Group 5 Report

Simple Calculator Program

**Group members**: Vincent Prestianni , Michael Sukantawanich, Edin Kalezic

**Repository**: https://github.com/Vinniesusername/SimpleCalc.git

**Project Mission**: Creating a simple useful calculation tool for easy mathematics and quick summaries.

**Goals**: Creating a functional calculator code using Java language, using NetBeans GUI tools in construction and testing.

**Architecture and Design**: Vincent Via Open Source Code in which we all agreed to start a new calculator code from scratch.

**Research**: Group effort

**Coding**: Main coding: Vincent

**App Testing & Functionality**: Michael

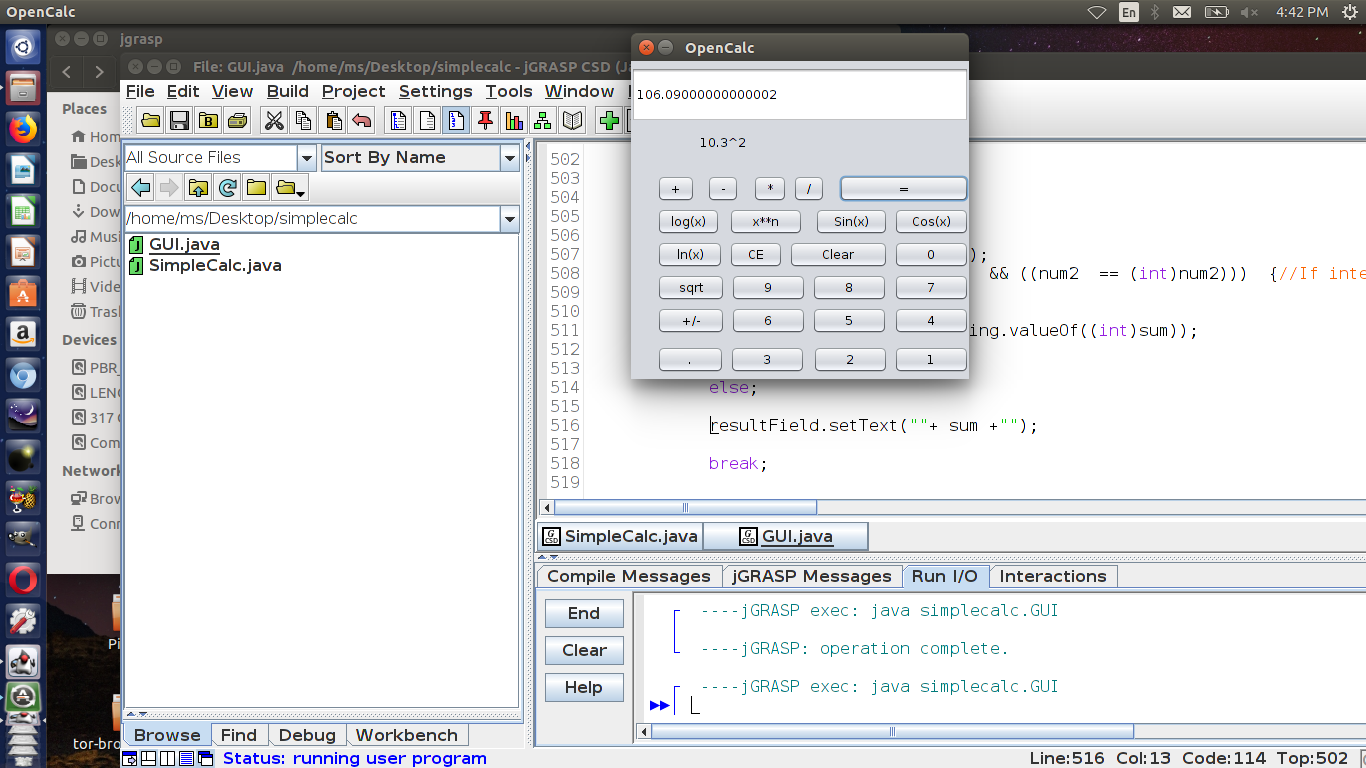
**Documentation**: Edin

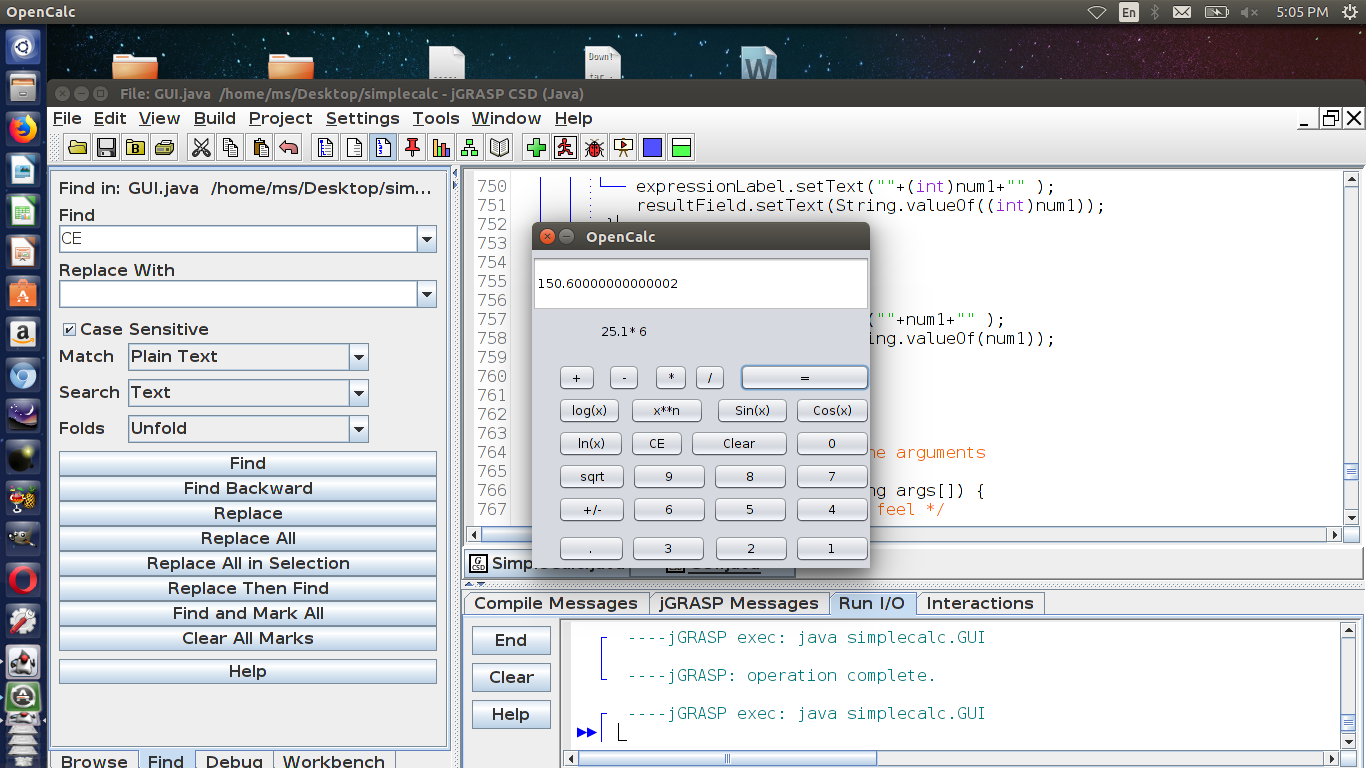
**Group Meetings and communications**: As a group we met in class and communicated via Email during the course of this project.

The calculator operates as expected with basic mathematical calculations, but we ran into an error of the decimal placement during testing.

