Chapter 4

Procedures and Functions

What is in This Chapter ?

In this set of notes, we will discuss how to make our code simpler and easier to understand through abstraction. That is, we will learn how to think of writing general code and then fill in the details later. We will do this by learning to write ***functions*** and ***procedures***, which are also called ***methods***. Methods are the set of instructions that perform some operations with the data or objects that you need to work with. Methods allow you to keep your code organized into logical modules, they can allow you to repeat code without having to re-write it and they also allow you to make your overall program simpler to understand.

4.1 Procedures

When solving real-life-problems, it is often necessary to break the problem down into manageable steps. For example, consider doing a jigsaw puzzle. If the jigsaw puzzle has many pieces (e.g., 5000), it can be an overwhelming feat to put it together. Some people feel sick at the thought of having to put the puzzle together, yet others find enjoyment in such a challenge. Those who decide to take on the challenge will usually always break the problem down into simpler, more easily-managed ***sub-problems***.

For example, consider how you would solve a jigsaw puzzle with the image shown here ? --->

Perhaps this would be your solution:

Algorithm: SolvePuppies

1. Pour out all pieces onto table
2. Flip all pieces over to show picture side
3. Separate edge pieces from inside pieces
4. Solve edge pieces
5. Solve sign portion
6. Solve puppies portion
7. Solve grass portion
8. Solve gate portion
9. Solve hay portion
10. Solve barn portion

Notice that we broke the problem down into simple easy-to-understand steps. Of course, the order in which we solve the particular parts of the image is unimportant, but do you see how breaking a problem down into simpler, smaller **procedures** can make it easy ?

In the above solution, many details are hidden. For example, there are no details about "how" to solve the "puppies portion" in step 6. The details are hidden (or left until later) because we are just trying to get an overview of the problem steps here. Eventually, we will have to work out all of the details. For example, consider the details that go into step 2.

It would involve a **WHILE** loop and an **IF** statement:

```
WHILE (there are still some upside down pieces) {
    piece = select (i.e., look at) a piece
    IF (piece is upside down) THEN
          flip piece over
}
```

And what about step 7, for example ?

```
FOR EACH (piece) {
    piece = select (i.e., look at) a piece
    IF (piece is has grass on it) THEN
          add piece to a incompleteGrassPiecesPile
}

completedGrassPortion = select a piece from the incompleteGrassPiecesPile
WHILE (incompleteGrassPiecesPile still has pieces) {
    piece = select a piece from the incompleteGrassPiecesPile
    IF (piece fits with completedGrassPortion) THEN
          join piece to completedGrassPortion
    OTEHRWISE
          put piece back into incompleteGrassPiecesPile
}
```

Do you see how "ugly", messy and complicated things can get ? I'm sure you agree that by keeping the main algorithm more *abstract*, we have a simpler overview of how to solve the problem without trying to struggle through the details.

This strategy of breaking down the problem into smaller pieces is often called ***divide and conquer*** and it represents the fundamental principle for problem solving. The idea of coming up with a clearer algorithm without too much details is known as ***abstraction***.

> *Abstraction is the process of reducing or factoring out details that are not necessary in order to describe an algorithm.*

When programming, abstraction is important because it allows us to focus on a few concepts at a time. This allows us to get the “big picture” first in regards to the problem solution. We can then “fill in” the specific details at a later point in time.

The ***main algorithm*** that is used to solve the problem will often have only a few steps. Each of the individual steps may be called ***sub-algorithms***. In computer science, these more detailed sub-algorithms are sometimes called ***modules***, ***functions*** or ***procedures***.

Example:

Let us look now at how to make a simple ***procedure*** in JAVA by looking at one of our previous examples ... our **LuigisPizzaProgram**:

```
import java.util.Scanner;

public class LuigisPizzaProgram {
	public static void main(String[] args) {
		System.out.println("Luigi's Pizza                                    ");
		System.out.println("-----------------------------------------");
		System.out.println("                      S(SML)  M(MED)  L(LRG)");
		System.out.println("1. Cheese              5.00    7.50   10.00 ");
		System.out.println("2. Pepperoni           5.75    8.63   11.50 ");
		System.out.println("3. Combination         6.50    9.75   13.00 ");
		System.out.println("4. Vegetarian          7.25   10.88   14.50 ");
		System.out.println("5. Meat Lovers         8.00   12.00   16.00 ");

		Scanner keyboard = new Scanner(System.in);

		System.out.println("What kind of pizza do you want (1-5) ?");
		int kind = keyboard.nextInt();

		System.out.println("What size of pizza do you want (S, M, L) ?");
		char size = keyboard.next().charAt(0);

		float cost = 4.25f + (kind * 0.75f);
		if (size == 'M')
			cost *= 1.5f;
		else if (size == 'L')
			cost *= 2;

		System.out.println("The cost of the pizza is: " +
				String.format("$%,1.2f", cost));
		System.out.println("The price with tax is: " +
				String.format("$%,1.2f", cost * 1.13));
	}
}
```

