Lab walk-through #3	
Topic	**Automated Software Testing (JUnit)**
Week Date/Time Location	3 and 4 – Session 1 and 2

1. Introduction

Software Testing is meant to avoid software failure.

A failure is caused by a fault in the code base. Fault identification and fault correction is known as debugging. Software testing is about identifying possible system failures and designing a test case that proves that this particular failure is not experienced by the software.

There are many kinds of software testing: unit testing, integration testing, function testing, acceptance testing and installation testing. At the day-to-day programming level, unit testing can be easily integrated in the programming effort by using a Unit Testing Framework.

'Testing can reveal only the presence of faults, never their absence.' – Dijkstra

2. Lab Objectives

The objective of this lab exercise is to introduce the concept of unit testing within the Java framework of JUnit. By the end of the lab the student must be able to design and set up JUnit tests for a program they write or are given.

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer
2. Completed the ADT lab from last week.

4. Lab Exercise

A unit test is a piece of code that exercises a very small and specific area of functionality. Usually a unit test checks a particular method in a particular context. The goal of unit testing is to isolate specific parts of the program and show that the individual parts are free of certain faults. For example, add a large value to a sorted list, and then confirm that this value appears at the end of the list.

Benefits of automated unit testing:

1. Facilitates change - regression testing can be easily applied by automatically re-running tests after changes are made to make sure that the implementation still works as intended.
2. Identifies defects early in the development cycle. Follow a technique of code-a-little, test-a-little.
3. Creates self-documenting code by adding relevant test cases that describe what the implementation was intended for.
4. Confidence increases with successful and meaningful tests.

For a large system the number of unit tests can go out of control, automated tests are useful in managing this. The Framework provided by Java allows the association between Classes and Methods to corresponding Test Classes and Methods. Automation is achieved by automatically setting up a testing

context, calling each test case, verifying their corresponding expected result, and reporting the status of all tests.

Step 1: Create and implement the class you want to test
You can use the class you created last week. The following class will be used in this exercise:

```
public class Money {
	private int fAmount;
	private String fCurrency;

	public Money(int amount, String currency) {
		fAmount= amount;
		fCurrency= currency;
	}

	public int amount() {
			return fAmount;
	}

	public String currency() {
			return fCurrency;
	}

	public Money add(Money m) {
		return new Money(amount()+m.amount(), currency());
	}


	public boolean equals(Object anObject) {
		if (anObject instanceof Money) {
			Money aMoney= (Money)anObject;
			return aMoney.currency().equals(currency())
					&& amount() == aMoney.amount();
		}
	return false;
	}
}
```

We have defined a class `Money` to represent a value in a single currency. We represent the amount by a simple `int`. When you add two `Money`s of the same currency, the resulting `Money` has as its amount the sum of the other two amounts. For the `equals` method, since `equals` can receive any kind of object as its argument we first have to check its type before we cast it as a `Money`.

Step 2: Create a Test Case class

This step can be done in two ways: manually or through Eclipse. Both approaches will be discussed here.

Eclipse has a plugin for JUnit. It allows you to quickly create **test case** and **test suite** classes to write your test code in. Eclipse allows testing by generating stubs automatically for testing class methods. Manually, you would have to create these stubs by yourself, but it is an unnecessary complication that does not speed up the automation in any way.

Eclipse also allows you to choose between JUnit 3 and JUnit 4. JUnit 3 requires all test case classes to inherit from a Java class called `TestCase` which has some pre-defined methods that are abstract and need to be implemented. JUnit 4 does not require this inheritance but it does need special annotations to become a test case class. In either case, when a test case class is running, the methods are called in the right order and tests are carried out. There are some important methods that you need to know about when using JUnit:

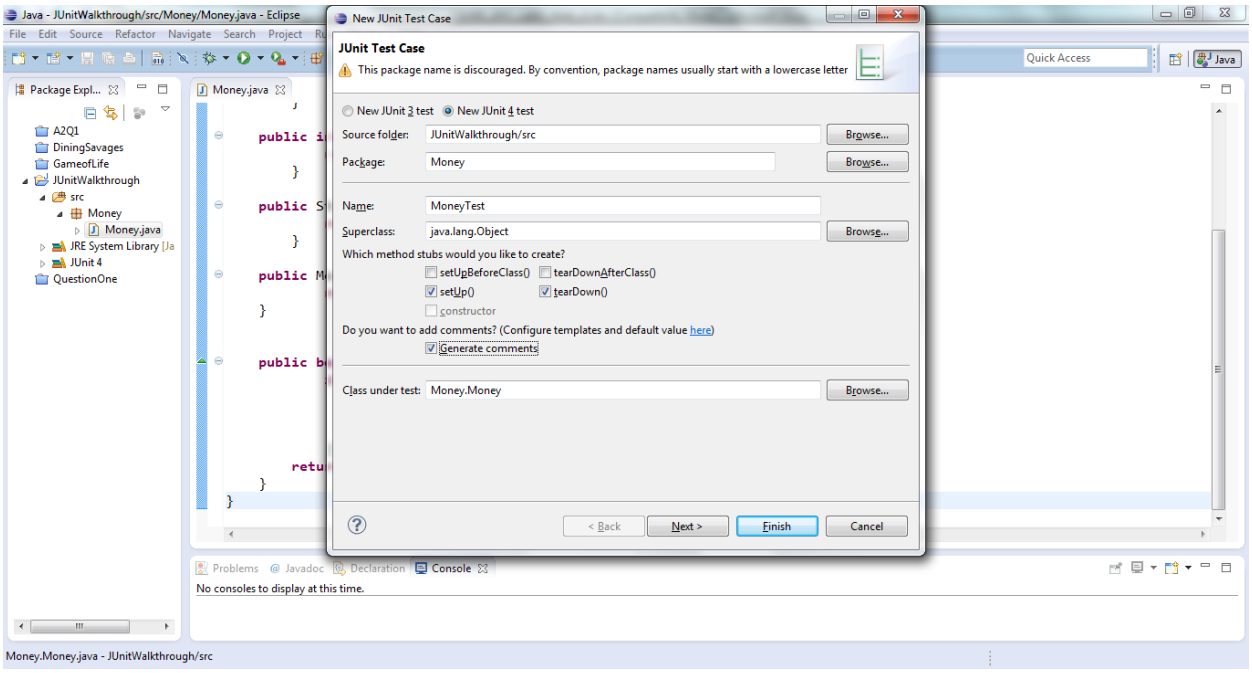
1. setUp() – this is automatically invoked by a class before any tests take place. It is responsible for setting up the context in which the testing will take place. This involves initialising fields, turning on logging, resetting environment variables etc. You have to implement this function. In JUnit 4, the initialisation method does not needs to be called setUp(). It just needs an @Before annotation. You can call the function whatever you like. For example,

```
@Before protected void initialise()
{
      System.out.println("Before testing");
}
```

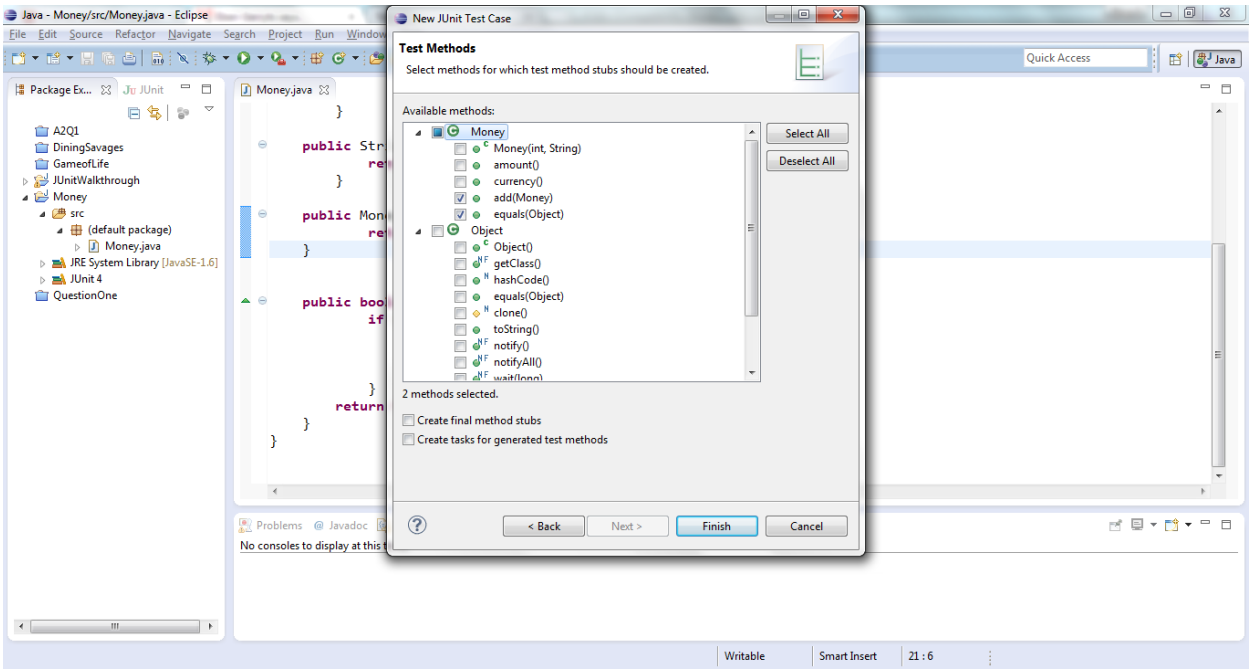
   In this walkthrough, we'll stick to the name `setUp`, but with the appropriate annotation for JUnit4.
2. tearDown() – This method can be used for any clean-up operations necessary after the testing is done, for example if we would need to manually call the garbage collector. Again, in JUnit 4 we only need to add an `@After` annotation to any method you prefer, like in the case above.
3. runTest() – You can override this method to run a particular test and assert its state. Here the word 'assert' is being used as a keyword in Java. This will be explained in Step 3.