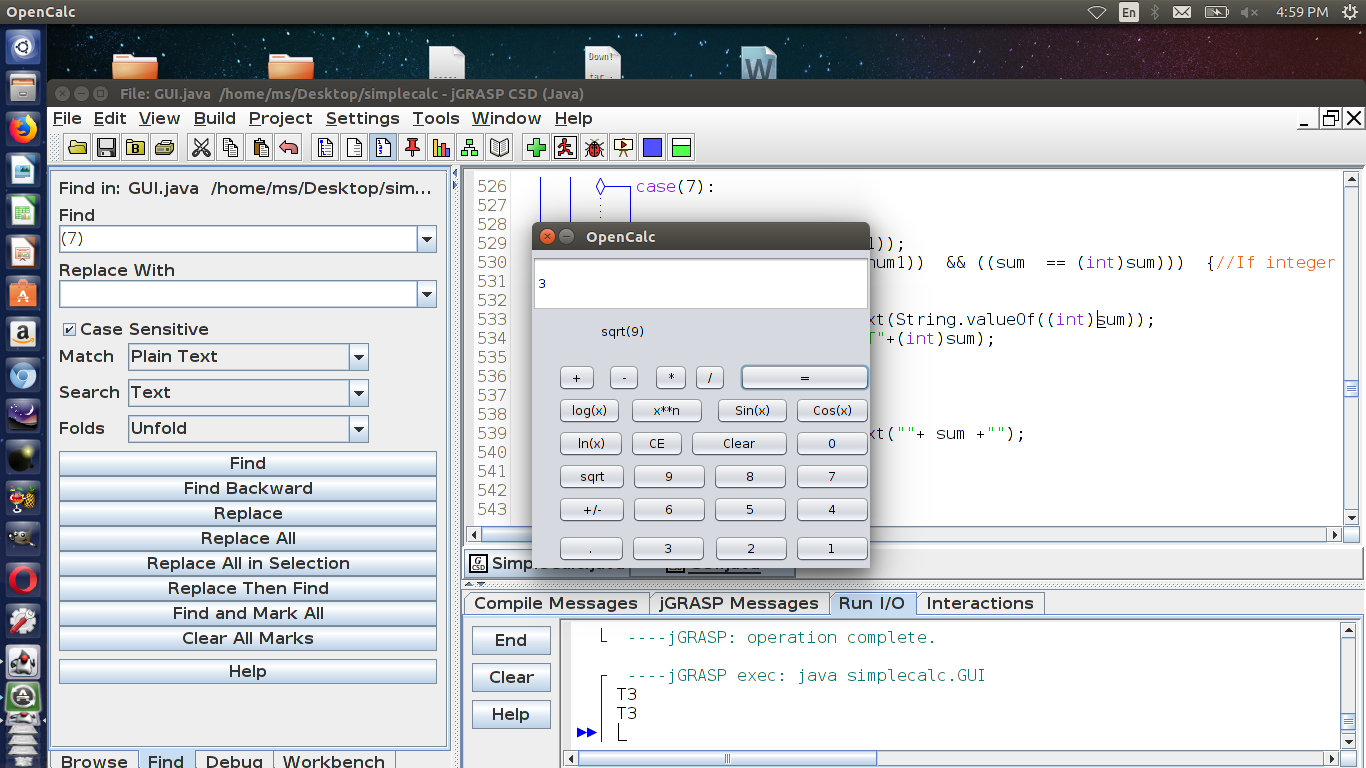
Looking into resolutions in correcting the error.

Double precision accuracy during testing (Decimal Error)



