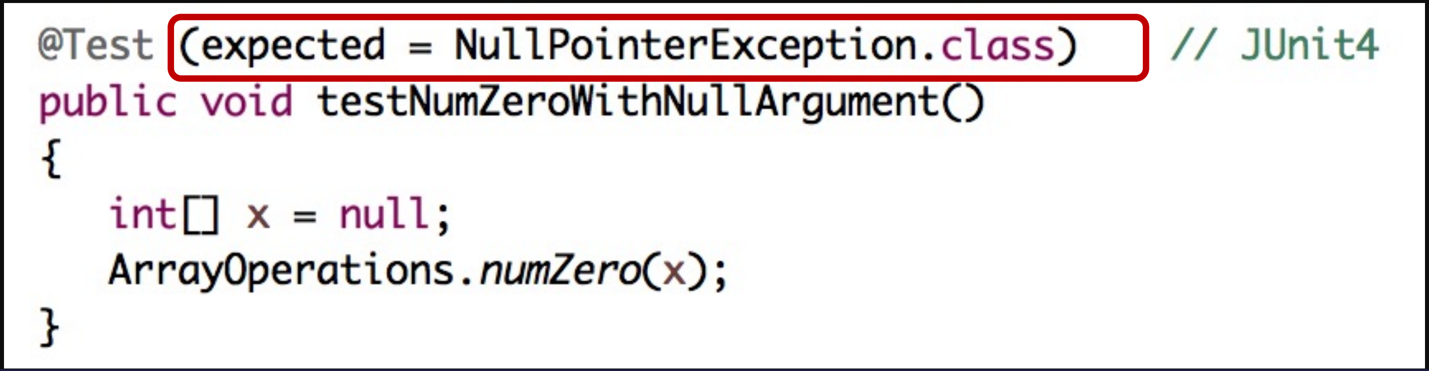
It is useful in situations when we wish to perform other assertions beyond the expected exception behavior

Exceptions as Expected Results

Verify if a given exception is raised using assertThrows



```
@Test      // junit5
public void testNumZeroWithNullArgument()
{
   int[] x = null;
   Assertions.assertThrows(NullPointerException.class,
      () -> {  ArrayOperations.numZero(x);  } );
}

@Test      // junit5
public void testNumZeroWithNullArgument_verifyExceptionMessage()
{
   int[] x = null;
   Exception exception = assertThrows(NullPointerException.class,
      () -> {  ArrayOperations.numZero(x);  } );
   assertEquals("array is null", exception.getMessage());
}
```