Note that if you are using JUnit 3 you should be aware of the run() method too. The run() method applies the Template Method Pattern and creates a skeleton of an algorithm for testing, deferring some steps to subclasses. These subclasses are where you add in your testing code. You do not have to implement this method in this assignment as you are using JUnit 4.

To create a new Test Case class in Eclipse, while your cursor is in the editor of your `Money` class, select File → New → JUnit Test Case. Make sure your Test Case class is in the same package as the class to be tested. There is a default name (`MoneyTest`) for your new class, but there is nothing special about it, you can change it if you wish.



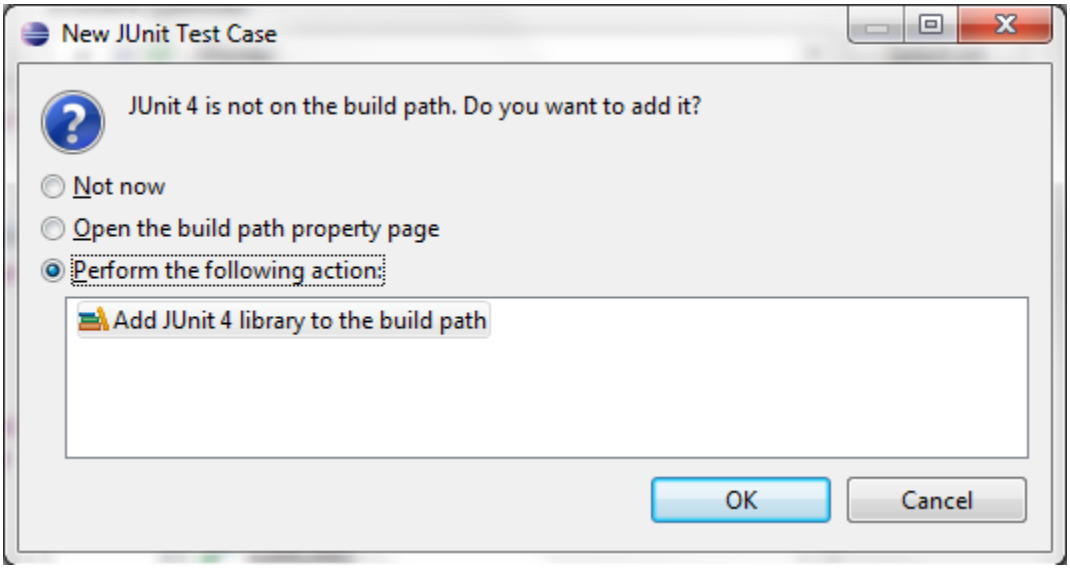
The radio options at the top of the new window ask you to pick between JUnit 3 and JUnit 4. Select JUnit 4. Under the options of "What method stubs would you like to create?" select the "setup()" and "tearDown()" methods. Also select "Generate Comments". Click "Next".