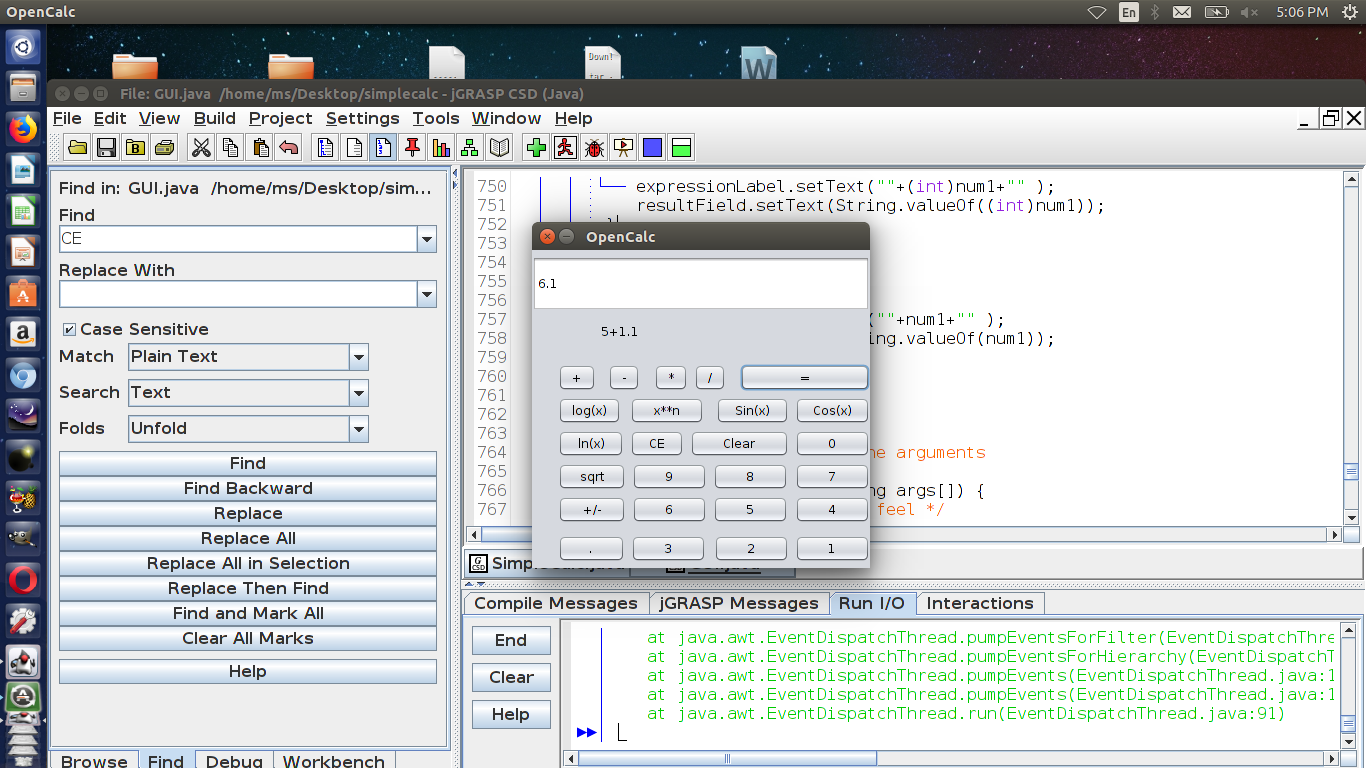
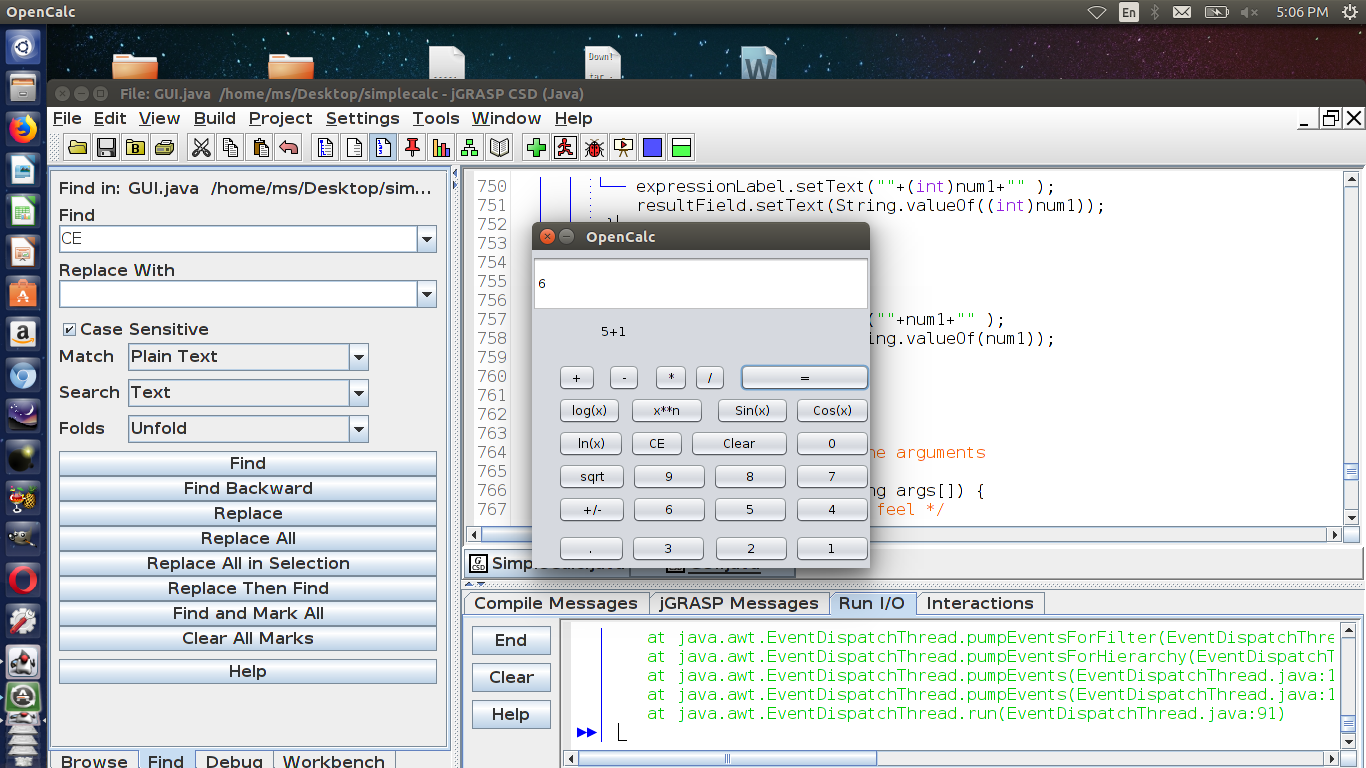