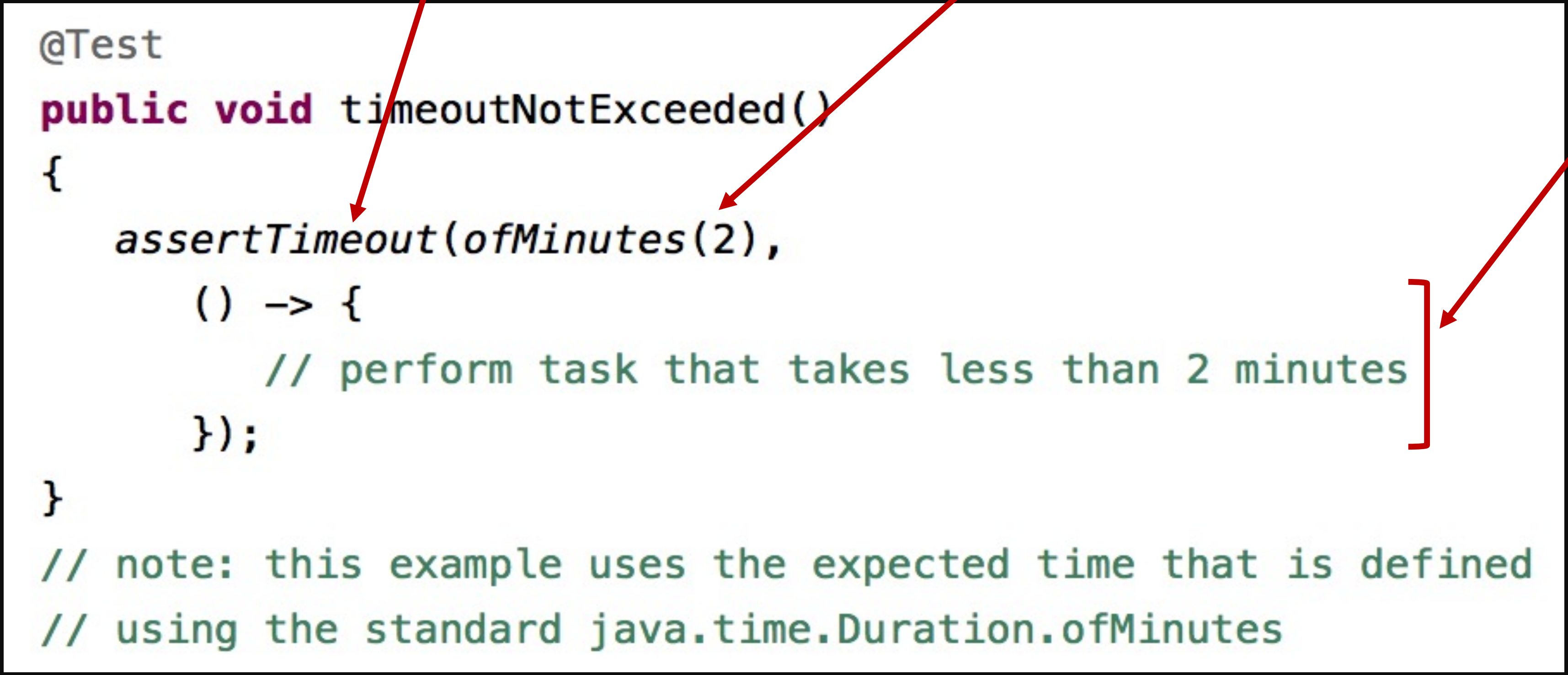
```
@Test (expected = NullPointerException.class)  // JUnit4
public void testNumZeroWithNullArgument()
{
   int[] x = null;
   ArrayOperations.numZero(x);
}
```

Asserting Timeouts

Verify if a given task or operation takes less then a certain period of time to complete using assertTimeout

