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 is faster and more reliable
- Evaluation of the logic of the code
- Preserved test cases to use with future modifications

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

JUnit4 and JUnit5

- JUnit4
  1. Download junit and hamcrest jars:
     - https://github.com/junit-team/junit4/wiki/Download-and-Install
  2. Add **both** jars:
     - Project > Properties > Java Build Path > Libraries > Add External JARs
  - JUnit API: http://junit.sourceforge.net/javadoc/
- JUnit5
  - Released in spring 2018
  - Supports lambdas and streams
  - Might requires an upgrade to eclipse
    - https://www.eclipse.org/community/eclipse_newsletter/2017/october/article5.php
  - Designed to work with software development/build systems such as Maven and Gradle
  - https://junit.org/junit5/

Using JUnit4

- 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 (e.g., empty and singleton lists)
- Run your test
  - Most IDEs have graphical test runners (including eclipse, NetBeans, and IntelliJ)
  - You can also run a *test runner* (e.g,. from the command line)

Example Test Runner

```
import org.junit.runner.*;
import org.junit.runner.notification.*;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(MyJUnitTestClass.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

JUnit Method Annotations

- @Test- the method can be run as a test case
- @Before- method must be executed before **each** test
  - public void setUp
- @BeforeClass- static method must be executed **once** before *all* tests
  - public void setUpBeforeClass
- @After- method must be executed after **each** test
  - public void tearDown
- @AfterClass- static method must be executed **once** after *all* tests
  - public void tearDownAfterClass
- @Ignore- 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) // int, short, long, byte, char, Object
  - uses the .equals method on Objects
- assertTrue(boolean)
- assertFalse(boolean)
- assertNotNull(Object)
- assertNull(Object)
- assertSame(Object, Object)
  - uses == on Objects
- assertNotSame(Object, Object)
- assertArrayEquals(expected, actual) // arrays of type int, long, short, char, byte, Object

- All methods take an optional first parameter: String message

Example Method Structure

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

Practice

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

Suite Tests

- A suite is a collection of classes that can be run together

```
@RunWith(Suite.class)
@Suite.SuiteClasses({MyTestClass1.class, MyTestClass2.class})
public class MyCombinedTestClass {  }
```

Practice

- Review the class that runs both the course and bank account tests.

Parameterized Tests

- Allows you to run the same test case with different inputs
  - Example: run multiple tests of the deposit method with positive, negative, zero
- All test methods will run with all parameter combinations

Creating Parameterized Tests

- Use the @RunWith(Parameterized.class) annotation on the class
- Create instance data variables and a constructor to store test data
  - The variables and parameters to the constructor are one “set” of data
- Create a static method that generates and returns test data tagged with the @Parameters method
  - Return type: Collection<Object[]>
  - Each element in the array is an Object[] that contains the test cases

```
return Arrays.asList(new Object[][] {
        {input1a, input2a, input3a, result1a, result2a} ,
        {input1b, input2b, input3b, result1b, result2b}
});
```

Practice

- Review the parameterized bank account tester
- Write a method to test whether a triangle is isosceles.

Exceptions

- You can test whether expected exceptions are properly thrown.
- Use the @Test(expected = ExceptionClass.class) tag

Practice

- Throw an exception if any triangle side lengths are negative. Test for this exception.
  - Note: because this method is in a parameterized class, it will be run with all of the test cases, even though we only need to run it once.