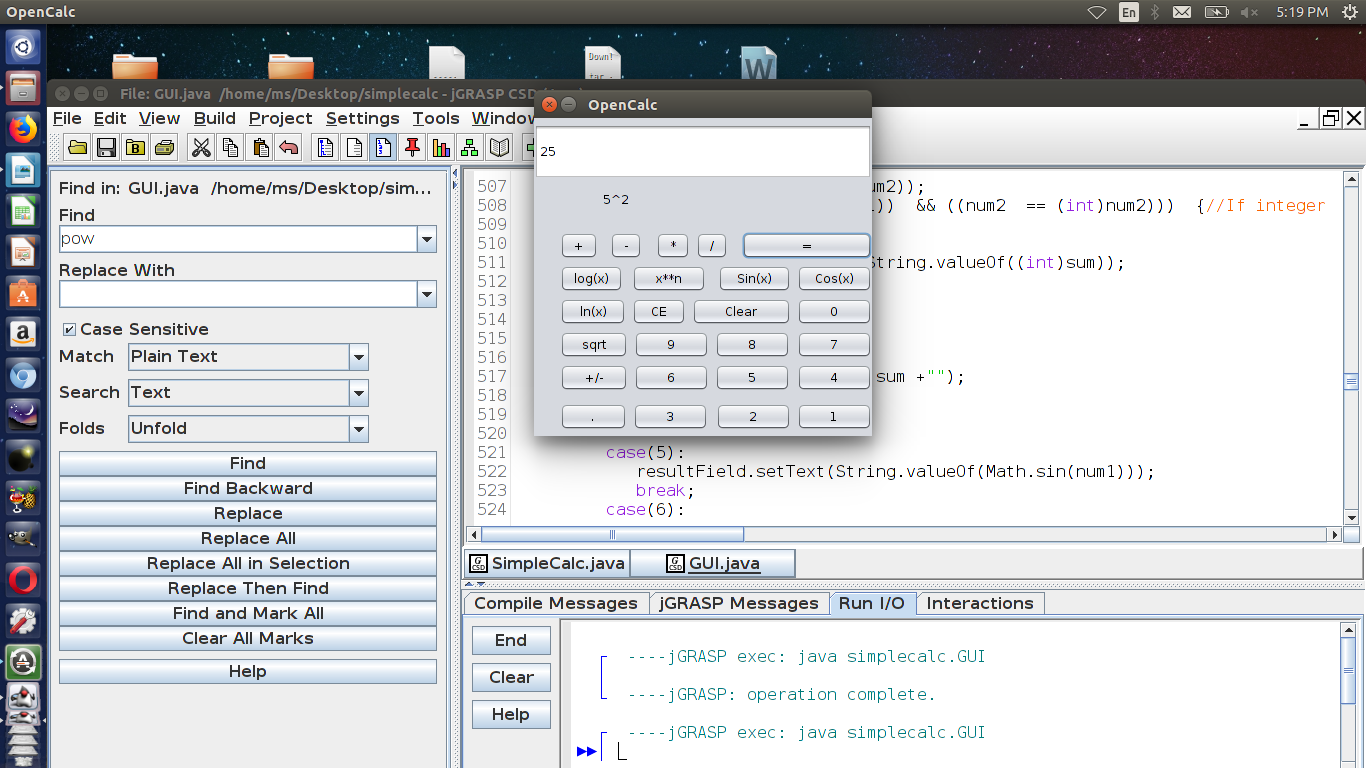
Integer operations with integer result



Type Casting Answer (Int + Int = Int)

(Int + double = double)





Accuracy Issues:

The fact that floating-point numbers cannot precisely represent all real numbers, and that floating-point operations cannot precisely represent true arithmetic operations, leads to many surprising situations. This is related to the finite precision with which computers generally represent numbers.