expected

actual output



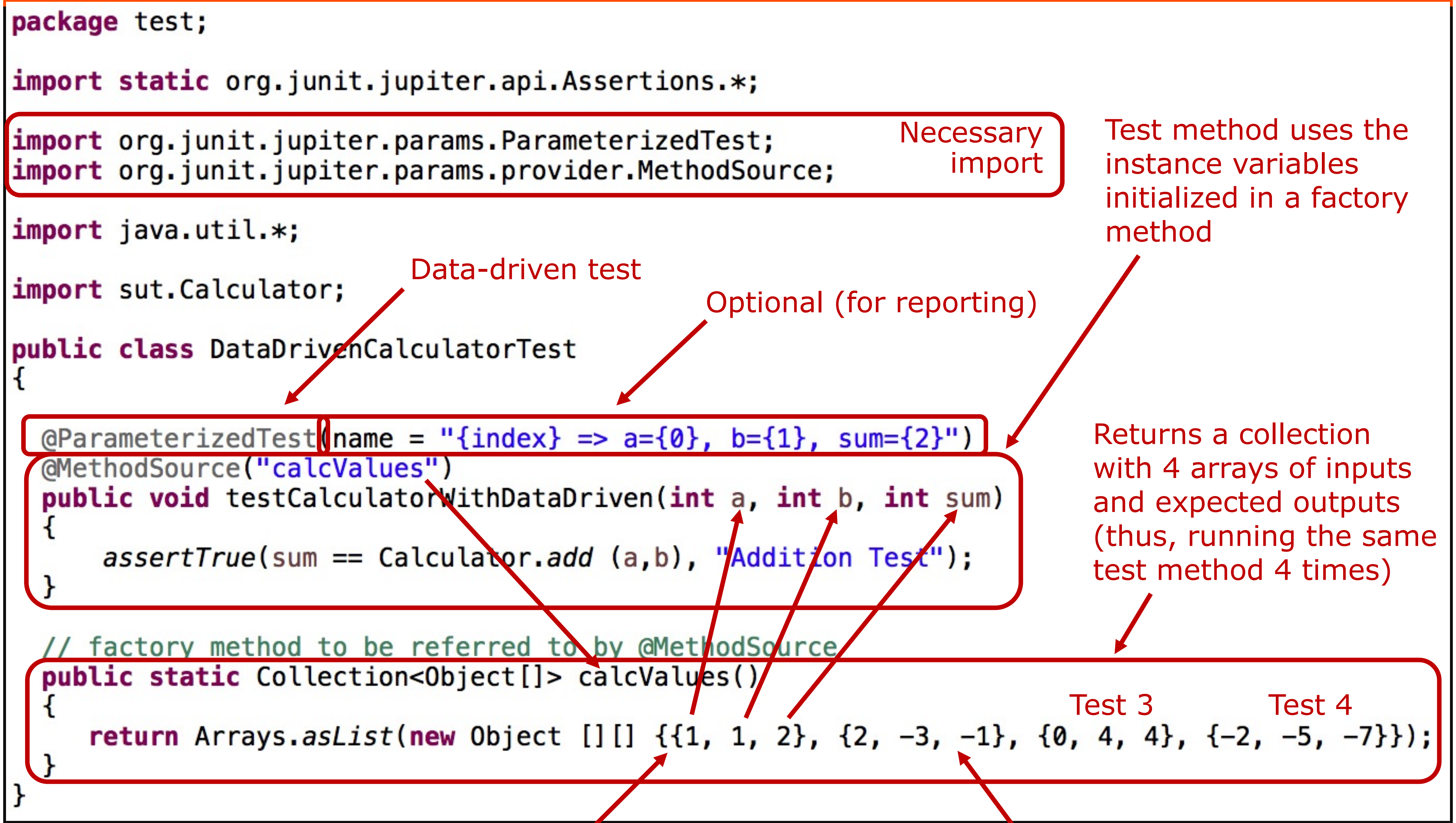
```
@Test
public void timeoutNotExceeded()
{
   assertTimeout(ofMinutes(2),
      () -> {
         // perform task that takes less than 2 minutes
      });
}
// note: this example uses the expected time that is defined
// using the standard java.time.Duration.ofMinutes
```

