**CMPS 12M**

**Introduction to Data Structures**

# Lab Assignment 3

## Objective

The purpose of this assignment is to learn how to use unit tests in Java and get some practice developing code in a test-driven-development way. You'll write an RPN calculator class to pass a suite of unit tests.

## Unit Testing

Since you've written at least one program before, you know that programs often have bugs. Even if you manage to get your code to compile and it looks like everything works when you run the program, it is incredibly easy to make mistakes in the code that lead to buggy behavior. The more complicated the program the more likelihood of bugs.

As you write bigger and more complicated programs made up of multiple classes and components it becomes harder to track down where problems originate. A mistake in the output of the final program could be caused by any of the many components that make up the program.

One way to make debugging easier is to test each component by itself to check for bugs. This type of testing is called *unit testing*. A single unit test is a self-contained test that verifies one aspect of behavior of one component or class in a software project. Unit tests are normally written to cover all the required behaviors of the class to be as comprehensive as possible.

Unit tests are separate from the main application code. Once the application is compiled it does not contain any unit tests. Typically a different compile step is used to compile unit tests into a test suite that can be run by a specialized test runner. A typical workflow is for a programmer to compile unit tests, run all the tests, and verify they are all passing. As long as all unit tests are passing the programmer can gain some assurance that many types of bugs are not present in the code. Of course if there is no unit test verifying some aspect of the code's behavior there is no assurance about that particular behavior.

As an example, consider the following Java class that defines a class to keep track of running total:

```
// Sum.java
// Define a class for adding numbers.

public class Sum {
    // Keep track of running total
    private int value;

    // Constructor
    public Sum() {
        value = 0;
    }

    // Get current value
    public int getValue() {
        return value;
    }

    // Add something to running total
    public void add(int v) {
        value += v;
    }

    // Reset total
    public void reset() {
        value = 0;
    }
}
```

This class by itself is not a full application. Even so we can write unit tests for it and run the tests to verify that the class works as intended.

# JUnit Tests

The unit test framework we will use is JUnit. JUnit is popular for Java code and is widely supported in IDEs and by other Java tools. To use JUnit we write a class called SumTest that includes methods to do unit tests on Sum. By convention a class that contains the unit tests for *Class* is called *Class*Test. Each method has an annotation @Test that indicates it is a unit test.

Here is a simple test suite that includes two tests:

```
// SumTest.java
// Unit tests for Sum class

import org.junit.*;
import static org.junit.Assert.assertEquals;

public class SumTest {

    @Test
    public void oneNumberTest() {
        Sum s = new Sum();
        s.add(10);
        assertEquals(10, s.getValue());
    }

    @Test
    public void twoNumberTest() {
        Sum s = new Sum();
        s.add(3);
        s.add(12);
        assertEquals(15, s.getValue());
    }
}
```

The first test, oneNumberTest(), works by creating an instance of Sum, calling the add() method to add 10 to the running total, then testing that the getValue() method returns 10. The assertEquals function is part of JUnit; it verifies that both inputs are the same value and throws an exception if they are not. The second test, twoNumberTest(), creates a new Sum object then calls add() twice to keep a running total of two numbers. It then checks that getValue() has the correct running total.

# Compiling JUnit Tests

The original code that is being unit tested requires no modifications to add unit tests. It does not require an import of JUnit. The new unit tests are put in a new file and define a new class then ends in Test. The unit tests do require importing JUnit and compiling with the appropriate JUnit libraries.

First get the JUnit library itself and put a copy of the library in your ITS lab3 directory:

https://github.com/junit-team/junit/releases/download/r4.12-beta-2/junit-4.12-beta-2.jar

JUnit depends on Hamcrest (a comparison library), so get a copy of that as well:

http://search.maven.org/remotecontent?filepath=org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar

The libraries are also available in the AFS directory:

```
/afs/cats.ucsc.edu/users/r/nwhitehe/cmps12/lab3/
```

Once the jar libraries are in your current directory together with your source files you can compile unit tests with the "-cp" option to specify classpath.

```
javac -cp "junit-4.12-beta-1.jar:hamcrest-core-1.3.jar:." SumTest.java
```

Nothing special is need to compile Sum.java.

# Running JUnit Tests

To run JUnit tests you need to use the correct classpath and specify a particular runner class within JUnit together with your unit test class.

```
java -cp "junit-4.12-beta-1.jar:hamcrest-core-1.3.jar:." org.junit.runner.JUnitCore SumTest
```

You also need to have the class you wish to test (in our case Sum) compiled and ready to load.