For example, the non-representability of 0.1 and 0.01 (in binary) means that the result of attempting to square 0.1 is neither 0.01 nor the representable number closest to it. In 24-bit (single precision) representation, 0.1 (decimal) was given previously as e = −4; s = 110011001100110011001101, which is

0.100000001490116119384765625 exactly.

Squaring this number gives

0.010000000298023226097399174250313080847263336181640625 exactly.

Squaring it with single-precision floating-point hardware (with rounding) gives

0.010000000707805156707763671875 exactly.

But the representable number closest to 0.01 is

0.009999999776482582092285156250 exactly.

Also, the non-representability of π (and π/2) means that an attempted computation of tan(π/2) will not yield a result of infinity, nor will it even overflow. It is simply not possible for standard floating-point hardware to attempt to compute tan(π/2), because π/2 cannot be represented exactly. This computation in C:

/\* Enough digits to be sure we get the correct approximation. \*/

double pi = 3.1415926535897932384626433832795;

double z = tan(pi/2.0);

will give a result of 16331239353195370.0. In single precision (using the tanf function), the result will be −22877332.0.

Possible Solution to the Decimal Error Researched:

BigDecimal operations

BigDecimal methods for adding and subtracting numbers are add() and subtract(), respectively. For example, to add 1,115.37 and 115.37, we could do the following:

BigDecimal balance = new BigDecimal("1115.37");

BigDecimal transaction = new BigDecimal("115.37");

BigDecimal newBalance = balance.add(transaction);

The BigDecimal's newBalance object now holds the value of 1,230.74. Similarly, to subtract 115.37 from 1,115.37, we could use this code:

BigDecimal balance = new BigDecimal("1115.37");

BigDecimal transaction = new BigDecimal("115.37");

BigDecimal newBalance2 = balance.subtract(transaction);

The BigDecimal's newBalance2 object now holds the value of 1,000.00. (Naturally, if we are talking about checkbook balances in real life, the subtract() method will be used much more often than the add() method, and the total amount subtracted from the checkbook balance will exceed the total amount added, or so it often seems.) You can accomplish multiplying and dividing with BigDecimal's multiply() and divide() methods. Multiplying is demonstrated in the following program:

import java.math.\*;

import java.text.\*;

import java.util.\*;

public class Multiply {

public static void main(String[] args) {

BigDecimal d = new BigDecimal("1115.32");

BigDecimal taxRate = new BigDecimal("0.0049");

BigDecimal d2 = d.multiply(taxRate);

System.out.println("Unformatted: " + d2.toString());

NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);

double money = d2.doubleValue();

String s = n.format(money);

System.out.println("Formatted: " + s);

}

}

The output for the above code is shown below:

Unformatted: 5.465068

Formatted: .46

Note the extra decimal places in the unformatted BigDecimal object as compared to the formatted output. In addition, formatting the value of the BigDecimal object causes the fraction -- greater than one half -- to be dropped. To manage the extra decimal places and the lack of rounding, we can use BigDecimal's setScale() method to set the number of decimal places. When using setScale(), we need to specify not only the number of decimal places, but how the number will be rounded, if rounding is necessary. The most common way of rounding -- round up fractions half or greater, and round down all other fractions -- can be specified with BigDecimal's constant ROUND\_HALF\_UP. Therefore, to set the number of decimal places to two and specify that fractions half and greater will be rounded up, we can write:

d2 = d2.setScale(2, BigDecimal.ROUND\_HALF\_UP);

Modifying the above program to add setScale(), we now have:

import java.math.\*;

import java.text.\*;

import java.util.\*;

public class Multiply2 {

public static void main(String[] args) {

BigDecimal d = new BigDecimal("1115.32");

BigDecimal taxRate = new BigDecimal("0.0049");