Data-Driven Tests

- Sometimes, the same test method needs to be run multiple times, with the only difference being the input values and the expected output

- Data-driven unit tests call a factory method for each collection of test values
  - Run each set of data values with the same tests
  - Implement data-driven testing with JUnit Parameterized mechanism

Example: JUnit5 Data-Driven Unit Test



```java
package test;

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

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.*;

import sut.Calculator;

public class DataDrivenCalculatorTest
{

  @ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
  @MethodSource("calcValues")
  public void testCalculatorWithDataDriven(int a, int b, int sum)
  {
     assertTrue(sum == Calculator.add (a,b), "Addition Test");
  }

  // factory method to be referred to by @MethodSource
  public static Collection<Object[]> calcValues()
  {
     return Arrays.asList(new Object [][] {{1, 1, 2}, {2, -3, -1}, {0, 4, 4}, {-2, -5, -7}});
  }
}
```

Necessary import

Test method uses the instance variables initialized in a factory method

Data-driven test

Optional (for reporting)

Returns a collection with 4 arrays of inputs and expected outputs (thus, running the same test method 4 times)

Test 3

Test 4

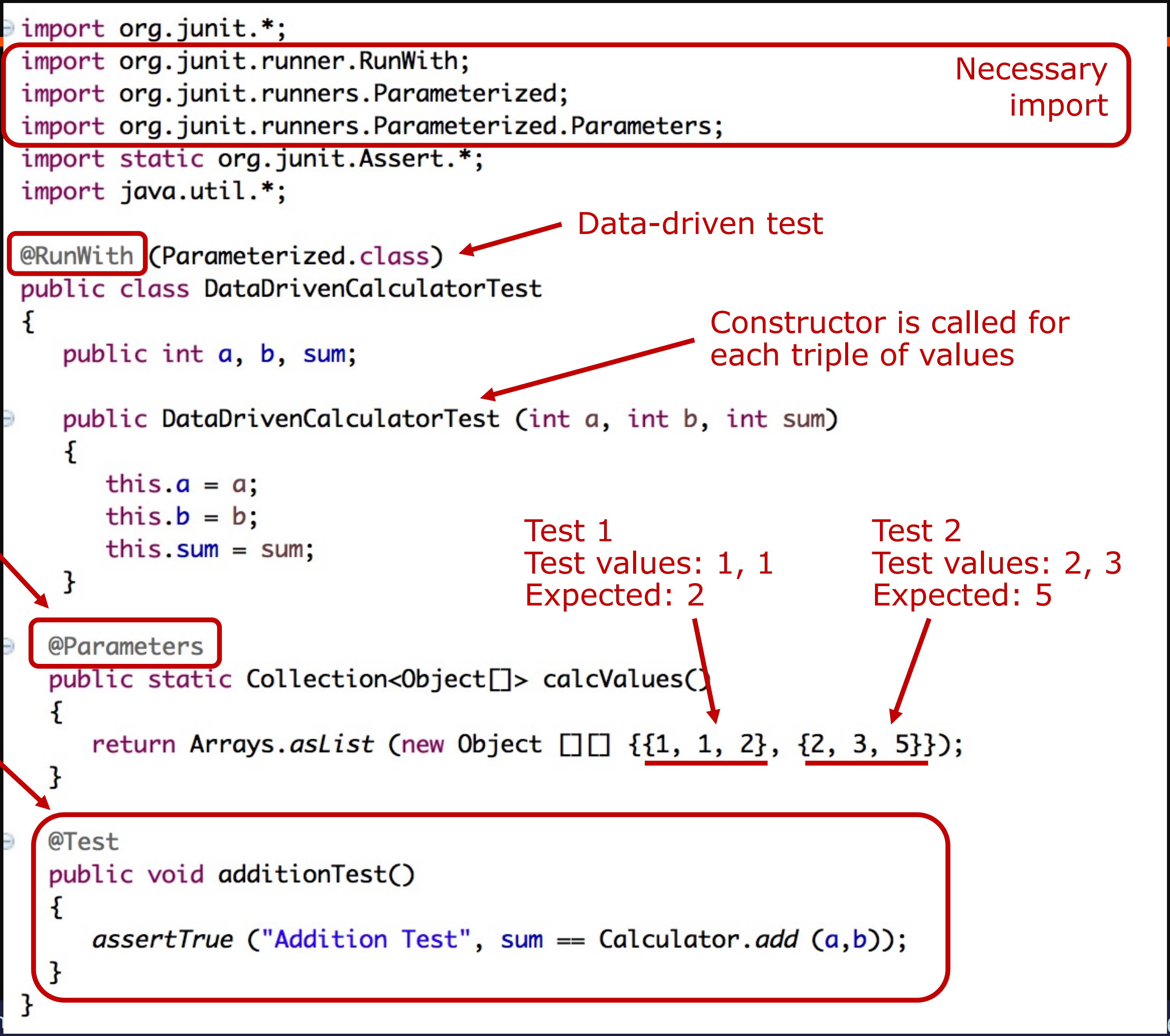
Test 1 Test values: 1, 1
Expected: 2

Test 2 Test values: 2, -3
Expected: -1

Example: JUnit4 Data-Driven Unit Test

Returns a collection with 2 arrays of inputs and expected outputs (thus, call the constructor twice)

Test method uses the instance variables initialized in the constructor call



```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

@RunWith (Parameterized.class)
public class DataDrivenCalculatorTest
{
   public int a, b, sum;

   public DataDrivenCalculatorTest (int a, int b, int sum)
   {
      this.a = a;
      this.b = b;
      this.sum = sum;
   }

  @Parameters
  public static Collection<Object[]> calcValues()
  {
     return Arrays.asList (new Object [][] {{1, 1, 2}, {2, 3, 5}});
  }

  @Test
  public void additionTest()
  {
     assertTrue ("Addition Test", sum == Calculator.add (a,b));
  }
}
```

Necessary import

Data-driven test

Constructor is called for each triple of values

Test 1
Test values: 1, 1
Expected: 2

Test 2
Test values: 2, 3
Expected: 5

Wrap-up

- Automate as much as possible to make testing efficient and effective
- Test frameworks provide very simply ways to automate our test
- Data-driven testing can suffer from a combinatorial explosion in the number of tests (cross-product of the possible values for each of the parameters in the unit tests)
- Test automation is not "silver bullet" .. It does not solve the hard problem of testing **"What test values to use?"**
- **"What test values to use?"** – solved by test design .. The purpose of **test criteria**

What's Next?

- Coverage-based test design