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 GiftCardTestin 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?

   One, the getIssuingStore() method.

4. JUnit has an Assertclass 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 descriptionis a human-readable Stringdescribing the test, expectedis the expected result, and actualis the result of actually running the code being tested.

How would you call this method and pass it the String"getIssuingStore()", the intissuingStore, and the intreturned by the cardobject's getIssuingStore()method?

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

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

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

6. Why isn't the class name needed?

If you look at the top of the class you will see that the import statement for org.junit.Assert.assertEquals has a static modifier. This tells the compiler that the class name can be omitted from calls to static methods.

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

Because GiftCardTest doesn't have a main() method, and, in the past, we've only been able to execute classes that have a main() method.

8. What output was generated?

None, but test results appeared in a new window. That window indicates that a test suite was run and that all of the tests passed.

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?

The test fails.

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.

It says that when the test described as "getIssuingStore()" was run, the result returned should have been 1337 but was 1338.

11. What mechanism is JUnit using to indicate an abnormal return?
    It's throwing an exception.

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).

```
@Test
public void getBalance()
{
   double       balance;
   GiftCard     card;
   int          issuingStore;

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

   assertEquals("getBalance()",
             balance, card.getBalance(), 0.001);
}
```

14. How many tests are in your test suite now?
    2.

15. Suppose you had put both calls to assertEquals() in one method (named, say, getIssuingStore()). How many tests would be in your test suite?
    1, though it would be capable of testing the range of both previous tests, which would make results complicated to analyzed

16. Re-compile and re-execute the test suite. How many tests were run?
    2, both of which 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.

```
@Test
public void deduct_RemainingBalance()
{
   double      balance;
   GiftCard    card;
   int         issuingStore;
   String      s;


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

   s = "Remaining Balance: " + String.format("%6.2f", 90.00);
   assertEquals("deduct(10.00)",
             s, card.deduct(10.0));
}
```

18. Execute the test suite.

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?
   This just means that all of the tests were executed. That isn't at all surprising.

3. How many of the tests passed?
   All three of them.

4. Does this mean that the GiftCard class is correct?
   It means that every functionality which you care to test is correct.

5. What is the statement coverage for GiftCard.java?
   55.7%, which means that only 55.7% of the code present was actually tested.

6. Select the tab containing the source code for GiftCard.java. What is different about it?
   Statements are now highlighted in different colors.

7. What do you think it means when a statement is highlighted in red?
   The statement was never executed by the tests. In other words, the statement was not covered.

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

```
      @Test
public void deduct_AmountDue()
{
   double      balance;
   GiftCard    card;
   int         issuingStore;
   String      s;


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

   s = "Amount Due: " + String.format("%6.2f", 10.00);
   assertEquals("deduct 110.00 from 100.00",
             s, card.deduct(110.0));
}

@Test
```

```
public void deduct_InvalidTransaction()
{
   double       balance;
   GiftCard     card;
   int          issuingStore;
   String       s;


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

   s = "Invalid Transaction";
   assertEquals("deduct -10.00 from 100.00",
             s, card.deduct(-10.0));
}
```

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

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.

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

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?
   100%

2. What branch does the test suite fail to test?
   So far, the test suite is only covering the storeID < 0 branch and not the storeID > MAX_ID branch.

3. Add a test to your test suite named constructor_IncorrectID_High() that covers the other branch.

```
@Test(expected = IllegalArgumentException.class)
   public void constructor_IncorrectID_High()
   {
      new GiftCard(100000, 100.00);
   }
```

4. Run EclEmma. What is the branch coverage now?
   All branches are covered.

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

Technically the test suite is complete, but I might add some fuzz testing for bad input to see if I can get it to break something (including inputs that should be considered illegal)

6. From a "black box" testing perspective, is your test suite complete? Conversely, can you think of tests that could be added?
No, because we ought to add fuzz testing for bad input to see if we can get it to break something (especially by using inputs that should be considered illegal). Similarly, if this function is run in parallel to some other function, I might design a test to see if we can hit and properly handle race conditions or something of the like.