EN.601.422 / EN.601.622

Software Testing & Debugging

Test Automation

- To use Software to design, generate, execute, and determine the outcome (i.e., compare actual vs. expected output) of test cases and/or control such or related activities.
- Revenue tasks vs. Excise tasks
  - Some tasks need human wisdom, intelligence, judgement, domain knowledge etc.
  - Some tasks can be automated following a well-defined process/algorithm/standard.

Software Testability

- **Testability**: how testable the software is
- **Two important factors to gauge testability**:
  - **Observability**: how easy it is to observe the behavior of a software in terms of its outputs and/or effects on the environment
  - **Controllability**: How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

Test Case Definition

- **Test case values**: input values necessary to complete the execution of the test case
- **Prefix values**: inputs necessary to establish the appropriate state so that the test case can be properly run
- **Postfix values**: inputs to be sent to the software after the test case has run
- **Expected Results**: The correct (i.e., expected) output for a given test case
- Test case: A test case is composed of test case values, prefix and postfix values, and expected results
  - A test case may contain more information pieces such as test case ID, test case summary test case author, last status (pass/fail), actual result etc., but at the very least a test case must include test case values and expected results
- Test Set (aka Test Suite): a set of test cases
- Test Framework (aka Test Tool): software tool that facilitates testing activities

JUnit

- Perhaps the most popular/well-known test framework
- Free software created by Kent Beck and Eric Gamma
- Mostly used for unit testing and integration testing
- Can be launched either from command-line or IDEs (e.g., Eclipse, IntelliJ, NetBeans etc.)
- xUnit frameworks
  - CUnit, CppUnit, PyUnit, JSUnit, NUnit, EUnit, RUnit and many more

JUnit

- One or more *test classes* where each test class contains:
  - test cases written as *test methods*
  - methods to setup program state before each test method is run (i.e., prefix values)
  - methods to update state after each test method is run (i.e., postfix values)
- A variety of *assert* methods available to check the actual result produced by a test case against expected output

Writing Tests

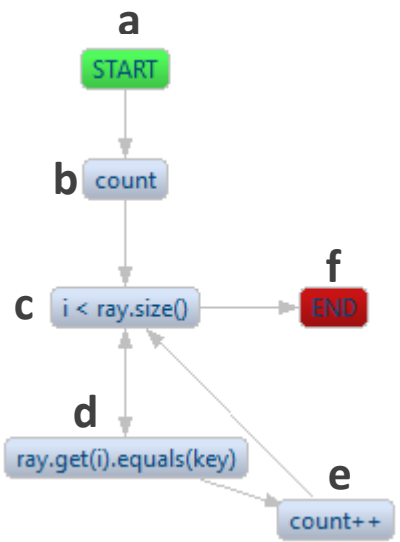
```
class MathUtils {
    // …
    public static int add(int a, int b) {
        return a + b;
    }
    // more math methods
}


class MathUtilsTest { // test class contains test methods (i.e., test cases)
  @Test
  public void testAdd() { // a test method corresponding to a single test case
    MathUtils mu = new MathUtils();
    assertEquals(8, mu.add(2, 6));
  }
}
```

Example

- Let's write tests to achieve path coverage of up to depth 2 for the *countOf* method:

```
public static int countOf(ArrayList<Integer> ray, int key) {
   int count = 0;
   for (int i = 0; i < ray.size(); ++i) {
      if (ray.get(i).equals(key))  {
         count++;
      }
   }
   return count;
}
```



Paths to cover

abcf
abcdcf
abcdecf
abcdcdcf
abcdcdecf
abcdecdecf
abcdecdcf

The Test Class for ArrayUtils

```
public class ArrayUtilsWbTest {
  @Test // abcf
  public void testCountOfEmptyArr() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    assertTrue(ArrayUtils.countOf(ray, 2) == 0);
  }
  @Test // abcdcf
  public void testCountOfArrSizeOneKeyNotExists() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 0);
  }
  @Test // abcdecf
  public void testCountOfArrSizeOneKeyExists() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdcdcf
  public void testCountOfArrSizeTwoKeyNotExists() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 3), 0);
  }
  @Test // abcdecdcf
  public void testCountOfArrSizeTwoKeyExistsFirst() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 1);
  }
  @Test // abcdcdecf
  public void testCountOfArrSizeTwoKeyExistsSecond() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdecdecf
  public void testCountOfArrSizeTwoKeyExistsFirstSecond() {
    ArrayList<Integer> ray = new ArrayList<Integer>();
    ray.add(1);
    ray.add(1);
    assertEquals(ArrayUtils.countOf(ray, 1), 2);
  }

} end of class ArrayUtilsWbTest
```

