Programming Exercise 2:

Assignment Statements

Purpose: Declaring/initializing variables, using assignment statements, and using some of the Java standard classes.

Background readings from textbook: Liang, sections 2.5–2.11.

Due date for section 001: Monday, January 25 by 10 am
Due date for section 002: Wednesday, January 27 by 10 am

Overview

The ability to use data stored in a computer's memory in an intuitive way is essential in writing computer programs. However, the way you use this data is fraught with hazards since the same ones and zeroes that the computer sees can be interpreted in many different ways. For example, the binary number

```
1000001
```

may represent the integer 65, it may represent the letter *A*, it may be a part of a program instruction like "add", or it may represent a memory address storing a datum.

In Java you can use a memory location just as you would a variable in mathematics. It has a name, and you manipulate that name as a changeable quantity. Java requires that you *declare* the type of data you plan to store in that variable before you ever use it. Then, each time you use that variable the compiler checks to see if that usage is appropriate for the type you declared your items. Here are a few of the data types you can use.

Java Type Name	Designates	Example
`char`	Single character	`'A'`
`int`	Integer number	`12`
`long`	Large integer number	10^{20}
`double`	Floating point number	`3.14`
`String`	String of characters	`"CSC 260"`

Part 1: Java Examples

1. Declaring variables.

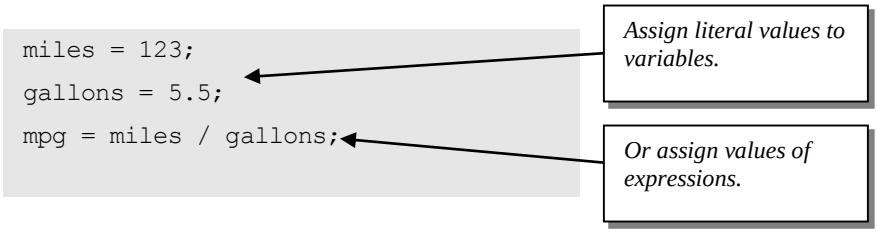
Data type of values stored in the following variables.

```
int miles;
double gallons, mpg;
```

Variable name.

You can declare two or more variables of the same type in one statement.

2. Assigning values to variables.



To do the computations you need to use Java's arithmetic operators:

Operator	Meaning	Example
+	addition	`7 + 5` → `12`
-	subtraction	`7 - 5` → `2`
*	multiplication	`7 * 5` → `35`

/	integer division
	or
	floating point division
		`7 / 5` → `1`
		or
		`7.0 / 5.0` → `1.4`

%	modulo (or mod), which is the remainder of an integer division	`7 % 5` → `2`

Most of these operators are straightforward. For instance, if you want to add three values, x, y, and z, and store the result in sum, you would do `sum = x + y + z;` If you want to add x and y and multiply the result by z and store the result in total, you would use `total = (x + y) * z;` The parentheses are required here because the order of operator precedence is that * happens before +, so without the parentheses, `total = x + y * z;` would perform y * z, add x, and store that value in total.

The mod operator (%) gives you the remainder of an integer division. For instance, 5 / 2 is 2 ½. So 5 % 2 is the remainder, or 1. 16 % 2 is 0 (there is no remainder) while 16 % 9 is 7 (16 / 9 is 1 7/9). The challenging operator to use is division if both the numerator and denominator are integer values. Consider 16 / 9. This gives you the value 1 because the result, 1 7/9 has a quotient of 1 and a remainder of 7. Normally, if we wanted to do 16/9 and store the result as an integer, we would probably round up giving us 2 instead of 1. Now consider the following assignment statement where z is a double, and x and y are int values of 16 and 9 respectively.

```
z = x / y;
```

What does z store? If you did this on a calculator, you would get 1.77777… (or about 1.778). But in Java, z would store 1.0. Why? Since x and y are both int values, x / y is an *integer* division meaning that the result is stored as an integer. The integer division of 16/9 is 1. We are not interested in the remainder when we use /. Since z is a double, the result, 1, is converted to a double, or 1.0. So we lose some accuracy. We need to force Java to perform a *double* division, not an integer division. How? We need to perform a *cast* to convert either the numerator or the denominator to be a double. A cast does not actually change the variable's type, it merely changes the number being sent to the division circuitry in the CPU. A cast is done by placing `(type)` immediately before the variable name where type is the *type* you are casting to (double in our case). Our assignment statement becomes `z = (double)x / y;` or `z = x / (double)y;`

Part 2: Common Pitfalls

1.

```
public class MyClass
{
    public static void main( String[] args )
    {
        x = 10;
    }
}
```

Syntax error!

