Sequential Program Execution

<table>
<thead>
<tr>
<th>Quick Start</th>
<th>Compile step</th>
</tr>
</thead>
<tbody>
<tr>
<td>once always</td>
<td>javac Realtor1.java</td>
</tr>
<tr>
<td>cd labs</td>
<td>Execute step</td>
</tr>
<tr>
<td>mkdir 1</td>
<td>java Realtor1</td>
</tr>
<tr>
<td>cd 1</td>
<td></td>
</tr>
<tr>
<td>cp ../0/Realtor.java Realtor1.java</td>
<td>Submit step</td>
</tr>
<tr>
<td>cp /samples/csc/156/labs/1/* .</td>
<td>submit csc156abc 1</td>
</tr>
<tr>
<td>emacs Realtor1.java &amp;</td>
<td></td>
</tr>
</tbody>
</table>

Maintenance of Existing Code

As software ages, the rest of the computational world moves on at a healthy pace. Hardware becomes faster allowing software developers to implement new features that users wish to purchase. Think of the software that was on your first cellular phone or your first digital music playback device such as an iPod. Some software features were the same, typically how to place or receive a phone call. Other software features may not have existed on your first device but may be present on your current one, such as new menu options. The whole notion of a downloadable application is nothing more than the installation of software to enhance the base system that you purchased with your device.

One of the first employment opportunities that computer programmers are often hired for is that of software maintenance. Essentially, take a program that is currently working (or was recently working) and modify it to demonstrate a new feature that users desire. It could be that we’re interested in changing the computation, or the screen color, or the changes can be more substantial than that. The fact is that our world relies upon millions of lines of working software, some of it is implemented in very old languages. We need an army of programmers to maintain that software so that we can keep up with where we came from, and more importantly, move forward with where we’d like to go.

Problem Statement and Specifications

The first step in the programming process involves communication. Typically, the individual who needs the problem solved is not the one who codes the program. Consequently a clear statement of the software’s goals and limitations should be written in English and shared between the users of the software who require it and the developers of the software who will create it. Once this process has begun, the developer, often in consultation with the user, will need to determine what values are to be input and what values are to be output. Usually, the values to be output are made clear by the specification. Input values should be limited to those that are required for the program that can not be calculated. For this class, your instructor is the individual who will provide you with a program statement and specifications.

Assignment 1 Statement: Write a program in a file named Realtor1.java that determines and displays the cost and commission involved with a home sale. These
values are calculated using 6% and 1.5% of the price of the home, respectively. Use the JOptionPane methods for all input and output.

\[ \Delta \nabla \frac{1}{2} \text{ lab extra credit} \ \Delta \nabla \]

Copy your solution to a file named Realtor11.java and further modify the program to read all input from the file Realtor11.txt and write all output to the file Realtor11.out. Displayed output to the file should be identical to the given assignment. Modify your compilation, execution and submission steps to \texttt{javac Realtor11.java, java Realtor11, and submit csc156abc 11, respectively.}

This is identical to the last assignment in our course, with the exception of modifying the input and the output. This would be a good time to closely examine the specifications of that exercise.

Assignment 1 Specifications:

Input  
the owner or seller  
the selling price  

Output  
the input variables seller and price  
the cost of selling the house  
the commission for selling the house  

Note that at this time, we do not have the computational tools to verify that the price is positive. We'll have to rely upon the user to ensure that that condition is met.

Object Analysis and Algorithmic Development

Before we begin the development of our algorithm, we will need to consider the data objects that we are using. Within object oriented programming languages, objects are considered to be the data to be stored as well as the methods that manipulate the data. For now, you can think of the methods of the Math class such as Math.sqrt(x) or Math.pow(x) when we refer to methods. In this exercise, all numeric data objects will be real valued. This will often be the case with programs that implement monetary algorithms. Consequently, all of the variables referred to above, except seller, will need to be declared as type double. The only other data objects that our program will use will be the seller and the messages in the output statements. These message values will be constant and are referred to as String.

A common place to begin the development of algorithms is to start with a general set of goals and refine them. This is often referred to as the Top Down Design strategy. Almost all of our algorithms can begin with the following steps.

1. Input  
2. Process  
3. Output  

Fortunately, we’ve already analyzed the input and the output. The processing that is required is covered in the last exercises’ opening discussion. When performing input from the keyboard, it is necessary to inform the user what is expected of them. For instance, our program has 2 values to be input and the user needs to be instructed as to their order. These instructions are referred to as prompts and merely consist of an output statement that gives the user directions. So, every time that our program requires input, we will need to prompt the user for it.