BigDecimal d2 = d.multiply(taxRate);

d2 = d2.setScale(2, BigDecimal.ROUND\_HALF\_UP);

System.out.println("Unformatted: " + d2.toString());

NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);

double money = d2.doubleValue();

String s = n.format(money);

System.out.println("Formatted: " + s);

}

}

Now the output is:

Unformatted: 5.47

Formatted: .47

Now the BigDecimal value is rounded to two digits, rounding the value up, and the formatted String correctly displays the rounded value. Other constants useful in rounding are ROUND\_HALF\_DOWN and ROUND\_HALF\_EVEN. The first, ROUND\_HALF\_DOWN, rounds fractions of half and under down, and all others up. The second, ROUND\_HALF\_EVEN, rounds half fractions to the even number (e.g., 2.5 rounds to 2, while 3.5 rounds to 4), and fractions greater or less than half to the closest integer. When dividing BigDecimal objects, we are required to specify how the result will be rounded. For this article, we will round halves up. The following program shows some sample division:

import java.math.\*;

import java.text.\*;

import java.util.\*;

public class Divide {

public static void main(String[] args) {

BigDecimal d = new BigDecimal("1115.32");

BigDecimal days = new BigDecimal("30");

BigDecimal d2 = d.divide(days, 2, BigDecimal.ROUND\_HALF\_UP);

NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);

double money = d2.doubleValue();

String s = n.format(money);

System.out.println(s);

}

}

Output from the above program is:

7.18

Calculating interest

For this example, assume that a sum of ,500 will receive interest payments at an annual rate of 6.7 percent. Payments will be calculated quarterly, and we will calculate the first quarterly payment. To do so, we will use the formula I=PRT, where I is the amount of interest, P is the principal (9,500), R is the rate (6.7 percent annually), and T is the time (0.25 years). The program is:

import java.math.\*;

import java.text.\*;

import java.util.\*;

public class Interest {

public static void main(String[] args) {

BigDecimal principal = new BigDecimal("9500.00");

BigDecimal rate = new BigDecimal("0.067");

BigDecimal time = new BigDecimal("0.25");

BigDecimal temp = principal.multiply(rate);

BigDecimal interest = temp.multiply(time);

NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);

double money = interest.doubleValue();

String s = n.format(money);

System.out.println("First quarter interest: " + s); }

}

Output from the above program is:

First quarter interest: 59.12

**Actual Solution used:**

**Testing procedure**

A manual testing procedure was conducted. Numbers of type integer and double were used for each calculator function. For example, 3+2, 3+2.1, 3\*2, 3\*2.1, etc. If the number had a decimal, the sum was displayed as a double type. If the numbers were integers, the sum would be Type Cast as an integer. Also, negative numbers were tested to see if they were mathematically valid. For example -2+2 = 0. The CE button was tested before and after numbers were operated. For example, with before complete operations, 3\*4 then CE would yield 3. With after a completed operation, 3\*4 =12, then CE would also yield 3. Integer and double numbers were tested for each of the mathematical functions buttons.

The calculator’s validity was tested. For example, sqrt(-2.0) = NaN.

The calculator's upper limits were tested, for example, 999999999999 ^ 999999999999 = Infinity.

This was probably the result of the sum variable being a double variable consisting of 64 bits. The previous calculation’s sum exceeded the 64 bit limit.

The calculator’s double precision accuracy was also tested, as a double value is a 64 bit number, consisting of a 52 bit mantissa or decimal. (Venners)

This would lead to decimal issues with calculations such as 10.3^2, mentioned later.

**Works Cited**

“Appendix D.” *Lesson: All About Sockets (The Java™ Tutorials > Custom Networking)*, 5 Apr. 2000, docs.oracle.com/cd/E19957-01/806-3568/ncg\_goldberg.html.

Nielsen, Robert. “Make Cents with BigDecimal.” *JavaWorld*, JavaWorld, 1 June 2001, www.javaworld.com/article/2075315/core-java/make-cents-with-bigdecimal.html.

Venners, Bill. “Floating-Point Arithmetic.” *JavaWorld*, JavaWorld, 1 Oct. 1996, www.javaworld.com/article/2077257/learn-java/floating-point-arithmetic.html.

Overall Function and design work well, simple to use and easy to navigate for the end user to easily understand what button does what and how to operate the calculator.