Santiago Bermudez

Lab: Skills - JUnit and EclEmma in Eclipse

Getting Ready: Before going any further, you should:

1. Setup your development environment.

2. Download the following files:
   GiftCard.java
   to an appropriate directory/folder. (In most browsers/OSs, the easiest way to do this is by right-clicking/control-clicking on each of the links above.)

3. If you don't already have one from earlier in the semester, create a project named `eclipseskills`.

4. Drag the file `GiftCard.java` into the default package (using the "Copy files" option).

5. Open `GiftCard.java`.

Part 1. JUnit Basics: JUnit is an open-source testing framework. It provides a way to write, organize, and run repeatable test. This part of the lab will help you become familiar with JUnit.

1. Create an empty JUnit test named `GiftCardTest` in the default package by clicking on File-New-JUnit Test Case
   Use the most recent version of JUnit; if necessary, add JUnit to the build path when asked.

   Note: Normally, you should put tests in their own package(s). To keep things simple, we will break that rule.

2. Copy the following code into `GiftCardTest`.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.rules.ExpectedException;


public class GiftCardTest
{
    @Test
    public void getIssuingStore()
    {
        double       balance;
        GiftCard     card;
        int          issuingStore;

        issuingStore = 1337;
        balance      = 100.00;
        card = new GiftCard(issuingStore, balance);

        assertEquals("getIssuingStore()",
                     issuingStore, card.getIssuingStore());
    }
}
```

3. A JUnit test suite is a class, much like any other class. Tests are methods that are preceded with the annotation `@Test`. (Note: An annotation provides information about a program but is not part of the program. Annotations have no effect on the operation of the program. Instead, they are used to provide information to tools that might use the program as input.)
   How many tests are in the test suite `GiftCardTest`?

Answer: Just 1, I believe.

4. JUnit has an `Assert` class that has a static `assertEquals()` method with the following signature that is used to compare expected and actual results:

```
public static void assertEquals(String description, int expected, int actual)
```

   where `description` is a human-readable `String` describing the test, `expected` is the expected result, and `actual` is the result of actually running the code being tested.

   How would you call this method and pass it the `String` `"getIssuingStore()"`, the `int` `issuingStore`, and the `int` returned by the `card` object's `getIssuingStore()` method?

Answer: Assert.assertEquals("getIssuingStore()", issuingStore, card.getIssuingStore());

5. How is this method actually being called in `GiftCardTest`?

Answer: assertEquals("getIssuingStore()", issuingStore, card.getIssuingStore());

6. Why isn't the class name needed?

Answer: If you look at the very top of the class where the first line of code is, you will notice a line that says "import static org.junit.Assert.assertEquals;". assertEquals is a static method. Since you can't use static methods without importing them explicitly in a static way, you would have to use a line like the one described above. "import static org.junit.Assert.assertEquals;" is basically an import statement for "org.junit.jupiter.api.Assertions.*;" that has a static modifier. This tells the compiler that the class name can be omitted from calls to static methods in this program. As writing Assert.assertEquals() can be redundant, some people may choose to use a static import instead.

7. Execute `GiftCardTest`. Why is it somewhat surprising that you can execute `GiftCardTest` without a main method?

Answer: Normally, we can only execute classes that have a main method. In theory, no Java program can run without a main method, but with JUnit, the main method is hidden. JUnit has a class with a main method and that class is what is being executed, which then calls the various tests in GiftCardTest.java. So, in a sense, you don't have to write a main method in the JUnit test case class.

8. What output was generated?

Answer: In the console window, no output was generated, but a JUnit window opened up, which showed runs and tests results. In this window, it showed all the runs, errors, and failures, as well as a finishing time. The window shows that a test suite was run and that all tests were passed (*for me, it only did one run).

9. To see what happens when a test fails, modify the `getIssuingStore()` method in the `GiftCard` class so that it returns `issuingStore + 1`, compile `GiftCard.java`, and re-run the test suite.
   Now what happens?

Answer: In the JUnit window, the tests fails. This time, the failure trace subsection of that window gives us results and the count for failures is set to 1. The bar that was green before is now red.

10. In the "Failure Trace", interpret the line:

```
java.lang.AssertionError: getIssuingStore() expected:<1337> but was:<1338>
```

   Note: You may have to scroll the "Failure Trace" window to the right to see the whole message.

Answer: It basically just says that when the test called "getIssuingStore()" was run, it had been expecting for the result to return as 1337, but instead it was 1338.

11. What mechanism is JUnit using to indicate an abnormal return?

Answer: It is just throwing an error, just like other development environments would.

12. Before you forget, correct the fault in the `getIssuingStore()` method.

13. The Assert class in JUnit also has a static `assertEquals`() method with the following signature:
`public static void assertEquals(String description, double expected, double actual, double tolerance)`
where `tolerance` determines how close to double values have to be in order to be considered "approximately equal".

Add a test named `getBalance()` that includes a call to `assertEquals()` that can be used to test the `getBalance()` method in the card class (with a `tolerance` of `0.001`).

14. How many tests are in your test suite now?

Answer: This time instead of 1, there are 2 tests.

15. Suppose you had put both calls to `assertEquals()` in one method (named, say, `getIssuingStore()`). How many tests would be in your test suite?

Answer: 1, if you just had that one method. If you added it in with the other two methods, it would be 3. Although, this just tells us that if we had multiple assertions in a @Test-annotated method, it would only count as one test.

16. Re-compile and re-execute the test suite. How many tests were run?

Answer: 2. Both of them passed.

17. The Assert class in JUnit also has a static `assertEquals()` method with the following signature:
`public static void assertEquals(String description, String expected, String actual)`
Using JUnit terminology, add a test named `deduct()` to your test suite that can be used to test the `deduct()` method in the `GiftCard` class. Note: Be careful, the variables that are declared in the `getIssuingStore()` method are local.

18. Execute the test suite.

Cont'd below.

Part 2. Coverage: This part of the lab will help you understand coverage tools and coverage metrics.

1. Read Eclipse_Eclemma.pdf.
2. Run `GiftCardTest` using EclEmma, click on the "Coverage" tab and expand the directories/packages until you can see `GiftCard.java` and `GiftCardTest.java`.
   Why is the coverage of `GiftCardTest.java` 100% and why is this uninteresting/unimportant?

Answer: This essentially just means that all of the tests were executed. It just tells us that there was no code that was not tested.

3. How many of the tests passed?

Answer: All 3 of them.

4. Does this mean that the `GiftCard` class is correct?

Answer: No.

5. What is the statement coverage for `GiftCard.java`?

Answer: 55.7%

6. Select the tab containing the source code for `GiftCard.java`. What is different about it?

Answer: There are now a bunch of statements highlighted in different colors, like green, red, yellow, and brown.

7. What do you think it means when a statement is highlighted in red?

Answer: The statement may not have been executed by the tests. It might never have been covered.

8. Hover your mouse over the icon to the left of the first if statement in the constructor. What information appears?

Answer: "2 of 4 branches missed."

9. Add tests to your test suite so that it covers all of the statements and branches in the `deduct()` method in the `GiftCard` class.

10. Your test suite still does not cover every statement in the `GiftCard` class. What is different about the statements that remain untested?

Answer: They involve throwing exceptions.

Cont'd

Part 3. Testing Methods that Throw Exceptions: This part of the lab will help you learn how to test methods that throw exceptions.

1. The easiest (though not the most flexible) way to test for exceptions is to use the optional `expected` parameter of the `@Test` annotation. For example, add the following test to your test suite.

```
@Test(expected = IllegalArgumentException.class)
public void constructor_IncorrectBalance() throws IllegalArgumentException
{
    new GiftCard(1, -100.00);
}
```

   Note: `IllegalArgumentException` is an unchecked exception. Hence, the code will compile even if it isn't re-thrown. If you are testing for a checked exception then the method must specify the exception.

2. Add a test to your test suite named `constructor_IncorrectID_Low()` that covers the case when the `storeID` is less than 0.

Part 4. Coverage and Completeness: This part of the lab will help you better understand code coverage and the completeness of test suites.

1. Run EclEmman on your current test suite. What is the statement coverage for `GiftCard.java` now?

Answer: 100%

2. What branch does the test suite fail to test?

Answer: It fails to test the storeID > MAX_ID branch.

3. Add a test to your test suite named `constructor_`IncorrectID`_High()` that covers the other branch.

4. Run EclEmma. What is the branch coverage now?

Answer: All of the branches are now covered.

5. From a "white box" testing perspective, is your test suite complete? Conversely, can you think of tests that should be added?

Answer: Not quite. Perhaps we should add a test where both an illegal storeID and illegal openingBalance were passed.

6. From a "black box" testing perspective, is your test suite complete? Conversely, can you think of tests that could be added?

Answer: No. Perhaps we should test multiple combinations of legal and illegal parameters, as opposed to just one on its own.