There is some freedom to the sequence of the steps involved with an algorithm, but some attention must be paid to any dependencies amongst the calculated values. To some extent, that dictates what order our calculations can proceed in. Let’s start at the end of our program and by analyzing the formulas we’ll use, see if we can spot the dependencies. To determine the cost and the commission associated with the sale of the home, we need
to know the price since both are calculated as percentages of that value. As long as we’re inputting the price, we might as well input the seller at the same time. This dependency leads us to the following sequence of statements.

1. *input* seller, price
2. *process* to determine cost
3. *process* to determine commission
4. *output* seller, price, cost, commission

Note that we’ve faithfully covered the order of the **input-process-output model**. However, this will not always be the case, but it does represent a common starting place when you are not sure how to begin working on an algorithm. Since each input statement will require a prompt so that the user understands when which values are needed, we can refine this to the following. Lines that are displayed in blue represent areas where you will need to modify or write code to complete the exercise.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Data objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) display an informational message</td>
<td>String constants</td>
</tr>
<tr>
<td>2) prompt for &amp; input seller</td>
<td>String constant &amp; variable seller</td>
</tr>
<tr>
<td>3) prompt for &amp; input price</td>
<td>String constant &amp; double variable price</td>
</tr>
<tr>
<td>4) calculate cost</td>
<td>double constant &amp; variable price</td>
</tr>
<tr>
<td>5) calculate commission</td>
<td>double constant &amp; variable price</td>
</tr>
<tr>
<td>6) display input values</td>
<td>String constants &amp; variable seller &amp; double price</td>
</tr>
<tr>
<td>7) display calculated values</td>
<td>String constants &amp; double variables cost &amp; commission</td>
</tr>
</tbody>
</table>

We’ll simply mention that as to the extra credit, there is no need to prompt files for input. Files give up the data without asking, however it is the programmer’s responsibility to know the order of the data that is in the file.

**Coding for Compilation**

Login to your UNIX account and create a new subdirectory of the labs directory called 1 to work on your assignment.

```bash
cd labs
mkdir 1
cd 1
```

The first thing to code in your program is the documentation. At the top of the program should be the author, the date, the purpose and the specifications. This is similar to what appeared at the top of your typing exercise and is already at the top of the file stored in your ~/labs/0/Realtor.java file that you can copy over to your local directory by issuing the following cp command. Comments in the Java language come in two varieties. Single line comments begin immediately after the double slash (//) characters. Comments in the Java language come in two varieties. Single line comments begin immediately after the double slash (//) characters. Multiple line comments can span several lines that begin with the /* and end with the */ symbols. The comments at the top of Realtor.java are of this type. Throughout your code, you should create comments of the pseudo code near the lines that execute those actions. Single line comments are appropriate for these. The pseudo code in Realtor.java is of this type.

```bash
cp ../0/Realtor.java Realtor1.java
dman Realtor1.java &
```
After the initial documentation, you should type in your program shell. A simplest program that one can write
would consist of the lines displayed in [Figure 1].

The statement that begins with import is used to identify classes that will be used when the application executes.
Whenever we wish to perform input from the keyboard we’ll use one or more of the java.util classes that are
available for this purpose. Later in the course we’ll discuss more of the details of exceptions and how you can
construct your own classes for this purpose. Every Java application must have at least one class, and we are
designating that class as Realtor1. During execution, class mp1 will be used to execute the method that is called
main. This is the area where execution of your application starts and where it ends.

Declaration of Variables

The declaration of variable memory in this language uses the following syntax.

```
data_type variable1, variable2, ..., variablen;
```

Optionally, any of the variables can be initialized to any simple expression involving constant values by extending
the above to the following.

```
data_type variable1 = expression1, variable2 = expression2, ..., variablen = expressionn;
```

The Java language has a large variety of data types. However, many of our applications can be handled by using
the data types String, boolean, int and double. In the Java language, variables only need to be declared
before we use them. However, we have so few in this program that we will not be overburdened by declaring
them all at once. We’ve already identified appropriate names for our variables and that they should be of types
double and String. Thus, the following declaration would be appropriate.

```
double price, cost, commission;
String seller;
```

The order of the variables here does not matter and since each is either an input variable or a calculated variable,
no initial values are required. The only other object that we’ll need to create is the Scanner object to support
input from the keyboard. Let’s name this object console and construct it with the following command.
Scanner console = new Scanner(System.in);

Our Scanner consists of an object named console that is initialized to the system variable System.in. System.in is to input as System.out is to output. The input that we’ll be performing will consist of searching for data from the keyboard that can be interpreted as an object of type double. This is done with the method console.nextDouble().

