Bucknell UNIVERSITY

CSCI 204 – Introduction to Computer Science II

Lab 5 – Junit, UML, and Enums

1. Objectives

In this lab, you will learn the following:

- Learn to write unit tests with JUnit
- Learn how to use an enumerated type (enum)
- Practice implementing `compareTo()` and `equals()`
- Review some basic UML and class design material

2. Introduction

Tools are critical in doing any successful work. Software design and development are no exception. Without good tools the difficulty level of programming would be increased dramatically. Imagine how difficult it would be if you were still using teletype terminals, or punched cards to enter programs into a computer!

In today's lab, you will learn to use **JUnit**, a unit testing framework for Java programs, and review Violet, a tool for creating UML diagrams that represent Java classes.

3. Getting Started

As usual, you should set up your Eclipse workspace for today's lab. Import your files you need from the file system at

`~csci204/2011-spring/student/labs/lab05`

`You should see the following files:`

- `BankAccount.class.violet`
- `BankAccount.java`
- `CashRegister.java`
- `Money.java`
- `Person.java`

You will notice that one of the files is a Violet UML file. Drag this file to the main project folder. Keep Java source code in the usual `src` folder. Commit your project to your subversion repository. Be sure to specify the correct repository path for your labs.

4. Reviewing an Existing Class Design

For the bank account exercise, there are multiple files. These files are:

- a fully implemented `BankAccount` class
- `BankAccount.class.violet` which is a UML diagram for the `BankAccount` class
- a `Person` class (which is missing something important)
- a `Money` enumeration class

Each of the classes contains a `main()` method. This is a common practice when you want to write code for a specific class that is designed to test only that class.

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. Before you continue, run the program to see how it works. Note that the expected result and the actual result are very close, but not identical.

2. `Money` Enumerated Type

`Money` is an *enumerated* type that is used to represent money. Read this section carefully, since you will be responsible for it at a future date!

An enumerated type (enum) is a special type in Java used to represent a fixed set of constants in your program. Take a look at this enum and you will see that it looks very similar to a class definition. Like classes, enums in Java are a type. The primary differences are:

- An enumerated type uses the word enum instead of class
- *Enum constants* are not your typical numeric constants you are used to. They are actual *instances* of the enum type. The instances are defined right inside the enum definition. These instances represent the constants themselves, and thus should follow the naming conventions for constants
- Enum types can define an *optional constructor*, and likewise, the constants themselves can have an optional list of arguments passed to the constructor. (This feature is utilized in the `Money` enum)
- Unlike class types, you cannot instantiate an enum type outside of what is defined in the enumeration itself. This results in a compile-time error. The Java implementation of enumerations ensures that those instances defined inside of the enum definition are the *only* instances that exist in your program, and that each one is unique.
- You can not extend an enum type (i.e. the final in the enum type is implicit)
- As a class type, enum definitions can define *data* contained in each instance

- As a class type, enum definitions can define *methods* that act on the data in each instance

These differences above are important! They clearly establish some distinguishing criteria that make Java enums *much* more powerful than simply declaring your own public final int constants.

2.1. Explore `Money`

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

For those that wish to further their knowledge and gain some appreciation of what you can accomplish with enumerated types, consult http://download.oracle.com/javase/1.5.0/docs/guide/language/enums.html

5. Classes and UML – a opportunity for review

You will be manually testing the `BankAccount` class and creating an associated UML diagram. You will also be creating and testing a new class called `Address`. At each step you are asked to test the classes. You should use the `main()` method in each class as your test method. You are also asked to create a UML diagram for all the classes.

1. The `BankAccount` Class

Run the `main()` method in the `BankAccount` class. Observe the results. Examine the implementation for the `BankAccount` class. Pay special attention to the testing components in the `main()` method.

Examine the UML diagram for the `BankAccount` class by double clicking on the `BankAccount.class.violet` file in your project. Examine the Java code and find the corresponding parts in the UML diagram. Close the UML diagram when you are finished.

2. The `Person` Class

Next, examine the class `Person`. This class doesn't compile because the `Address` class doesn't exist yet. This is handled in the next section. When you have finished writing the `Address` class, you will test it by running its `main()` method.

