1 Objectives

- Learn to write unit tests with JUnit .
- Explore enumerated type in Java.

2 Preliminaries

In Eclipse, create a new Java Project and import the two files you will need from

`~cs479/public_html/2009-fall/testingEx`

2.1 Cash Register

`CashRegister` is a very simple representation of a cash register. You use its methods as follows.

1. Use `recordPurchase()` to record the amount of a purchase.
2. Use `enterPayment()` to enter the amount of the payment.
3. Use `giveChange()` to determine how much change is due. This method resets the instance fields to zero.

The `main()` method contains an example. Run the program to see how it works. Note that the expected result and the actual result are very close, but not identical.

2.2 Money Enumerated Type

`Money` is an *enumerated type* that is used to represent money. An enumerated type (`enum`) is used to represent a type that has just a few instances. Take a look at this `enum` and you will see that it looks very similar to a class definition. The only differences are that it uses the word `enum` instead of `class`, and it declares all of its instances right inside its definition. Open the `Money` enumerated type now and take note of these features.

At the beginning of the `enum` definition you will see the declaration of its instances. Each declaration (`PENNY`, `NICKEL`, ...) invokes the constructor for the enumerated type and creates one instance. To reference the `PENNY` instance use `Money.PENNY`. See the `main()` method in `CashRegister` for more examples.

3 JUnit Testing

In this exercise, you will write JUnit tests for the `CashRegister` class.

To test the methods in a class, you will produce another class that contains all of the tests.

3.1 Create a Test Class

Eclipse will do most of the work of creating a test class for you. Here are the steps you need to follow.

- Begin by highlighting `CashRegister.java` in the package explorer. Right click and select **New** → **JUnit Test Case**.
- In the dialog that appears, click the radio button that says **New JUnit 4 test**. Eclipse will have selected `CashRegisterTest` as the name of the class. There's no reason to change this.

  Click on the check box that tells Eclipse to generate comments.

  Click **Finish**.

  A dialog box will appear with a warning that JUnit 4 is not on your build path. Click on the **OK** button to add it.

3.2 Write a Test

Each test that you write *must* be preceded by an *annotation* that identifies it as a test. Enter the following test stub in the new class. `@Test` is an annotation that identifies this new method as a test.

```
@Test
public void testSimpleCase() {
}
```

Eclipse will indicate that it doesn't recognize the `@Test` annotation and offer to import `org.junit.Test`. Ask it to do that. Now enter the rest of the test.

```
private static final double EPSILON = 1.0e-12;

@Test
public void testSimpleCase() {
    CashRegister register = new CashRegister();
    register.recordPurchase(1.82);
    register.enterPayment(1, Money.DOLLAR);
    register.enterPayment(3, Money.QUARTER);
    register.enterPayment(2, Money.NICKEL);
    double expected = 0.03;
```

```
    double actual = register.giveChange();
    assertEquals(expected, actual, EPSILON);
}
```

Eclipse should indicate an error when it sees `assertEquals()`, but it will suggest the import statement you need to fix it. Import the suggested file.

Note: *All* test methods should be public, have a void return type, and have no parameters. You should chose the name of each test method carefully since the name is what is reported by JUnit when an error is discovered.

3.2.1 Things to Watch For

Here are some things that you should pay attention to when comparing doubles.

1. It is almost always an error to check if two doubles are equal. Remember the output we got when we ran the main program? It told us that the actual result was

   ```
   0.030000000000000027
   ```

   when we expected 0.03. The problem is that there is almost always some inaccuracy when computing with doubles with fractional parts. Instead of checking that two doubles are equal, it is *always* better to see if they are very *close*. That's why `assertEquals()` has an additional parameter when comparing doubles. You need to tell it how close the doubles should be before they are considered equal. In our test we are telling JUnit that two doubles must be within 10^{-12} of each other to be considered equal. We defined this tolerance in the constant `EPSILON` which you will use for other tests.

   This third parameter for `assertEquals()` is not used when comparing most other types of objects.

2. If you omit the third parameter to `assertEquals()` when comparing doubles, it is not a syntax error, but you will get unexpected results. For example, if you eliminate `EPSILON` and change the expected result to 0.02 (which is wrong), the test will pass! It's not clear why this happens, but be aware that the problem exists.

3.3 Run the Test

Remove the `main()` method in `CashRegister`. It's no longer necessary since you have written JUnit tests to do the same thing.

Select **Run** → **Run As** → **JUnit Test**. You should see a green bar indicating that the test succeeded.

Click on the **Package Explorer** tab, and if necessary open `CashRegisterTest`. Change the "expected" variable to 0.04 and run under JUnit again. You should see a red bar and a list of tests that failed each marked with a blue x. In this case, only one test failed. Double click on the blue x next to testRecordExpense and Eclipse will highlight the assert statement that failed.

When you write tests, you should try to test as much of your code as possible. Write tests for all of your methods except for setters and getters. Try to have tests that tries both sides of every if statement.

3.4 Testing an Exception

Click on the **Package Explorer** tab, and if necessary open `CashRegister`. You should see that `recordPurchase()` needs a test to see if it throws an exception when its parameter is negative. Enter the following test for the exception.

```
@Test(expected = IllegalArgumentException.class)
public void testRecordExpense() {
    CashRegister register = new CashRegister();
    register.recordPurchase(-3.12);
}
```

The `@Test` annotation tells JUnit that we expect this method to cause an illegal argument exception. If it does not happen, it's an error. Click the **JUnit** tab in the upper left of the Eclipse window and press the run button. It's a green circle with a white arrow inside. You should see a green bar.

Change the the -3.12 to 3.12 in the above test and run again to see the test fail.

4 Fixtures

Code that is repeated before each test is called a *fixture*. So far, you have written two tests. Each test needed to create a `CashRegister` object before performing the remainder of the test. Instead of repeating the code, you will create a fixture. Code that should be performed before each test is preceded with a `@Before` annotation. Code that is performed afterwards is preceded with `@After`. You will need to have Eclipse supply the proper import statement.

Begin by creating a `CashRegister` instance field.

```
private CashRegister register;
```

Then create a method which will be executed before each test.

```
@Before
public void setUp() {
    register = new CashRegister();
}
```

Using the name `setUp` is traditional, but not required in JUnit 4.

Now you can eliminate the creation of the `CashRegister` object in each of your test methods. Do that now and rerun your tests. Everything should still be working.

The method that is run after each test is traditionally called `tearDown`. Again, the name is not required, but it is traditional. We don't really need a `tearDown()` method, but let's write one anyway to get the practice. Set the `register` instance variable to `null` after each test. The `setUp()` method will create a new `CashRegister` object for us.

```
@After
public void tearDown() {
    register = null;
}
```

Run your tests again to make sure everything is working.

JUnit has test methods called `assertTrue` and `assertFalse` that you may find useful when writing your tests. Each accepts a boolean condition that it expects to be true or false. For example, if you were writing a test in which you expected the result to be less than 4, you could write

```
assertTrue(result < 4);
```

5 References

1. The complete JUnit 4 API is available at `http://junit.sourceforge.net/javadoc_40`.

2. The JUnit home page on SourceForge is here `http://junit.sourceforge.net/`. You will find documentation and additional links.