Programming Exercise 7:
Static Methods

Purpose: Introduction to writing methods and code re-use.

Background readings from textbook: Liang, sections 6.1–6.5.

Due date for section 001: Monday, February 29 by 10 am
Due date for section 002: Wednesday, March 2 by 10 am

Overview

As we learn more and more about programming and want our programs to do more, our programs get larger. At some point, it becomes too complicated to design, write, debug and understand a program if all of the code is located in one method. In order to support modularity, the idea that our program consists of roughly independent modules, each a specialist in its own task, we divide our programs up. In Java, these modules are implemented as *methods*. Modularity has several advantages:

- The programmer can work on solving the overall problem first, and work on the details later.
- Coding becomes easier and faster if you can re-use already coded modules. Small modules that do one thing well are more likely to be useful in other parts of this and other programs.
- Finding an error in a small module is much easier than looking for it in a large unit.
- Reading and understanding someone's program written with small modules is easier.

You have already written one method in each of your programs, main. But you have also used other people's methods. Whenever you have a message passed to an object, whether it is System.out.println, or Math.sqrt or your Scanner doing input such as in.next, you are calling a method to handle the given task. In these cases, the methods are called println, sqrt, and next. Now, you will gain experience in writing your own methods.

A method definition consists of a *header* and a *body*. The header contains all the information that other code needs to use the method and the body contains the details of performing the task. In order to execute a method, we invoke it using a method call. The method call merely lists the method by name. When your program is executing and it reaches a method call, control is transferred from that part of your program to the method. The method executes and when it terminates, control returns to the next instruction after your method call.

As a method is somewhat independent of the rest of your code, you need some mechanism to tell that method what you want it specifically to do. For instance, when you call Math.sqrt, you have to tell sqrt what number you want the square root of, or when you call println, you have to tell System.out what to output. You supply such information as *parameters*. Parameters are the values that are sent to the method. We refer to the mechanism by which parameters are sent as parameter passing. We pass parameters in Java by placing the values in parentheses after the method's name. For instance, you might have Math.sqrt(5) or Math.sqrt(x) to inform the sqrt method to process the value 5 or the value stored in the variable x.

In this lab, you will explore writing methods in your program and then calling them. There is a peculiarity in Java in that if your class has a main method, that main must be a type of method called a static method and all other methods of that class must also be static. The term static in essence means "shared among all instances of this class". But for us, at least in this lab, we will just put the word static in the header and not worry about it.

Part 1: Java Examples

1. An example of a static method that returns no value:

```
public static void printChar( String message, int x )
// Prints every xth character of message starting with the very
// first character
// PARAMETERS
//     message: String of which we will print parts of
//     x: the number of characters to skip between loop iterations
{
       for(int i = 0; i < message.length(); i=i+x)
       {
              System.out.print(message.charAt(i));
       }
}
```

The method's header contains the visibility of the method, public meaning available, the modifier static, the return value type, the method's name, and its parameter list.

Comment header that describes the method's purpose and how to communicate with it.

2. An example of a static method that returns a value:

```
public static String reverse( String message )
// Given a String, create and return the message in
// reverse order
// PARAMETERS
//     message: the String that we want to reverse
// RETURNS
//     a String with the characters of message reverse
{
       String temp="";
       for(int i=message.length()-1;i>=0;i--)
              temp=temp+message.charAt(i);
       return temp;
}
```

The method's body contains the code encapsulated in the method. The return statement sends the results back to the caller.

Notice the use of `temp="";` A String with no value cannot be concatenated on to. So we start temp with the value "" meaning it has a value but so far stores no characters. The for loop iterates from the last character to the first in message, adding it to temp. If message="abc" then the loop starts at character 2 (the 'c') and adds it to temp, so temp is "c", then it adds 'b' and then 'a' giving us temp="cba", which is then returned.

3. Method to compute and return the factorial of an int number.

```
public static int factorial( int number )
// Computes the factorial of the given number.
// PARAMETERS number: the number to compute the factorial of
// RETURNS the value of the factorial
{
       int result = 1;
       for( int n = 2; n <= number; n++ )
       {
              result *= n;
       }
       return result;
}
```

Notes:

- The parameters are enclosed with parentheses and they are the means by which calling code can give the method information to use in accomplishing its task. A method is allowed to have no parameters, but the method name must always be followed by parentheses whether there are parameters or not.
- When a method is of type static, we can invoke it directly from the class rather than a specific object. Recall for Scanner, we created an instance such as in, and then passed it messages like in.next() and in.nextInt() whereas for println and sqrt, we could pass these directly to the class itself as in System.out.println and Math.sqrt. This is because println and sqrt are static methods and next and nextInt are not.
- The term public is a *visibility modifier* which dictates who can call this method. A public method can be called by any other method. We explore visibility later in the semester.
- A return value of **void** means no value is returned from the method and that a return statement is not needed. A return type of String or int means that the method must have at least one return statement. With if/if-else/nested if-else statements, it is possible that a method could have multiple return statements.
- Java method names should follow the same naming conventions as variable names.
- To call a method, list the name of the method, and in parentheses, place values or variables to be passed as parameters. The method call is the statement that calls the method. The names of parameters in the method call do not have to match the names of parameters in the method header. What has to match is the type, number and order.
- Methods are defined inside classes; you cannot define a method inside another method's definition.

Part 2: Common Pitfalls

1. Assuming that the factorial method is defined as in the above example, then the following call will produce an error:

```
System.out.println( factorial( 2.4 ) );
```

Syntax error!