The next screen requires you to select the methods you are going to test in your class. Select the check box next to the Money class. This setting creates a test method that corresponds to each method that you have implemented in your Money class. Deselect the constructor and the accessor methods, since we want to concentrate on checking the methods that modify the object first. Click Finish to see the new class that is generated.

Why would you want to test the constructor and accessor methods? So that you can make sure they are working as intended. For example, you could check to make sure that the constructor actually creates an object with the arguments given to it, and that the accessor methods do not inadvertently change the object.

If you see the following message, click OK to continue to your class.



Note that the class comes with the correct import of packages and, if necessary the right inheritance (only for JUnit 3). This saves you time and allows for more automation, as you will see soon.

Step 3: Code the actual test cases

First we have to develop the context(environment) within which the test cases can be created. Here we start with two dummy Money objects. We create the objects in the setUp function where we can control the values of each object and test our functions on them. Add the following code to your Test Case class.

```
public class MoneyTest {
    …
        private Money f12CAD;
        private Money f14CAD;

        @Before
        public void setup() throws Exception {
                f12CAD = new Money(12, "CAD");
                f14CAD = new Money(14, "CAD");

        }

    …
}
```

Next we look at two methods that are the real unit tests in action. An **assertion** is a statement in the Java programming language that enables you to test your assumptions about your program.

Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the test fails. By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behaviour of your program, increasing your confidence that the program has not found an error yet. Experience has shown that writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs.

Add the following code to the `testEqualsObject` method stub.

***Note**: Remember to remove the call to the fail method that has been automatically generated, otherwise your test will fail immediately.*

What is this method checking?

```
@Test
public void testEqualsObject() {
      assertTrue(!f12CAD.equals(null));
      assertEquals(f12CAD, f12CAD);
      assertEquals(f12CAD, new Money(12, "CAD"));
      assertTrue(!f12CAD.equals(f14CAD));
  }
```

The first Assert statement checks to see if the object f12CAD is not null. We know it is not null since in the setup of this test, we have given it a concrete value of 12. Therefore, we assert that it cannot be null.

The second assert statement checks that f12CAD is equal to itself. This might seem a bit tedious, but the test needs to be water tight and we need to assert whether our `equals` method works in all inputs and conditions. This is the sort of thinking you must adopt to make your unit tests complete.

The third assert statement checks to see if the object f12CAD is the same as a new temporary object we create with the same attributes.

Lastly, the fourth statement confirms that the equals method fails if we compare two objects that we know are not the same.

Can you think of any circumstance we have missed?

The `testSimpleAdd` function works in exactly the same way. We manually work out what we expect the result of the method `add` to be and we check to see if the add function produces the same result. Add this code to the `testAdd` method stub provided.

```
@Test
public void testAdd() {
      Money expected= new Money(26, "CAD");
      Money result= f12CAD.add(f14CAD);
```

```
            assertTrue(expected.equals(result));

        }
```

As you start to design test cases for your methods, you can start to see some holes in our design. What if someone gives you a negative value of currency? We haven't stopped to check for that anywhere in our code. Where would you put that test? Testing allows you to take a step back from your implementation and deliberately look for possible bugs in your system.

Whichever class you are implementing, it is worthwhile to take some time to design test cases for your system. JUnit is also a much more elegant way to design tests, rather than a main method that might clutter up your implementation.