Follow the Pseudo Code

Step 1

// display an informational message

The general form that output to the monitor takes in Java is given by the following syntax.

System.out.println(expression1 + expression2 + ... + expressionn);

The object System.out is the standard output object that is usually connected to the monitor. Each of the expressions will be completely evaluated before any output will occur. Moreover, since the concatenation operator +, evaluates from left to right (i.e. is right associative), we will first evaluate and display expression1, then expression2 on through expressionn. For this point in our program, we only need to have one expression to print, a String constant. The println() or print() methods can generate a line skip for us using the char value of '\n'. If we use the formatted method printf() we must can use the char value '%n' at the end of our String. Consequently, the following would be an appropriate display message for our program.

System.out.printf("This program calculates the cost to sell a home%n" + "and the commission paid to an individual sales agent.%n%n" + "The user is asked for the last name of the seller and the%n" + "sales price.%n%n");

For this exercise, we’ll need to modify our input and output statements to take advantage of the JOptionPane methods. To begin with, we’ll need the following statement at the top of our file to make these methods available to us.

import javax.swing.JOptionPane;

Since we’re not going to format this output statement, we’ll need to perform our line skips with the '\n' character. Let’s store this displayed message into a String variable.

String display_message = "This program calculates the cost to sell a home\n" + "and the commission paid to an individual sales agent.\n" + "The user is asked for the last name of the seller and the\n" + "sales price.\n";

To print this message, we use the JOptionPane.showMessageDialog() method that expects 4 parameters:

<table>
<thead>
<tr>
<th>Component</th>
<th>Frame for the display, use null for default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>String to be displayed</td>
</tr>
<tr>
<td>Title</td>
<td>String title to be displayed</td>
</tr>
<tr>
<td>Message type</td>
<td>JOptionPane.type.MESSAGE, where type is one of ERROR, INFORMATION, WARNING, QUESTION or PLAIN</td>
</tr>
</tbody>
</table>

5
Consequently, we code the following statement to display the variable `display_message`. The results are shown in Figure 2.

```java
JOptionPane.showMessageDialog(null, display_message, "Lab 1 Description", JOptionPane.INFORMATION_MESSAGE);
```

**Steps 2-3**

// prompt for and input seller
// prompt for an input cost

To perform input, we’ll use the `JOptionPane.showInputDialog()` method that combines the prompt with the input operation. The method expects one parameter that is the `String` prompt and returns the `String` that was entered at the keyboard. So, to input the `seller` from the keyboard, we would use the following statement. The display of this prompt is shown in Figure 3.

```java
seller = JOptionPane.showInputDialog("Please enter the owner’s last name");
```

This works well enough for `String`, but for the `price` variable, we’ll want to perform arithmetic and will need the `String` converted to `double`. The manner in which to proceed is to read the data from the keyboard into a `String` object just as we did with `seller`. The display of this prompt is shown in Figure 4.

```java
String price_string = 
    JOptionPane.showInputDialog("Please enter the sales price of the home");
```
We follow this by using the `parseDouble()` method of the `Double` wrapper class to perform the conversion for us. If the data were type `int` rather than type `double`, we would use the `Integer.parseInt()` method for conversion.

```
price = Double.parseDouble(price_string);
```

**Steps 4-5**

// calculate cost  
// calculate commission

These two variables need to be calculated, and so assignment statements are required. The general format of the assignment statement in this language is the following.

```
variable = expression;
```

The expression on the right is completely evaluated before being copied over to the variable memory. For this exercise, we need to calculate 6% of the `price` for the `cost` and 1.5% of the `price` for the `commission`. These assignments are relatively straightforward and are coded as shown.

```
cost = 0.06 * price;
commission = 0.015 * price;
```

**Checkpoint**

Although we’re not done, at this point, we can complete a version of this program rather easily and it might be a good idea to compile and execute this much before we build more. The idea here is to try and discover our errors before we generate too much code. The trick to debugging programs for logic errors is to discover the line of code that contains the error. If you can’t find the line, you’ll never find the error. So, by searching for these errors over a smaller set of code, you are increasing the likelihood that you can find any errors.

Since we have the values calculated that we wish to display, we can simply repeat the output statements that were used in the previous exercise to display something to help us feel confidence in the code that we have thus far. Then, it simply becomes a matter of replacing those output statements with the ones being asked for in this assignment.