The parameter passed in the call to the method must be of the same number and type as declared in the method's head. In this case, method factorial must be passed an **int** *value, not 2.4.*

2.
```
public static int factorial( int number )
{
       int result = 1;
       for( int n = 2; n <= number; n++ )
       {
              result *= n;
       }
}
```

Syntax error!

If the return type listed in the head is anything but void *then the method must have a* return *statement.*

3.
```
public int someFunction( int number )
{
       ...
}
```

Possible syntax error!

If this method is in a program with main, the method must be static!

Part 3: Problem

The GraffArt store has asked you to implement a computer program to determine the number of cans of paint, along with the cost, to paint a room of a given size with a given color. To simplify the design of the program and to learn about modularity, write this program using several methods.

Method 1: This method will get input from the user: one dimension of the room. Assume the room has 4 walls and a ceiling to paint and is rectangular. Two walls will have size *height*length* and two will have *height*width*. The ceiling's size is *length*width*. The floor will not be painted. This method is passed two parameters, a **String** parameter indicating which dimension is being requested (height, length, or width), and a **Scanner**. Both the String and the Scanner will be declared in main. The String will be used in a prompt to inform the user which dimension to enter. The Scanner will be used an input statement. This method will then get the input from the user as an int, stored in a temporary variable like result from method 3 of part 1 of this lab. This value will be returned. You will call this method 3 times, once for height, once for length and once for width in order to get the 3 dimensions of the room.

```
public static int getSize( String caption, Scanner input )
{
        System.out.print("Enter the " + caption + " of the room:  ");
       int value = input.nextInt();
       return value;
}
```

Method 2: Write a method to get the color of the paint to used. It will be similar to getSize except it will return a string of the color's name and will receive only the Scanner as a parameter.

Method 3: Write a third method to compute and return the number of paint cans needed to paint the given room. Assume one can of paint can cover 365 square feet of wall. Give this method a descriptive name. The method will receive 3 int parameters (the 3 dimensions as input) and will return the number of cans of paint as an int value. Note that if you do squareFootage/365, this will truncate the result. We actually need to round up to the next integer. For instance, 3.2 cans of paint would be converted to 3 but we really want 4. We must use Math.ceil to do this. Unfortunately, while Math.ceil rounds up, it also returns a double (for instance, 3.2 becomes 4.0). We therefore need to cast the result to an int. To compute the square footage of the room, determine the area for the four walls (remember two of the walls will be height*length and two will be height*width) and the ceiling. These five areas are summed up to give you the total square footage that you will then divide by 365.

Method 4: Write a method that computes and returns the total cost. The cost will be a double. This method will receive as parameters the number of cans and the color of the room. Use if statements or a nested if-else statement to determine the cost of paint given the color and the following table.

Color	Green	Orange	Mauve	Eggshell	White	Other
Cost per can	$3.68	$4.25	$3.69	$4.25	$3.25	$6.00

The cost may be reduced if any of the following discounts apply. Add this logic to the method.

- If the number of cans of paint purchased is more than 10, the customer receives 10% off.
- If the number of cans of paint purchased is more than 5 but 10 or less and they are using either Eggshell or White, the customer receives a 6% discount.

- If the number of cans of paint purchased is more than 5 but 10 or less and they are using any other color, the customer receives a 4% discount.

Method 5: Write an output method which will provide the final output to the user. It will state the dimensions of the room (height, width, length), the color of the paint, the number of cans of paint, and the total cost. Be sure to *format* the values appropriately and use correct grammar.

Method 6: Of course you need a **main** method. It will act as coordinator, leaving all the details to the other methods you have written. Your main method might look something like this:

```
public static void main(String[] args)
{
      Scanner input=new Scanner(System.in);
      DecimalFormat df=new DecimalFormat("…"); // add format
      String dimension, color;
      int height, width, length, numCans;
      double cost;
      // call getSize 3 times for "height", "width", "length"
      // call getColor to get the Color of the room
      // call computeNumberOfCans to determine number of cans
      // call computeCost to compute the cost of the paint cans
      // call your output function – pass it the dimensions, color,
      //       number of cans of paint, cost and df variables
}
```

Part 4: Writing and Testing Your Program

Unfortunately in Java, you can't test out your program one method at a time. You have to write the main method and if your main method calls other methods, then those methods have to be written as well. To write your program, first make sure you understand every method as described in section 3. Next, write your main method but comment out every method. Next, write one method, such as getSize. Uncomment the call(s) to the method you wrote, save and compile your program to see if it compiles. If not, your errors will be in main and/or the other method. Once done, write the next method and uncomment the call(s) to it. Continue to do it this way until your program is completed. Then you can test it out. For instance, if you have the method call `color=getColor(input);`, to comment it out, change it to be `// color=getColor(input);`. Run your program on the following three inputs to test out the logic. If you do not get the answers given, try to fix your logical errors.

Height: 10	Length: 14	Width: 12	Color: Green	Num Cans: 2	Cost: $7.36
Height: 12	Length: 18	Width: 21	Color: Mauve	Num Cans: 4	cost: $14.76
Height: 16	Length: 28	Width: 26	Color: Eggshell	Num Cans: 7	cost: $27.96

Once your program is correct, run your program on the data below. Collect the output in a text file and print out your source code and outputs and hand them in or email them to your instructor.

Height	Length	Width	Color	Height	Length	Width	Color
18	20	25	Green	10	19	33	Mauve
12	16	14	Orange	9	66	56	Blue
12	22	20	White	16	28	22	White
30	85	40	Eggshell	24	35	20	Black