Step 4: Run the tests

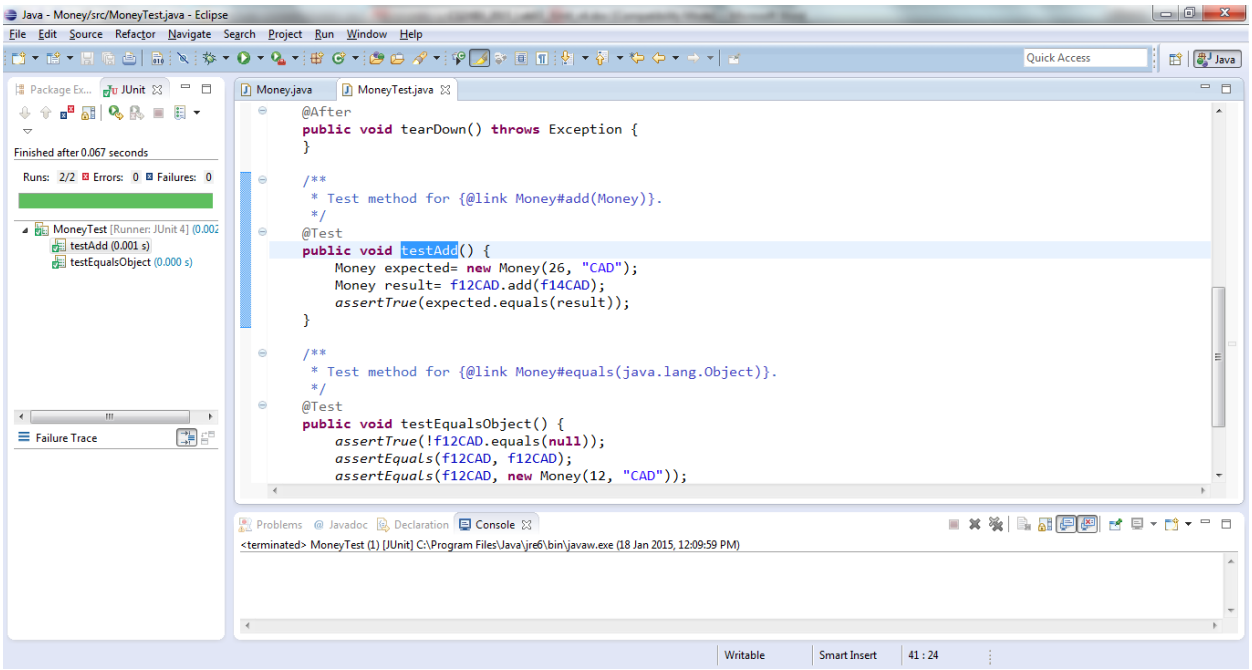
In Eclipse, right click on your test case class. Select Run →Run As → JUnit Test. This will run all the tests within the class you selected.

Alternatively you can write a Test Runner. **Test runner** is used for executing the test cases and its purpose is similar to the client code you have implemented in Lab02. Create the following TestRunner class and test your code this time using your own code. Do you see any difference?