How can we work backwards now .. to re-write the program using high-level abstraction ? That is, how can we hide the details ? Well, the program actually performs just 4 basic steps:

```
(1) Display the menu
(2) Get the user's choices
(3) Calculate the cost
(4) Display the results
```

So, we can re-write this program using ***procedures*** as follows:

```
import java.util.Scanner;

public class LuigisPizzaProceduresProgram {
	public static void main(String[] args) {
		displayMenu();
		getUserInput();
		computeCost();
		displayResults();
	}
}
```

Notice how we are writing these steps. These 4 steps are actually ***procedure calls*** (also known as ***method calls***) That is, they each *call* (or *invoke* ... or *run*) a *procedure* ... which

contains steps to solve the particular procedure with that name. The parentheses are necessary because that distinguishes these procedure names (i.e., ***method names***) from simple variable names.

The code above, however, will not compile. We actually need to go and write the code for each of these 4 methods. Here is how we do this:

```
import java.util.Scanner;

public class LuigisPizzaProceduresProgram {

	public static void displayMenu() {
	}

	public static void getUserInput() {
	}

	public static void computeCost() {
	}

	public static void displayResults() {
	}

	public static void main(String[] args) {
		displayMenu();
		getUserInput();
		computeCost();
		displayResults();
	}
}
```

You may notice that the individual methods look a lot like the **public static void main(...)** code that we have been using already. That is because that code is called the ***main procedure*** (or ***main method***)... it is the procedure that your program begins at. Here is the standard format for a procedure:

```
public static void procedureName () {
	// Write your procedure's code here
}
```

The **public** keyword indicates that anyone can call this procedure from any code (we will talk more about this at the end of the course of in COMP1406).

The **static** keyword indicates that this is a kind of "global" procedure which is also known as a ***class method***. For now, do not worry about this. Later we will talk about another kind of method called an ***instance method***.

The **void** keyword indicates that there is no value to be returned from the method. That is ... the code inside the procedure does something, but does not return an answer. For example, if you ask someone what their name is, you expect information back as an answer. However, if you ask someone to leave the room ... there is no information expected back ... just obedience to the request to leave the room.

The **procedureName** is any name that you decide upon ... but you should choose something descriptive that indicates the purpose of the procedure.

The brace characters (i.e., **{ }**) indicate the code's body:

> *The **body** of a function or procedure is the code that is evaluated each time that the function or procedure is called.*

Now, the code that we wrote will ***compile***, but it does not ***do*** anything. That is because we have to fill in all the details.

Here is the completed **displayMenu()** method:

```
public static void displayMenu() {
     System.out.println("Luigi's Pizza                                ");
     System.out.println("-------------------------------------------");
     System.out.println("                      S(SML)  M(MED)  L(LRG)");
     System.out.println("1. Cheese              5.00    7.50   10.00 ");
     System.out.println("2. Pepperoni           5.75    8.63   11.50 ");
     System.out.println("3. Combination         6.50    9.75   13.00 ");
     System.out.println("4. Vegetarian          7.25   10.88   14.50 ");
     System.out.println("5. Meat Lovers         8.00   12.00   16.00 ");
}
```

That is all that we should put in there. It is all the code responsible for displaying the initial menu. So, when the main program runs, it first calls this **displayMenu()** method ... which causes the JAVA interpreter to go inside the method and run the code that is in there. After the last line of the method is evaluated, the procedure is considered completed, and the JAVA interpreter goes back to the main method (i.e., from where the procedure was originally called) and continue with the program there ... which means that the following procedure (called **getUserInput()** is called next). What does that code look like ?

```
public static void getUserInput() {
     Scanner keyboard = new Scanner(System.in);

     System.out.println("What kind of pizza do you want (1-5) ?");
     int kind = keyboard.nextInt();

     System.out.println("What size of pizza do you want (S, M, L) ?");
     char size = keyboard.next().charAt(0);
}
```