If you succeed in running the unit tests you should see something like this:

```
JUnit version 4.12-beta-1
..
Time: 0.006

OK (2 tests)
```

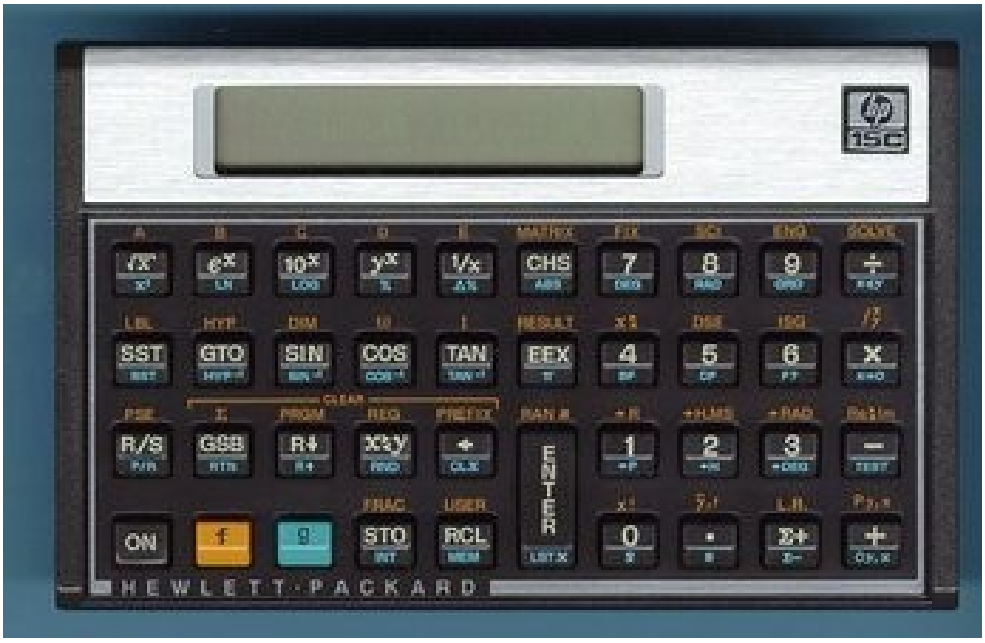
If any of the tests fail you will see more output describing what happened.

Try modifying one of the unit tests to check an incorrect final sum (for example change the 10 to 11 in the first test). Recompile the unit tests and run them to see the failure. Write a new unit test that checks keeping a running total of 3 different numbers, where one of them is negative. Compile and run your new suite of unit tests.

# Test Driven Development

One style of development is called TDD (test driven development). The idea is to always start by writing unit tests and follow by writing code to make the tests pass. By writing the tests first it forces you to understand what you are trying to accomplish. Focusing the coding effort on making tests pass means less time is wasted on features and generality that is not actually required for the project. Once the tests pass it also gives you the confidence to refactor and simplify your code without worrying about breaking anything. If something breaks you will notice immediately when a unit test fails.

# RPN Calculators




Reverse Polish notation (RPN) is a style of calculation invented by the Polish logician Jan Łukasiewicz. It has the advantage that parentheses are eliminated. Operators are placed after operands which makes it easier to parse than standard infix notation.

For example, the expression 2 * 3 + 4 * 5 with normal precedence is written in RPN as:

2 3 * 4 5 * +

The famous HP-15C calculator made RPN famous. You can try out the calculator online here:

http://hp15c.com/

To calculate 2 * 3 + 4 * 5, you would click 2, Enter, 3, *, 4, Enter, 5, *, +.

# Task

Your task is to implement a Calc class that does RPN arithmetic on doubles. You are given a set of unit tests that your implementation must pass. Unit tests are available in the file:

`/afs/cats.ucsc.edu/users/r/nwhitehe/cmps12/lab3/CalcTest.java`

Your Calc class must have the following methods:

| | |
|---|---|
| push | Push a number on the stack (ENTER key) |
| pop | Remove top element from stack, return it |
| peek | Return top element of stack without removing it |
| add | Replace top two elements of the stack with their sum |
| subtract | Replace top two elements of the stack with their difference |
| multiply | Replace top two elements of the stack with their product |
| divide | Replace top two elements of the stack with their quotient |
| depth | Return the number of elements currently in the stack |

Your calculator class should use double precision to keep track of numbers. It must be able to handle up to 100 numbers in the stack. If an operation would be invalid, such as an "add" when there is only one element in the stack, your method must throw an exception.

Once your implementation is passing all the unit tests, add three new unit tests.

- One should compute a compound expression with multiple operations and verify that the answer is correct.
- The second should do the same computation half way through, then verify that depth() and peek() return the correct values for the partially completed expression.
- The third should do something that is required to throw an exception that is not an exact duplicate of an existing unit test.

Verify that your implementation passes your new unit tests.

Now add a new feature using TDD. The new feature is the "1/x" button which transforms the number on top of the stack to its reciprocal; call the method "reciprocal". First write a new unit test using the feature. Modify your implementation to have an empty method for reciprocal. Verify that with your new test the unit tests are compiling but failing. Then implement the feature to make the test pass.

## What to Turn In

All files you turn in for every assignment and lab should begin with a comment block that includes your name, CruzID, class, date, filename, short description of the file's role in the assignment, and any special instructions related to the file. Also create a file called README. The README file should have the normal comment block, then list all the files being submitted (including itself) along with any special notes to the graders.

For this lab, submit the following files:

```
README
Makefile
Calc.java
CalcTest.java
```

To submit, use the submit command.

```
submit cmps12b-nojw.f14 lab3 README Makefile Calc.java CalcTest.java
```

You do NOT need to submit the JUnit or Hamcrest libraries.

Your makefile should have a default target that builds Calc.class and a phony target "test" that builds and runs all the unit tests.