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

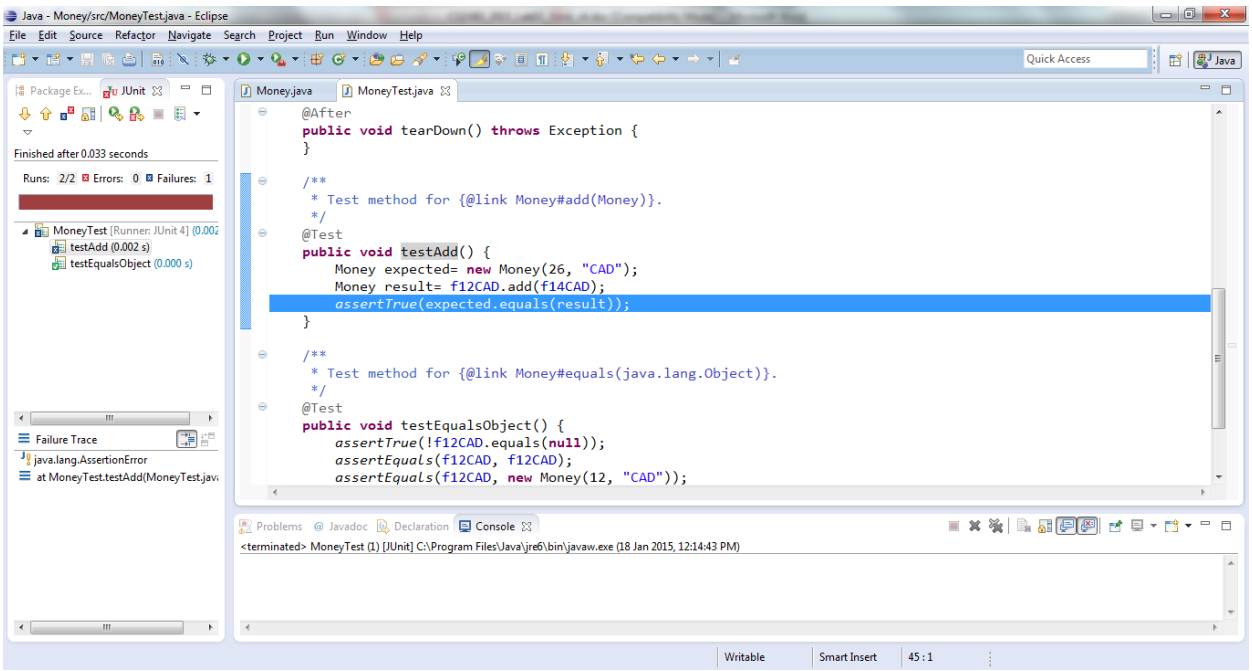
Step 5: Analyse the test results

After the tests have run, you'll see a bar near the top of the left hand pane (as shown in the screenshot below). If it is green, all tests have passed and everything has finished executing. If it is red, some tests have failed. You can see all the tests in the left hand pane, below the bar. Double-click on a test and the navigation will highlight it in your code. At the bottom left hand corner, you can see the test's failure trace, if applicable.



Identifying the errors might be confusing at first. Sometimes there could be a problem with how you designed your test, rather than your code. JUnit will not tell you if your testing is incomplete or if different edge cases have not been covered. It is up to you to figure this out. More often than not, failed tests represent some refinement required in your implementation.

If you followed the instructions in this exercise carefully, then you should see no errors. To simulate an error, make a change to the add function and subtract the two values instead. Run the tests again. Remember to double click the failed test on the left hand side to navigate to the test. Eclipse shows you exactly where your test failed.



Test Suites

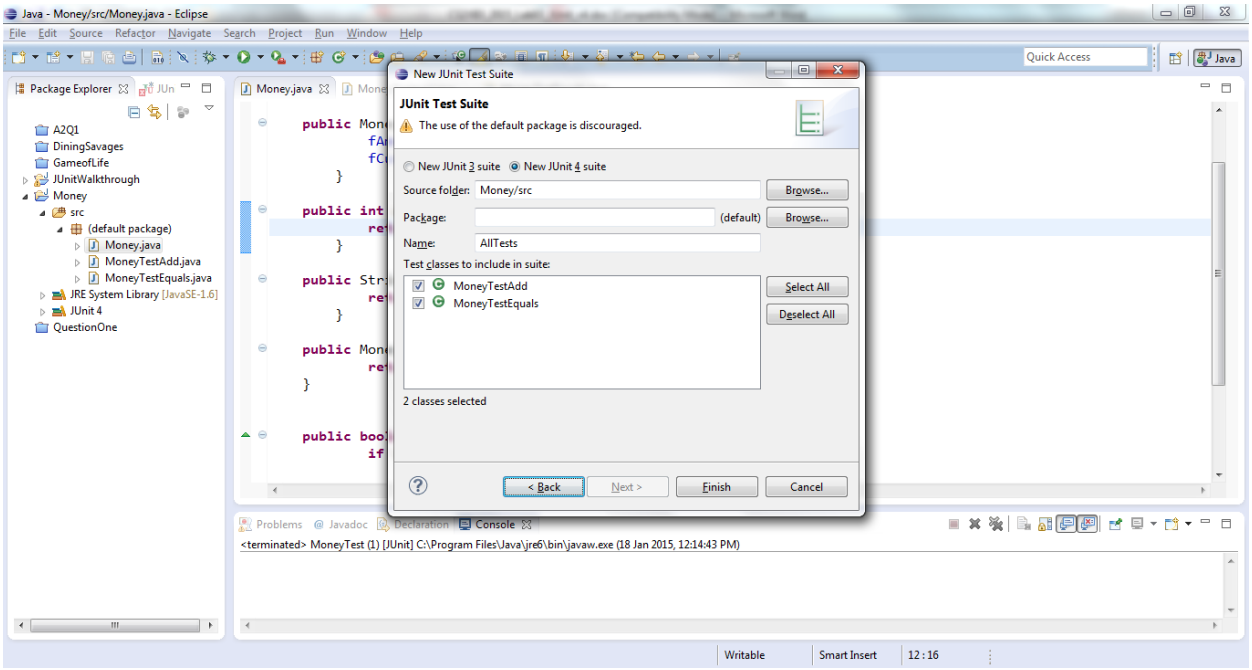
A test suite is a collection of test case classes that can be tested in a single batch. This is a simple way of running one program that runs all test cases at once.