This code gets the **kind** and **size** values from the user. The code compiles and runs ... but we will see a problem when we go to write the next method:

```
public static void computeCost() {
     float cost = 4.25f + (kind * 0.75f);
     if (size == 'M')
          cost *= 1.5f;
     else if (size == 'L')
          cost *= 2;
}
```

This code performs all the calculations and produces the cost of the pizza. There is a problem though. The code will not compile. Here is the error that you will see:

```
error: cannot find symbol
```

The reason is that the **kind** and **size** variables are actually defined in the **getUserInput()** method, but we are trying to use them outside of that method (i.e., in this **computeCost()** method).

Rule: *Within a method, you cannot access variables that are defined in other methods.*

In fact, in general, the brace characters (i.e., **{ }**) represent what is known as a ***code block***. Any variables that are defined within a code block, are not accessible outside of that code block. So, we need a way of sharing the **kind** and **size** variables between the two methods so that we can given these variables values from the **getUserInput()** method, and then use these values in the **computeCost()** method.

To do this, we need to define the variables outside of the methods. We can define the **kind** and **size** variables as ***static variables*** at the top of our program. This makes them ***global*** to the program so that every procedure/method that we write, will be able to access and modify them. We will need to do this for the **cost** variable as well. Here is the completed program:

```
import java.util.Scanner;

public class LuigisPizzaProceduresProgram {

	public static int	kind;
	public static char	size;
	public static float	cost;

	public static void displayMenu() {
		System.out.println("Luigi's Pizza                                ");
		System.out.println("-------------------------------------------");
		System.out.println("                    S(SML)  M(MED)  L(LRG)");
		System.out.println("1. Cheese            5.00    7.50   10.00 ");
		System.out.println("2. Pepperoni         5.75    8.63   11.50 ");
		System.out.println("3. Combination       6.50    9.75   13.00 ");
		System.out.println("4. Vegetarian        7.25   10.88   14.50 ");
		System.out.println("5. Meat Lovers       8.00   12.00   16.00 ");
	}

	public static void getUserInput() {
		Scanner keyboard = new Scanner(System.in);

		System.out.println("What kind of pizza do you want (1-5) ?");
		kind = keyboard.nextInt();

		System.out.println("What size of pizza do you want (S, M, L) ?");
		size = keyboard.next().charAt(0);
	}
```

```
	public static void computeCost() {
		cost = 4.25f + (kind * 0.75f);
		if (size == 'M')
			cost *= 1.5f;
		else if (size == 'L')
			cost *= 2;
	}

	public static void displayResults() {
		System.out.println("The cost of the pizza is: " +
					String.format("$%,1.2f", cost));
		System.out.println("The price with tax is: " +
					String.format("$%,1.2f", cost * 1.13));
	}

	public static void main(String[] args) {
		displayMenu();
		getUserInput();
		computeCost();
		displayResults();
	}
}
```

All that we did here, was to extract the variable type definitions and put them at the top of the program. They are not inside any procedures now. The braces that surround them are the ones that begin and end the whole program ... so we can use them anywhere in our program.

Notice that we are no longer specifying the types for the variables when we use the variables within the methods:

```
	kind = keyboard.nextInt();
```

instead of:

```
	int kind = keyboard.nextInt();
```

That is because the variables are now already defined earlier ... so we can just use them now.

So there you go. We have written a program with 4 procedures. However, you may not be overly excited cause the program looks bigger than it was before! That is because there is some overhead (i.e., extra lines of JAVA code) when using procedures/methods. However, keep in mind that the goal of this exercise was to understand how to simplify an algorithm so that it is easier to understand. You must admit, when looking at the main method, that the overall program is quite simple in structure. Still, with such a small program like this, the advantage of abstracting like this is not readily apparent. However, when our programs become larger, you will see how important it is to create methods so that your code is easily understood and maintainable.

We likely would not make a procedure for every step of a program because too many procedures make the code complicated-looking again and it also slows down your program because there is a little bit of timing overhead each time that the JAVA interpreter calls a procedure.

