Unit Testing with JUnit

Unit Testing

- Evaluating small pieces of code to make sure they function correctly.
- In object-oriented languages, the pieces are often classes and methods.

Benefits of Unit Testing

- Automated testing
- Reliable testing
- Forces an evaluation of the logic of the code
- Preserved and documented test cases to use with future modifications
- Puts a formal structure around testing

JUnit

- An open source testing framework that runs repeatable, automated tests
- Use *annotations* to flag methods for testing
- Use *assertions* to write tests that detect bugs
- JUnit5
  - https://junit.org/junit5/
  - https://junit.org/junit5/docs/current/user-guide/
  - https://junit.org/junit5/docs/5.0.1/api/index.html?org/junit/jupiter/api/

Using JUnit

- Create your class
- Create a test case class
  - Create one or more test methods for each method.

    @Test

    testNameOfMethod()
- Test valid and invalid inputs
- Test border conditions and special cases
  - Example: positive/negative numbers, min/max numbers, empty/singleton lists
- Run your test
  - Most IDEs have graphical test runners (including eclipse, NetBeans, and IntelliJ)

JUnit Method Annotations

- @Test- the method can be run as a test case
- @BeforeEach- method is executed before **each** test method
- @BeforeAll- static method is executed **once** before *all* tests
- @AfterEach- method is executed after **each** test method
- @AfterAll- static method is executed **once** after *all* tests
- @Disable- temporarily disable a particular test (use with @Test to create a test method you want to skip for now)

JUnit Assertions

- assertEquals(expected, actual, threshold)
    - version for each primitive type and Object
    - uses the .equals method on Objects and Double.equals(…)/Float.equals(…)
- assertTrue(boolean) and assertFalse(boolean)
- assertNotNull(Object) and assertNull(Object)
- assertSame(Object, Object) and assertNotSame(Object, Object)
    - uses == on Objects
- assertArrayEquals(expected, actual)
    - version for arrays of each primitive type and arrays of Objects

- All methods take an optional last parameter: String message

Using JUnit5

- Most IDEs have built-in support for JUnit5.
- Imports:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.*;
import org.junit.jupiter.params.provider.*;
```

Example Method Structure

```
@Test
public void testMyMethod() {
        create the necessary variables
        invoke myMethod
        gather the result
        make an assertion
}
```

Practice

- Review the NumberUtils and NumberUtils classes.
- Review unit tests for the BankAccount’s deposit and withdraw methods.
- Review unit tests for the add and drop methods in the Course class.

Parameterized Tests

- Allows you to run the same test case with different inputs
- Benefit: Reduces repeated/duplicated code
- Benefit: Group similar tests together
- Be careful not to group together tests that are really testing different things!

- Tag method with @ParameterizedTest

Specifying Parameters with @ValueSource

- For test methods with a single parameter of type String, int, long, or double, you can use @ValueSource annotation

- Example:
```
    @ParameterizedTest
    @ValueSource(ints = {1, -1, 0})
    public void test(int value) {
       // test of value
       // assertion
    }
```

- Name in the annotation is: strings, ints, longs, or doubles
- You can add a @NullSource annotation to test for null.

Specifying Parameters with @CsvSource

- You can specify different types of parameters in a comma-separated list with the @CsvSource annotation.
- Example:

```
@ParameterizedTest
@CsvSource({
    "1, false, 'hello'",
    "2, true, 'bye'"})
public void test(int number, boolean status, String word) {
     // test and assertion
}
```

Specifying Parameters with @MethodSource

- You can also specify different types of parameters by creating them as a Stream<Argument> in a method.
- Example:

```
@ParameterizedTest
@MethodSource("createValues")
public void test(int value, boolean status, String word) {
   // test and assertion
}

private static Stream<Arguments> createValues() {
   return Stream.of(
      Arguments.of(1, false, "hello"),
      Arguments.of(2, false, "bye"),
      Arguments.of(3, true, "later"));
}
```

Specifying Parameters- Displaying Name

- You can modify the name displayed using the parameters.

```
@ParameterizedTest(name = "value={0} with status={1} for word={2}")
@MethodSource("createValues")
public void test(int value, boolean status, String word) {
     // test and assertion
}
```

Practice

- Review the two versions of the parameterized bank account tester.
- Review the parameterized triangle tester.

Testing for Exceptions

- You can (and should!) test whether expected exceptions are properly thrown.
- Use assertThrows(ExceptionClass, Executable)
  - Executable is like Runnable, but can throw an exception
  - Use a lambda for this to invoke your method
- Example:

```
assertThrows(
   IllegalArgumentException.class,
   () -> methodToTest()
);
```

Practice

- Throw an exception in the isTriangle(…) method if a negative side length is passed in. Test for this exception.

Unit Testing Other Topics

- There is much more to learn! Some topics include:

- Display name and name generators
- Launching from the console
- Specifying parameters with enums
- Repeated tests
- Nested tests
- Test suites (running multiple test classes together)