3. The `Address` Class

Create a new class called Address. Here are some guidelines that will let you experience some more of the efficiency that can be had by using an IDE such as Eclipse. Some of this may include some steps that you haven't used before, so do not skip!

4. When you ask Eclipse to create the class for you, check the box that says **public static void main(String[] args)** so that it will generate a `main()` method for you. You will use this method for testing. You should also check the **Generate comments** box.

5. Add instance variables for street, city, state, and the zip code. All should be of type `String`. As always, make the instance variables private.

6. Use **Source → Generate Constructor using Fields. . .** to produce a constructor that will accept initial values for each of the instance variables. Check the box next to each instance variable in dialog that appears, if they aren't checked already. Check the **Generate constructor comments** and **Omit call to default constructor super()** boxes. Fill in the comments.

7. Use **Source → Generate Setters and Getters. . .** to generate mutator (setters) and accessor (getters) methods for your class. Check all of the instance variables and check the **Generate method comments** box. Complete the comments for each method.

8. Write a `toString()` method for the class. It should produce a string containing a nicely formatted address.

9. Add a `requestStatement(Address addr)` to the `BankAccount` class that prints a confirmation message to the user which includes the provided `Address` argument.

10. Double click on `BankAccount.class.violet` to open the UML diagram for the `BankAccount` class. In the project window, drag `Address.java` into the UML diagram. A diagram for `Address` will appear. Save the file and close it.

Be sure you include a `main()` method in the `Address` class to test it out. Add code to test each of the methods in the `Address` class. Run your tests. If you are unsure how to do this, look at the `main()` method in `Person`.

4. Remove Errors

Once you have completed the `Address` class, everything should compile. If there are any remaining warnings or errors, correct them. Run the `main()` method in `Person` and make sure it is working properly. (Though we are not asking for a copy of the output, this will be tested as part of your grade.)

5. Complete the UML Diagram

Double click on the UML diagram again and drag `Person.java` into the window. A diagram for `Person` should appear. You should now have three classes in your UML diagram.

Complete the UML diagram by

- adding the missing information to the fields and methods in the UML format shown in class.
- indicating the relationships between the classes.

You may need to move the boxes around to get the diagram to look nice. Explore some of the options available in the relationships lines to see how you can adjust the way the lines automatically route in your diagram.

You can find more information about Violet at http://sourceforge.net/projects/violet.

6. JUnit Testing

In the previous exercise, you had to manually test your classes by printing to the console in the `main()` method. However, there is a more modular and automated way to test Java code and it even has special support in Eclipse. In this section of the lab 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.

1. Create a Test Class

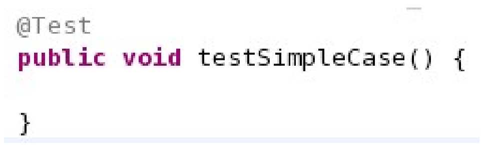
Eclipse will do most of the work of creating a test class for you. Here are the steps you need to follow.

- 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. It is a good idea to name all of your test classes with the word `Test` at the end of the name of the class being tested, so there is no need to change this.
- Click on the check box that tells Eclipse to generate comments.
- Confirm that the **Class under test** field is set to `CashRegister`
- Click Finish
- Unless you've already included the JUnit library into the build path for this project, you will see a warning appear telling you that JUnit 4 is not on your build path. Click on the radio button that tells Eclipse to perform the action of including the library (and confirm that the following action is to add the JUnit 4 library to the build path.) Click on the **OK** button.

You now have a new empty test class.