Generally, you should only make a procedure if you have a good reason for doing so. Remember, a procedure's purpose is to simplify your code and make things more clear to the person trying to read your code and modify it. Here are three main reasons that you might want to create a procedure:

- Your **code is too long**, breaking into procedures makes it more manageable and easier to deal with.
- A chunk of code is **too complicated and hard to follow**. Breaking it up may make the code more readable and understandable.
- A portion of code may **need to be repeated** many times. Making a procedure will prevent code from being duplicated in many places.

You may have noticed that the **main()** method that we write for each program contains information within the parentheses (i.e., **`String[] args`**). This information is called the procedure's incoming ***parameters***. As it turns out, we are allowed to pass along some information each time that we call a procedure, provided that the procedure is expecting this information.

We do this in real life. For example, if we want to build a shed, we are going to need material s, such as wood, plastic, metal pieces, nuts, bolts, screws etc... These are absolutely necessary to perform the procedure of building the shed. So, you can think of method parameters as "***objects that you will need to solve the problem***".

Another way to think of it is when we need to fill out a paper form for someone. The form requires particular information, which we will need from that person, otherwise we cannot complete the form. So, you can also think of method parameters as "***information that you will need to solve the problem***".

In JAVA, here is the format for passing in a parameter to a function:

```
public static void procedureName (type name) {
    // Write your procedure's code here
}
```

The parameter must be declared just like a variable (i.e., with a type followed by a name). While inside the procedure, the parameter is available for us to use … just as we would use any other variable.

Example:

Write a program that displays a table of Celsius values in one column and it's corresponding Fahrenheit values in a second column beside it (partially shown here on the right). The program should show values from the -40 to +40 range as shown here on the right, by making use of the formula for converting a Celsius temperature to Fahrenheit: F = (C x 9/5) + 32.

Your program should have a procedure called **CelsiusToFahrenheit()** that takes a single incoming parameter, which is an **int** indicating a Celsius temperature. The procedure should then display the Fahrenheit temperature for that Celsius value.

To begin this, let us write a **main()** method that displays the table's headings and a loop that displays all the Celsius values:

```
  C     F
-40   -40
-39   -38
-38   -36
 .     .
 .     .
 .     .
 -3    27
 -2    29
 -1    31
  0    32
  1    33
  2    35
  3    37
 .     .
 .     .
 .     .
 38   100
 39   102
 40   104
```

```
public class TemperatureTableProgram {

      public static void main(String[] args) {
            System.out.println("  C     F");

            for (int c=-40; c<=40; c++) {
                  System.out.print(String.format("%d", c));
                  System.out.println();
            }
      }
}
```

Now we need to write the procedure that takes the Celsius value and then computes the Fahrenheit value and prints it:

```
public static void CelsiusToFahrenheit(int celsius) {
      int fahrenheit = (celsius * 9/5) + 32;

      System.out.println(fahrenheit);
}
```

Notice that the parameter is declared just like a variable ... with a type, followed by a name of our choice. Within the code, the parameter is used just like a variable.

Now, we put it all together by calling the **CelsiusToFahrenheit()** method from the **for** loop:

```
public class TemperatureTableProgram {

	public static void CelsiusToFahrenheit(int celsius) {
		int fahrenheit = (celsius * 9/5) + 32;
		System.out.println(String.format("%6d", fahrenheit));
	}

	public static void main(String[] args) {
		System.out.println("  C     F");

		for (int c=-40; c<=40; c++) {
			System.out.print(String.format("%4d", c));
			CelsiusToFahrenheit(c);
		}
	}
}
```

Notice that we pass the loop variable **c** as a parameter to the **CelsiusToFahrenheit()** method. So, each time the method is called, it gets the current value of **c** from the loop. Inside the method, the value is "nick named" **celsius** because that is the name that we chose for the parameter. So, when we use **celsius** in the method, we are actually referring to the value of **c** that is in the **for** loop. This is called ***pass-by-value*** because the *value* of the variable **c** is passed into the method. Therefore, if we try to change the value of **celsius** from within the method, it will NOT affect the value of **c** outside the method:

```
public static void CelsiusToFahrenheit(int celsius) {
	int fahrenheit = (celsius * 9/5) + 32;

	System.out.println(fahrenheit);
	celsius = 25; // This is useless, and will NOT affect the value of c
}
```