**Steps 6-7**

// display input values  
// display calculated values

Code the following statements in the same manner that you did in the previous exercise.
Save your program to disk by choosing the Save command from the Files menu of your emacs session. Then, compile your program by choosing the Compile... command from the Tools menu and change the make -k that is displayed to javac Realtor1.java. Compiler errors can be parsed with the keystroke C-x ' and need to be repaired before your program can execute. When your program has compiled, click on your xterm window to access your command line prompt, and issue the command java Realtor1. When you are prompted for the seller and the price, use the same values that you did in the previous exercise. If you do not get identical results, you should closely examine how you have coded Step 2 through Step 5 before continuing.

At this point, we are ready to display our two calculated results with appropriate formatting. This is going to involve an output statement that includes not only String constants as well as the input values but also the double calculated values cost and commission. We can and probably should consider using format descriptors for numeric values as well. You were introduced to format specifiers using the System.out.printf() methods of the previous exercise. We'll provide a more general introduction here. The format specifiers for general, character and numeric types have the following syntax.

```java
%[argument_index$][flags][width][.precision]conversion
```

With the meaning of the syntax shown below. Our output statements have been displaying the data as real numbers with 2 digits precision shown and accepting all other defaults.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument_index</td>
<td>optional int indicating the position of the argument in the list Defaults to 1</td>
</tr>
<tr>
<td>flags</td>
<td>optional format characters</td>
</tr>
<tr>
<td>width</td>
<td>optional int indicating minimal number of characters written</td>
</tr>
<tr>
<td>precision</td>
<td>optional int indicating maximal number of digits written</td>
</tr>
<tr>
<td>conversion</td>
<td>argument interpreted as should below</td>
</tr>
<tr>
<td>conversion value</td>
<td>interpretation</td>
</tr>
<tr>
<td>'s', 'S'</td>
<td>obtained by invoking toString()</td>
</tr>
<tr>
<td>'c', 'C'</td>
<td>a Unicode character</td>
</tr>
<tr>
<td>'d'</td>
<td>a decimal integer</td>
</tr>
<tr>
<td>'e', 'E'</td>
<td>a decimal number in scientific notation</td>
</tr>
<tr>
<td>'f'</td>
<td>a formatted decimal real number</td>
</tr>
<tr>
<td>'%'</td>
<td>a literal '%'</td>
</tr>
<tr>
<td>'n'</td>
<td>the platform-specific line separator</td>
</tr>
</tbody>
</table>

A manner in which to prepare formatted output statements to be displayed in a JOptionPane.showMessageDialog() method would be to store the formatted messages into String objects using the String.format() methods and then concatenate the results into one invocation of the JOptionPane.showMessageDialog() method. For example, we could declare and initialize three String objects as follows.

```java
String out1 = String.format("%nThe " + seller + ", home sold for $%.2f\n", price),
out2 = String.format("The cost to sell the home was $%.2f\n", cost),
out3 = String.format("The selling or listing agent earned $%.2f\n", commission);
```

Note how the String.format() method uses the '%n' char to generate a line skip in the same manner that the JOptionPane.showMessageDialog() method does. All that remains now is to replace the System.out.printf() statements with the following. The resulting output is displayed in Figure 5.
Testing for errors

You need to save your program, compile and execute it as you did up above at the Checkpoint. Use your
calculator to verify the correctness of your program. If you have any errors, they probably are limited to the
code that you wrote to replace the original code for Step 6 or Step 7 since you had the code running correctly
before you replaced those lines. There is a correctly functioning version of Realtor1.java at this link and
correct sample output to Realtor11.java at this link.

Printing and submitting

Once you are satisfied with the correctness of your program, print it as you did with the previous exercise by
using the following command that assumes that you are working in the room 1234 at Oakton. Retrieve your
copy from the printer.

    printer 1234 Realtor1.java

Finally, submit your program with the following command that assumes that you are registered in section abc
of CSC 156.

    submit csc156abc 1

If you pursue the extra credit option, store your solution in a file named Realtor11.java and print and submit
it with the following commands.

    printer 1234 Realtor11.java
    submit csc156abc 11

Maintenance

As software vendors have struggled to keep up with new releases of their products that address ever chang-
ing hardware improvements, they’ve discovered that using object-oriented programming methods have helped
shorten the development time. This is because these methods increase the reusability of code. Any code that is
rewritten is more likely to have errors introduced to it that delay production releases and increase programming costs. Consequently, every software environment who’s authors are interested in periodic upgrades has been loaded with object-oriented methodologies. Java is a true object-oriented language and we’ll want to attempt to construct maintainable programs whenever this is possible. This includes solid, clear software design that is well documented with variable names that reflect the values that they represent. Due to the length of time that we have to devote to this course, we will not be able to fully examine the maintenance issue, however we do want to incorporate attractive maintenance features into our code whenever we can.