Utilizing Test Fixture

```
public class ArrayUtilsWbTest {
  ArrayList<Integer> ray;
  @BeforeEach
  public static void setUp() {
    ray = new ArrayList<Integer>();
  }
  @Test // abcf
  public void testCountOfEmptyArr() {
    assertTrue(ArrayUtils.countOf(ray, 2) == 0);
  }
  @Test // abcdcf
  public void testCountOfArrSizeOneKeyNotExists() {
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 0);
  }
  @Test // abcdecf
  public void testCountOfArrSizeOneKeyExists() {
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdcdcf
  public void testCountOfArrSizeTwoKeyNotExists() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 3), 0);
  }
  @Test // abcdecdcf
  public void testCountOfArrSizeTwoKeyExistsFirst() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 1);
  }
  @Test // abcdcdecf
  public void testCountOfArrSizeTwoKeyExistsSecond() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdecdecf
  public void testCountOfArrSizeTwoKeyExistsFirstSecond() {
    ray.add(1);
    ray.add(1);
    assertEquals(ArrayUtils.countOf(ray, 1), 2);
  }
} end of class ArrayUtilsWbTest
```

Utilizing Test Fixture

```
public class ArrayUtilsWbTest {
  ArrayList<Integer> ray;
  @BeforeAll
  public static void setUp() {
    ray = new ArrayList<Integer>();
  }
  @AfterEach
  public static void tearDown() {
    ray.clear();
  }
  @Test // abcf
  public void testCountOfEmptyArr() {
    assertTrue(ArrayUtils.countOf(ray, 2) == 0);
  }
  @Test // abcdcf
  public void testCountOfArrSizeOneKeyNotExists() {
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 0);
  }
  @Test // abcdecf
  public void testCountOfArrSizeOneKeyExists() {
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdcdcf
  public void testCountOfArrSizeTwoKeyNotExists() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 3), 0);
  }
  @Test // abcdecdcf
  public void testCountOfArrSizeTwoKeyExistsFirst() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 1), 1);
  }
  @Test // abcdcdecf
  public void testCountOfArrSizeTwoKeyExistsSecond() {
    ray.add(1);
    ray.add(2);
    assertEquals(ArrayUtils.countOf(ray, 2), 1);
  }
  @Test // abcdecdecf
  public void testCountOfArrSizeTwoKeyExistsFirstSecond() {
    ray.add(1);
    ray.add(1);
    assertEquals(ArrayUtils.countOf(ray, 1), 2);
  }
} end of class ArrayUtilsWbTest
```

Test Instance Lifecycle

- In order to allow individual test methods to be executed in isolation and to avoid unexpected side effects due to mutable test instance state, JUnit creates a new instance of each test class before executing each *test method:*
  - Can override this by: **@TestInstance(Lifecycle.PER_CLASS)**

Exceptions

```
/** counts how many times key occurs in ray
 * @param ray the ArrayList to be searched
 * @param key the key value to search for
 * @return the count of how many elements in ray are equal to key
 * @throws NullPointerException (NPE) if ray is null
*/
public static int countOf(ArrayList<Integer> ray, int key) {
  int count = 0;
  for (int i = 0; i < ray.size(); ++i) {
    if (ray.get(i).equals(key))  {
      count++;
    }
  }
  return count;
}
```

Exceptions

```
@Test
public void testCountOfRayNull() {
  ray = null;
  try {
    ArrayUtils.countOf(ray, 2);
  }
  catch (NullPointerException e) {
    return;
  }
  fail("null must fail");
}
```

```
@Test
public void testCountOfRayNull() {
  ray = null;
  assertThrows(NullPointerException.class, () -> {
    ArrayUtils.countOf(ray, 2);
  });
} //assertThrows is only supported in JUnit 5
```