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 method as a test.



```
@Test
public void testSimpleCase() {

}
```

Eclipse should indicate that it does not recognize the `@Test` annotation. When you hover the cursor over the error, it should offer to `import org.junit.Test`. Ask it to do that, and the error should disappear. Now, enter the rest of the code in the method:

```
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.

2.1. Things to Watch For

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

11. 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.

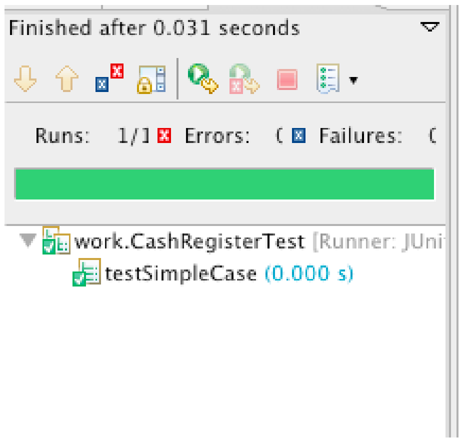
    This third parameter for `assertEquals()` is not used when comparing most other types of objects.

12. 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. 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 `CashRegisterTest.java` in the panel on the left, right-click, and select **Run As…** , then select **JUnit Test**. It may take a little time for the test to run the first time. After it does, you will see a green bar indicating that the test succeeded. If you had multiple tests, you would see a green checkmark next to each test that succeeded, or a blue or red X next to those that did not. If you have trouble getting the test to run, get help before continuing.


Finished after 0.031 seconds

Runs: 1/1 Errors: (Failures: (

work.CashRegisterTest [Runner: JUni

testSimpleCase (0.000 s)

7. Test Coverage

When you write tests, you should try to test as much of your code as possible. *Test coverage* refers to the amount of your code that is covered by tests. The test we implemented above was a very general test. It tested out all of the primary methods of the class, but didn't do any exhaustive testing on individual classes. If the test failed, it would be very difficult to determine exactly which method failed. Therefore, you should plan on writing *individual* tests for all of your methods, commonly known as *unit tests*. Unit tests are, by definition, the smallest testable part of an application. In object-oriented designs, the smallest testable parts are usually individual methods in each class. This is yet another reason why a good design has small methods with very well defined preconditions and postconditions. Your unit tests will be designed around those conditions. (NOTE – it is common to ignore individual testing of get and set methods, since those will be inherently tested in your other methods.)

1. Testing an individual method

Click on the **Package explorer** tab, and if necessary open `CashRegister`. Examine the `recordPurchase` method. Let's add another test to verify that our method for recording a purchase works properly. Examine the method, and you'll notice that it doesn't do much other than add the amount to the purchase, or throw an `IllegalArgumentException` if a bad amount was passed to the method. We can test to verify that the exception is thrown correctly. How can we use the JUnit framework to do this?

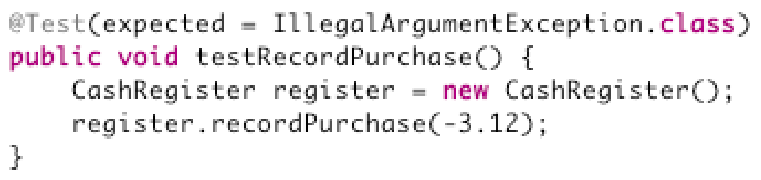
1.1. Exploring Javadoc for JUnit

First, explore a bit of the javadoc for the JUnit framework, available at http://kentbeck.github.com/junit/javadoc/latest/ . Click on `org.junit` in the package window in the top left. You will see that there are not too many classes or annotation types listed here. And fortunately, this represents the majority of what you will ever use with JUnit, at least at this stage.

Click on **Test** in the **Annotation Types** section. There are two optional arguments that you can pass to `@Test`. One argument is `expected`. Read the example code at the top of this javadoc page to get an understanding of how simple this is to use. Let's implement it for ourselves.

1.2. Testing Exceptions Thrown in a Method

Enter the following test in `CashRegisterTest` class:

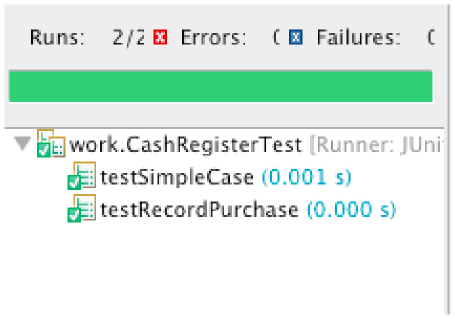