We can actually pass in many parameters to a method. In this case, each parameter must have a type and a name ... and each type/name pair must be separated by a comma:

```
public static void procedureName (type1 name1, type2 name2, type3 name3) {
    // Write your procedure's code here
}
```

Example:

Recall our **TripExpenseProgram** that we wrote in chapter 2. See if you can write a method called **CalculateDifference()** that takes the 4 trip expense costs as parameters and then displays how much money Steve owes Bob or vice versa ...

```
public static void CalculateDifference(float f, float h, float g, float e) {

    float each = (f + h + g + e) / 2;
    float difference = each  - (g + e);

    if (difference > 0)
        System.out.print("Steve owes Bob ");
    else
        System.out.print("Bob owes Steve ");

    System.out.println(String.format("$%,1.2f", difference));
}
```

Example:

It is "Special Ladies Night" at the local Tim Hortons. Every female who is 12 years of age or younger gets a 50% discount. Also, every retired woman gets 50% off. The rest of us get no discount :(. Write a method called **ComputeDiscount()** that takes three parameters ... a person's **age**, a person's **gender** (i.e., 'M' or 'F') and a boolean indicating whether or not the person is **retired**. The method should then display on the screen an appropriate discount amount.


```
public static void ComputeDiscount(int age, char gender, boolean retired) {
     int   discount = 0;

     if ((gender == 'F') && (age <= 12 || retired))
          discount = 50;

     System.out.println(discount);
}
```

4.2 Functions

In our examples so far, we have been computing and printing answers out from within a procedure. This is actually not a very good way to write programs. Input and output to a program depends on the kind of user interface ... so it is good to isolate all code having to do with getting user input and display answers. What is often done instead, is that when we compute an answer in a method, we then *pass back* or ***return*** that answer to the main method, or to whoever called our method. Once we return a value, the method is no longer called a *procedure* but is now called a ***function***.

Functions are created the same way that procedures are, but with one exception. Instead of **void**, a function must declare the ***type*** of the value returned as follows:

```
public static typeR functionName (type1 name1, type2 name2, type3 name3) {

    // Write your function's code here

    return _______;
}
```

Here, t_{ypeR} is the ***return type*** of the function:

*A **return type** is the type of the value that is returned from a function.*

So a function is exactly the same as a procedure, except that it must return a value which has the same type as specified as its return type.

Notice as well that somewhere in the function's body, you MUST have a ***return*** statement that returns a value which has the same type as t_{ype}. Your function may have many return statements since you may want to return different values under different circumstances.

Example:

Here is a revised **TemperatureTableProgram** where the procedure is modified to be a function that *returns* the Fahrenheit value instead of *printing* it:

```
public class TemperatureTable2Program {

    public static int CelsiusToFahrenheit(int celsius) {
        int fahrenheit = (celsius * 9/5) + 32;
        return fahrenheit;
    }

    public static void main(String[] args) {
        System.out.println("  C     F");

        for (int c=-40; c<=40; c++) {
            int f = CelsiusToFahrenheit(c);
            System.out.println(String.format("%4d%6d ", c, f));
        }
    }
}
```

Notice how the **System.out.println()** statement is now removed from the method. Also, notice that the **return type** of the method is specified as **int** instead of **void**. Then, at the end of the method, we returned the `fahrenheit` value. We could have reduced the code even further by noticing that we are only using the `fahrenheit` variable once ... so it is not needed. Here is the simplified method:

```
public static int CelsiusToFahrenheit(int celsius) {
    return (celsius * 9/5) + 32;
}
```

Lastly, notice that we merged all the printing into one line in the **main()** method.

Again, since we are only using the **f** variable once, we can eliminate the variable altogether by calling the **CelsiusToFahrenheit()** method directly within the **String.format()** function call:

```
public static void main(String[] args) {
     System.out.println("  C     F");
     for (int c=-40; c<=40; c++) {
          System.out.println(String.format("%4d%6d ", c, CelsiusToFahrenheit(c)));
     }
}
```

Example:

Let us modify our ComputeDiscount() method to return the discount, instead of printing it out:

```
public static void ComputeDiscount(int age, char gender, boolean retired) {
     int   discount = 0;

     if ((gender == 'F') && (age <= 12 || retired))
          discount = 50;

     System.out.println(discount);
}
```

The return type needs to change from void to int ... and then we can simplify everything:

```
public static int ComputeDiscount(int age, char gender, boolean retired) {
     if ((gender == 'F') && (age <= 12 || retired))
          return 50;
     return 0;
}
```

Example:

Consider a function called Minimum() that takes 3 integers as parameters and returns the minimum of the three numbers. How would you write this ?