Parameterized Tests

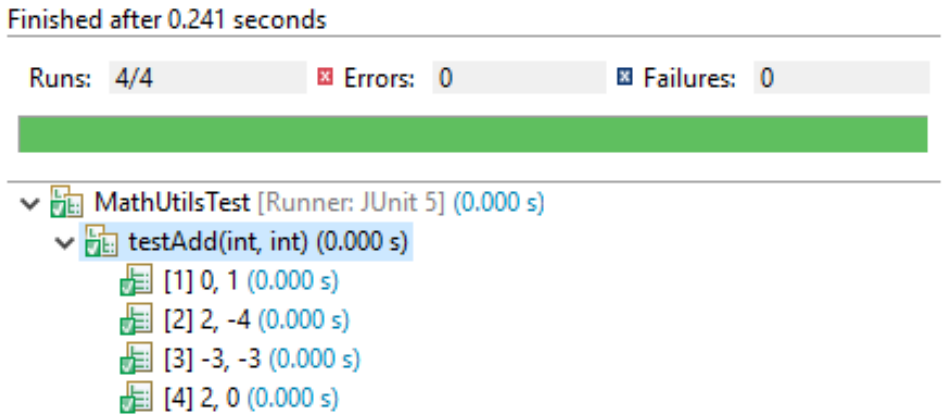
- A common unit testing pattern is to test a single method with different input values
  - One way is to write a separate test method for each test case
  - lots of duplicate code

Parameterized Testing

Parameterized Tests

- Let's use parameterized testing to test the "add" method:
- Instead of writing four different test cases, write one!

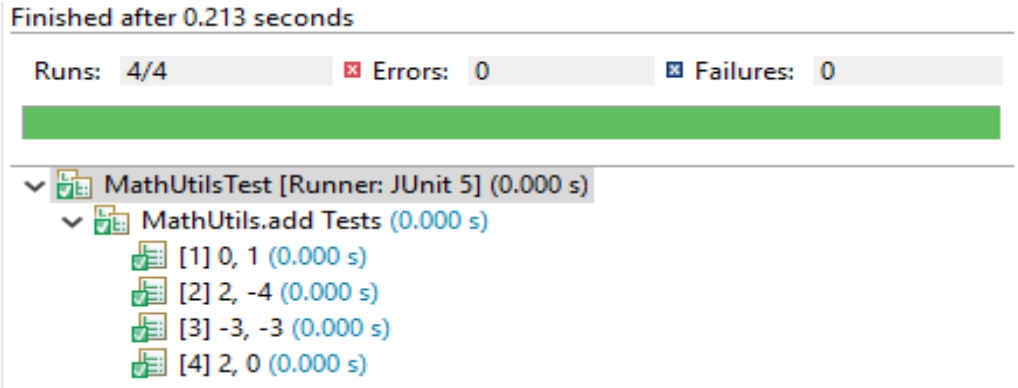
```
@ParameterizedTest
@CsvSource({"0, 1", "2, -4", "-3, -3", "2, 0"})
public void testAdd(int a, int b) {
  MathUtils.add(a, b);
}
```



Parameterized Tests with Expected Output

- Let's use parameterized testing to test the "add" method:
- Instead of writing four different test cases, write one!

```
@DisplayName("MathUtils.add Tests")
@ParameterizedTest
@CsvSource({"0, 1, 1", "2, -4, -2", "-3, 3, 0", "-2, -2, -4"})
public void testAdd(int a, int b, int expected) {
  assertEquals(expected, MathUtils.add(a, b));
}
```

