More JUnit

CS 3250
Software Testing

[Ammann and Offutt, “Introduction to Software Testing,” Ch. 3]
[https://junit.org/junit5/docs/current/user-guide/]
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 {
    @BeforeEach
    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");
    }
}
JUnit – Test Methods

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

2) Execute program under test

3) Assert expected vs. actual test outputs
JUnit – Test Methods

1) Setup test case values
2) Execute program under test
3) Assert expected vs. actual test outputs
4) Printed if assert fails

```java
@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");
}
```
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
def setUpBeforeClass() throws Exception:
    # prefix actions executed once before all tests

@AfterAll
def tearDownAfterClass() throws Exception:
    # prefix actions executed once after all tests

@BeforeEach
def setUp() throws Exception:
    # prefix actions executed once before each test

@AfterEach
def 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.
@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.
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.
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

```java
@Test
void testDependentAssertions()
{
    assertAll("test dependent assertions",
               () -> {
                   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

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`

```java
@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 than a certain period of time to complete using `assertTimeout`.

```java
@Test
gpublic 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: JUnit4 Data-Driven Unit Test

```java
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));
    }
}
```

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

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

Test method uses the instance variables initialized in the constructor call
Wrap-up

• Automate as much as possible to make testing efficient as well as 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” however .. It does not solve the hard problem of testing “What test values to use?”

• This is test design .. The purpose of test criteria

What’s Next?

• Putting testing first