Test Suites are used for a number of test case classes that need to be run together. For example, you might have a separate class for each method in your implemented class because each method needs a different environment in which to run, so the setUp and tearDown methods would be different. For each environment you use, you could have a test case class that tests the necessary methods in that environment.

Using the Eclipse wizard is the easiest method. Again, the underlying code is just generating code stubs to glue all your tests together.

For the above Money class, say you have one class MoneyTestAdd that tests only the add method, and a second class called MoneyTestEquals that only tests the equals method implemented in your Money class.

To create a test suite in Eclipse, with your cursor in the Money class, go to File→New→Other…→ Java → JUnit → JUnit Test Suite… and click Next to select the test case classes you want to include in your suite(both MoneyTestAdd and MoneyTestEquals). Make sure you've selected "New JUnit4 Suite", change the name of the suite if you wish, and then click finish.



Quite simply right-click on your Test Suite class and select Run →Run As → JUnit Test. This runs both your classes together and the feedback is provided in exactly the same way as before.

Within this new class you can orchestrate which tests would be run before others and under what conditions. But to run all your classes together, you don't need to alter any code in the generated class.

Alternatively you can do all of this this manually, as described below.

Suppose you have 3 test case classes to run together: TestJUnit1, TestJUnit2 and TestJUnit3, and you would like to add the class stubs that glue them together manually. Create the following class. The name is up to you, this is just an example.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@Suite.SuiteClasses({
           TestJUnit1.class,
           TestJUnit2.class,
           TestJUnit3.class
})
public class AllTests {
}
```

Run this class. It allows you to run all test cases included in the suite as well as get some extra information such as the number of test cases run or the warnings that have come up during the tests. To find out how to do this, look up the online documentation for Test Suites using JUnit4.

5. Further Practice Problems

For further practice, write test suits for the following classes.

5.1 Start working on Exercise 1.3.4

"Write a stack client Parentheses that reads in a text stream from standard input
and uses a stack to determine whether its parentheses are properly balanced. For example,
your program should print true for [()]{}{[()()]()} and false for [(])."

First identify what are the permutations of required behaviours of your stack client.
Second create a tabular expression of the specifications (hint: you have created a tabular expression for solving quadratic equations for your Software Engineering Principles course).
Third write Junit tests to assert if the program behaves as expected.

5.2 Write JUnit test for the following implementation of Counter

```
public class Counter implements Comparable<Counter> {
      private final String name; // counter name
      private int count; // current value
      // create a new counter
      public Counter(String id) {
      name = id;
      }
      // increment the counter by 1
```

```
	public void increment() {
	count++;
	}
	// return the current count
	public int tally() {
	return count;
	}
	// return a string representation of this counter
	public String toString() {
	return count + " " + name;
	}
	// compare two Counter objects based on their count
	public int compareTo(Counter that) {
	if (this.count < that.count) return -1;
	else if (this.count > that.count) return +1;
	else return 0;
	}
}
```

5.3 Write JUnit test for the following implementation of VarianceAccumulator
VarianceAccumulator
Write test suits to validate that the following code computes both the mean and variance of the numbers presented as arguments to addDataValue() (Ex. 1.2.18 2C03 textbook):

```
public class VarianceAccumulator
{
	private double m;
	private double s;
	private int n;
	public void addDataValue(double x)
	{
		n++;
		s = s + 1.0 * (n-1) / n * (x - m) * (x - m);
		m = m + (x - m) / n;
	}
	public double mean()
	{
		return m;
	}
	public double var()
	{
		return s/(n - 1);
	}
	public double stddev()
	{
		return Math.sqrt(var());
	}
	public String toString()
	{
		return "Mean (" + n + " values): " + String.format("%7.5f",
```

```
mean());
      }

      public static void main(String[] args)
      {
            int n = Integer.parseInt(args[0]);
            VarianceAccumulator a = new VarianceAccumulator();
            double[] v = new double[n];
            double total = 0;
            for (int i = 0; i < n; i++)
            {
                  double x = StdRandom.uniform();
                  v[i] = x;
                  total += x;
                  a.addDataValue(x);
            }
            double mean = total / n;
            double s = 0;
            for (int i = 0; i < n; i++)
            {
                  double d = v[i] - mean;
                  s += d * d;
            }
            double stddev = Math.sqrt(s / (n-1));
            StdOut.println(a.mean() - mean);
            StdOut.println(a.stddev() - stddev);
      }
}
```