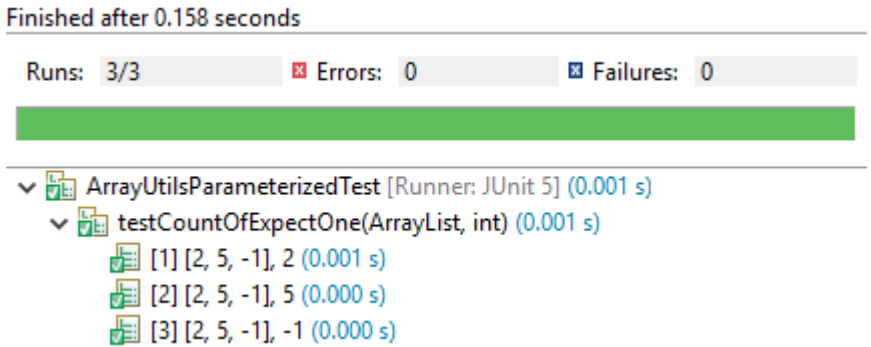


Parameterized Tests

- Now, let's use parameterized testing to test the "countOf" method for few inputs that we expect to return value of 1:

```
@ParameterizedTest
@MethodSource("createListAndKey")
public void testCountOfExpectOne(ArrayList<Integer> ray, int key) {
  assertEquals(1, ArrayUtils.countOf(ray, key));
}

private static Stream<Arguments> createListAndKey() {
  ArrayList<Integer> ray0 = new ArrayList<Integer>();
  ray0.add(2);
  ray0.add(5);
  ray0.add(-1);
  return Stream.of(
    Arguments.of(ray0, 2),
    Arguments.of(ray0, 5),
    Arguments.of(ray0, -1));
}
```



@Nested

- Can have nested test classes
- Useful to group test cases

```
@DisplayName(“Tree Test")
class TreeTest {
      // …
      @Nested
      @DisplayName(“Add new node")
      class AddNode {
            // …
      }
      // …
      @Nested
      @DisplayName(“Remove a node")
      class RemoveNode {
            // …
      }
      // …
}
```

Tag and Filtering

- Tags are used to filter which tests are executed for a given test plan. For example, a development team may tag tests with values such as "fast", "slow", etc. and then supply a list of tags to be used for the current test plan
- Can give test methods and/or test classes one or more tag names

```
@Tag(“all")
class ClassATest {
    @Test
    @Tag("development")
    @Tag("production")
    void testCaseA(TestInfo testInfo) {
    }
}
```

@RepeatedTest

- Run a test more than once

```
@RepeatedTest(10)
public void testRepeat() {
    // …
}
```

Order of Execution

- By default in JUnit, the test methods could run in any order
  - When unit testing, test cases must be independent of each other
- Possible test case execution orderings:
  - **Alphanumeric**: sorts test methods alphanumerically based on their names and formal parameter lists.
  - **OrderAnnotation**: sorts test methods numerically based on values specified via the **@Order** annotation.
  - **Random**: orders test methods pseudo-randomly and supports configuration of a custom seed.

Order of Execution

```
@TestMethodOrder(OrderAnnotation.class)
class OrderOfExecutionDemo {

    @Test
    @Order(1)
    void nullValues() {
        // perform assertions against null values
    }

    @Test
    @Order(2)
    void emptyValues() {
        // perform assertions against empty values
    }

    @Test
    @Order(3)
    void validValues() {
        // perform assertions against valid values
    }

}
```

Order of Execution

```
@TestMethodOrder(Alphanumeric.class)
class OrderOfExecutionDemo {

    @Test
    void testA() {
        // run this first
    }

    @Test
    void testB() {
        // run this next
    }

    @Test
    void testC() {
        // and run this last
    }
}
```

@Timeout

- Allows a test to fail if execution time exceeds the specified limit

```
class TimeoutDemo {

    @BeforeEach
    @Timeout(5)
    void setUp() {
        // fails if execution time exceeds 5 seconds
    }

    @Test
    @Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
    void failsIfExecutionTimeExceeds100Milliseconds() {
        // fails if execution time exceeds 100 milliseconds
    }

}
```

Assumptions

- Used to validate assumptions before/after test execution

```
public class TestAssumptions {

    @Test
    public void assumeThatFileSeparatorTest(){
        // only execute the test if OS is Windows
        assumeTrue(File.separatorChar == '\\');
        // TEST
    }

    @Test
    public void assumeNotNullTest(){
        Person p = Department.getHead();
       // only execute the test if the head is  not null
        assumeFalse(p == null);
       // TEST
    }

    @Test
    public void assumeServerIsRunningTest(){
        boolean isServerRunning = Server.ping();
       // Only execute the test if server is up
        assumeTrue(isServerRunning);
        // TEST
    }
    @Test
    public void assumeUserIsAliTest(){
       // only execute the test if signed
          in as ali
      assumeTrue(System.getenv("USERNAME")
                                .equals("ali"));
        // TEST
    }
}
```

@Disabled

- Used to disable a test case
  - useful when you temporarily want to ignore some test cases

```
class DisabledTestsDemo {

    @Disabled("Disabled until bug #42 has been resolved")
    @Test
    void testWillBeSkipped() {
    }

    @Disabled("Disabled until feature #23 is implemented")
    @Test
    void testWillBeExecutedLater() {
    }

    // ...

}
```