```
@Test(expected = IllegalArgumentException.class)
public void testRecordPurchase() {
    CashRegister register = new CashRegister();
    register.recordPurchase(-3.12);
}
```

The `@Test` annotation tells JUnit that we expect this method to cause an `IllegalArgumentExeception` to be thrown if there is a bad parameter passed to the method. Our test is designed to verify that this happens. If it does not happen, it's an error.

Once you are finished entering the code, 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.

(NOTE – if the tab is not available, just run the test class as a JUnit test as above. The JUnit tab should appear so you can follow the shortcut next time.) You should now notice that both tests have passed, verified by the green bar above the test appearing.



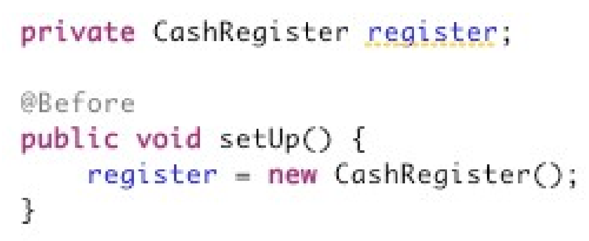
```
Runs:   2/2   Errors:   (   Failures:   (

work.CashRegisterTest [Runner: JUni
    testSimpleCase (0.001 s)
    testRecordPurchase (0.000 s)
```

8. Fixtures

Tests often need to run against a background of a known set of objects. This set of objects is called a test *fixture*. Sometimes, the same fixture ends up being recoded over and over for each test. Consider our own test setup so far. So far, you have written two tests. Notice something similar about each test? A new `CashRegister` object needed to be instantiated 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`.

Begin by creating a `CashRegister` instance field in the test class. Then, create a method which will be executed before each test using the @Before annotation:



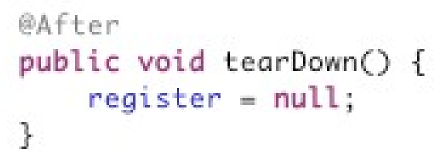
```
private CashRegister register;

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

It is traditional to name the method `setUp()`, but not required.

Now, 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 fine.

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. Be sure to let Eclipse import the correct files as prompted.



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

Rerun your tests one more time and verify that everything is OK.

1. Complete Remaining Tests

It is difficult to know how complex you should make a test. We strive for 100% test coverage, meaning, every possible element, structure, and piece of code has been exhaustively verified to work flawlessly under all possible conditions. In reality, that is very difficult to attain! Theoretically, there are an unlimited number of possible inputs that might be fed into some programs, unlimited number of external events and other tasks that may be running in the operating system, and so on. You can not possibly test every possible case! However, it is quite reasonable to make assumptions about your domain you are working in, about your expected input, make assumptions about external events, and work for test coverage that will test the most likely scenarios your program will encounter that can also directly affect the correctness of your program.

Your tests should be reasonably complex. Again, this is difficult to quantify, but one easy method is to count the number of branching constructs contained in your test (i.e. how many if statements, loops, etc.). These usually correspond to different paths that can be traversed through the method being tested. Remember, your goal is to test *all* possible paths your method may ever encounter! For starting off, try to work to develop test methods that test at least *two* different paths.

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

assertTrue(result < 4);

Be sure that you experiment by changing some of the values in your test cases above and watch what happens in the JUnit explorer panel when a test fails.

2. Compare Cash Registers

Suppose you would like to compare cash registers to one another. Introduce a new `String` instance field called `name` that you will use when comparing cash registers. You will compare the lexicographic (alphabetical) ordering of the names. You will have to modify the constructor so that each cash register gets a name when you create it.

Begin by specifying that the `CashRegister` class implements

```
Comparable<CashRegister>
```

Then write your `compareTo()` and `equals()` methods for the `CashRegister` class. Fortunately, you can do this quite easily by using the already existing corresponding methods in the `String` class. Write tests to check your two new methods. (Consult the Java API and/or the class material on the Comparable interface if you can't recall how to implement this.) You are only implementing these two methods. You do not need to implement any sorting or searching to test it out, unless you want to.

9. Completing your work

Write Javadoc comments for all of the non-JUnit test methods. Also, be sure complete the UML diagram to reflect the existing classes and their relationships correctly. Submit everything to your subversion repository.