Variable x is used without being declared. The compiler will give an error message pointing to this line.

2.

```
public class MyClass
{
    public static void main( String[] args )
    {
        Int x = 10;
    }
}
```

Syntax error!

The type, int, is misspelled as it should not be capitalized.

3.

```
Public class MyClass
{
    public static void main( String[] args )
    {
        int x = 10.1;
    }
}
```

Syntax error!

The type, int, does not match the literal value you are attempting to store there. 10.1 is a real number and must be stored under a floating point type: float or double

4.

```
Public class MyClass
{
    public static void main( String[] args )
    {
        int x = 1; y = 2;
    }
}
```

Syntax error!

Although you are able to declare multiple variables and assign them on the same line of code, the semicolon after x = 1 ends this line, so y = 2; is literally a separate instruction and it y will not have been declared as an int..

5.

```
Public class MyClass
{
    public static void main( String[] args )
    {
        int x = 1, y = 2;
        double z;
        z = x / y;
    }
}
```

Logic error!

We should either cast x or y as a double in the assignment statement to ensure that z is storing the actual result of the division. As is, z will store 0 but z should store 0.5. We can fix this by doing either

z = (double) x/ y;

or

z = x / (double) y;

Part 3: Problem

You have been asked by *Black and Gold Construction Company* to write a program to help them compare construction costs on different projects. Your program computes the cost per square foot of floor space for a given building based on its total cost and the building's dimensions. For simplicity, all buildings will be rectangular, given as the building's length and width. The cost of a building is

given as width * length * cost per square foot * floors + base cost of the building. For instance, a building that is 3 floors and is 200x300 feet with a cost per square foot of $11 and a base cost of $250,000 has a total cost of 200 * 300 * 11 * 3 + 250,000 = $2,230,000. For our program, we will know the dimensions of the building, the number of floors, the base cost and the overall cost. We will use these to compute the cost per square foot. Our formula is given below (you should be able to figure out the formula yourself with a little algebra):

```
Cost per square foot = (cost – base cost) /
                       (width * length * number of floors)
```

Note that all input will be given as int values but cost per square foot should be a double. You will have to perform a cast.

Start your program with an identifying comment header similar to the one in Lab 1. Next, declare the following six variables.

```
int length, width, nStories;
int baseCost, totalCost;
double squareFootCost;
```

You will not input the length, width, nStories, baseCost or totalCost from the user. Instead, we will *hard-code* these values in the program using assignment statements, for instance `length = 300;` Initially, use the following values to test your program.

length of foundation	300
width of foundation	200
number of floors	3
base cost	$250,000
total cost	$2,000,000

Next, include your assignment statement to compute the squareFootCost. Remember to provide any proper casts needed so that you obtain an accurate squareFootCost value. Finally, finish the program by outputting the results. Use the following two statements.

```
System.out.println( "The cost per square foot of floor space for a "
                      + length + " by " + width + " building" );
System.out.println( "of " + nStories + " stories with a base cost of "
                      + baseCost + " and a total cost of " + totalCost
                      + " is " + squareFootCost + "." );
```

Notice spaces after words like “a” and “by”. This will ensure that the values stored in variables will be output with spaces around them to make the output easier to read.

Compile and run your program. If everything works correctly, you should see the following text in the console window.

```
The cost per square foot of floor space for a 200 by 300 building
of 3 stories with a base cost of 250000 and a total cost of 2000000 is 9.722222222222221.
```

The output of the cost per square foot looks ugly, nothing like a dollar amount. Java contains many built-in classes that can do all sorts of things for us such as handle keyboard or file input, generate random numbers, create GUI components and also format floating precision numbers. We will use a class to do this last thing in order to handle our output. The class is called **DecimalFormat**. To class this class, we have to do several things.

1. Import the class using an `import` statement. All import statements occur before the class header. Add the following after your initial comments but before your class header. We can also import all of java.text by specifying `java.text.*` instead, but it is more efficient to just import DecimalFormat itself.

```
import java.text.DecimalFormat;
```

2. We must declare a variable of type DecimalFormat and then create an instance of it. To create an instance, we use the word `new`. We can declare the variable and create the instance in one or two instructions. The first way is to do:

```
DecimalFormat df;
df = new DecimalFormat(…);
```

   Combining these, we get

```
DecimalFormat df = new DecimalFormat(…);
```

   Note that the … in parentheses is supposed to be a String describing the format we want to use. Our format should specify that this is a dollar amount, so it starts with $, and then we want to specify the number of digits we expect on the left and right side of the decimal point. To say "output a 0" we use 0, and to say "output a digit unless it is 0", we use #. The reason for this difference is that the number 1234.5678 would output as 0001234.5678 if we insist on 7 digits before the decimal point but if we use #, any leading zeroes are omitted. On the other side of the decimal point, we can force trailing zeroes to appear if desired by using 0 in our format. If our number is 1234.5, we can force this to appear as 1234.50. Our format will be `"$#,###.00"`. This means that if the number has 4 digits on the left of the decimal point, we automatically get a comma. If the number has fewer than 4 digits, there are no leading zeroes and no comma. Add the proper instruction to your program. This can appear anywhere in your main method before your last `System.out.println` statement but it is best to put it with your other variable declarations, either before `int length` or after `double squareFootCost`.

3. We modify the output of squareFootCost to use this formatting. We do so by replacing squareFootCost in the println statement with `df.format(squareFootCost)`.

Save, compile and run your program. Does the output look nicer?

4. We will also add the date to your output. Fortunately, there is another Java class called Date that we can use. It is located in java.util. Add the proper import statement to import Date.

5. We need to declare a variable of type Date and create an instance of it. This will be similar to what you did for DecimalFormat and df above except that there will be nothing in parentheses, that is, the right side of the assignment statement will be `= new Date( );`

6. Add an output statement before the other two statements to output: "Date of estimate: " followed by the date and two blank lines. Recall you can output a blank line using `\n`.

Save, compile and run your program to make sure it works.

7. Finally, add a String variable for the client's name. Assign it the value of "Frank Zappa". Add to your output the date with the client's name. It might look like the following:

```
Date of estimate is Wed Jan 06 10:01:17 EST 2016 for Frank Zappa
```

Part 4: Test Your Program

Test your program by changing the values to the new ones given below; recompile and run your program after each change to see if you get the same output as shown.

Input				Output
Foundation Dimensions	**Number of Floors**	**Base Cost**	**Total Cost**	**Cost Per Square Foot of Floor Space**
510 x 722	9	$950,000	$169,583,110	$50.89
251 x 161	1	$100,000	$281,475	$4.49

If you do not have this output, reconsider the logic you used in your computations. Fix the logic problems and run your program again. Continue the debugging process until all of your results are correct.

Part 5: Another Enhancement

As a last step in this assignment, we want to compute the average of two cost per square foot values for two buildings. We will need to store two squareFootCost values, so rename squareFootCost to be squareFootCost1 and declare a second variable called `squareFootCost2`. Also add another double variable called `average`.

Make sure you change your assignment statement and the output statement for squareFootCost to use squareFootCost1. After your output statements, assign your variables (length, width, nStories, baseCost, totalCost) new values. In this case, the first computation will be for the first building above and the second will be for the second building. Copy and paste your assignment statement and output statements at the bottom of your program, substituting squareFootCost with squareFootCost2. Now, add two more instructions, another assignment statement and an output statement. The assignment statement will compute the average of the two squareFootCost variables. The output statement should output two blank lines (using \ns) and then output the computed average, using df.format to output it using a proper dollars and cents formatting.

Part 6: Submitting Your Assignment

After running the program, copy the output and paste it as a comment at the bottom of your source code. Print out and hand in or email your source code to your instructor.