We would need to begin by getting the method's ***structure*** (also known as method's ***signature***) correct. That is, we need to write out the first line with the proper return type and parameters:

```
public static int Minimum(int first, int second, int third) {

}
```

This code will not compile though. We would get an error stating:

error: missing return statement

That is because we specified that the code must return an **int** but we never return an **int**. We can think about getting the answer into a variable, perhaps called **minimum** ... and then returning it at the end of the method. This is a good start, because it allows our code to compile and return something which is the proper type, even though the answer is not yet correct.

```
public static int Minimum(int first, int second, int third) {
     int   minimum = 0;

     // Put interesting code here

     return minimum;
}
```

Now we need to start comparing numbers. The first number would be the answer only if it is less than the other two numbers. If it is not, then we know that the answer must be either the second or third numbers ... so we can just choose the lowest of those.
What do we do if one or more numbers are equal ? What if all the numbers are equal ? In those cases, when comparing two numbers that are equal, it does not matter which of those numbers we return as an answer since they are the same.

```
public static int Minimum(int first, int second, int third) {
     int   minimum = 0;

     if ((first < second) && (first < third))
           minimum = first;
     else {
           if (second < third)
                 minimum = second;
           else
                 minimum = third;
     }
     return minimum;
}
```

Hopefully, the above code makes sense to you. We can reduce this code even further by realizing that once we set the minimum value to something, we are never really doing anything with it except for returning it. So, we don't need the **minimum** variable. We can simply **return** the answer whenever we know it as follows:

```
public static int Minimum(int first, int second, int third) {
	if ((first < second) && (first < third))
		return first;
	else {
		if (second < third)
			return second;
		else
			return third;
	}
}
```

Notice that we have three **return** statements here, but that only one of them can be evaluated. You are allowed to have many **return** statements in a function ... provided that code does not directly follow a **return** statement. Whenever a **return** statement is reached ... the function stops and returns that value and the function does not continue evaluating more code.

We can actually simplify even more by realizing that once since the first **if** statement has a **return** in it, then it is implied that the code after the **if** body is an **else** case. That is, we don't need to specify **else**, because if the code did not return from the **if** statement's condition, the rest of the code must automatically be considered as part of an **else** since the method did not return.

So we can write this, which is shorter and does the same thing:

```
public static int Minimum(int first, int second, int third) {
	if ((first < second) && (first < third))
		return first;
	if (second < third)
		return second;
	return third;
}
```

It would be good to test this out by writing a test program:

```
public class MinimumProgram {
	public static int Minimum(int first, int second, int third) {
		if ((first < second) && (first < third))
			return first;
		if (second < third)
			return second;
		return third;
	}

	public static void main(String[] args) {
		System.out.println(Minimum(23, 45, 57));
		System.out.println(Minimum(45, 23, 57));
		System.out.println(Minimum(57, 45, 23));
	}
}
```

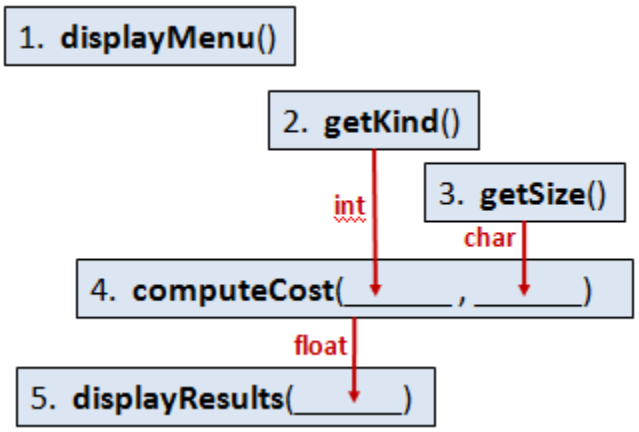
The program should print out the number 23 three times.

Example:

We can also consider modifying our Luigi's pizza program to use functions instead of procedures. Try doing it so that the number of variables used altogether is kept small (e.g., no global variables). Just pass each function's return value as parameters into other functions.

```
import java.util.Scanner;

public class LuigisPizzaFunctionsProgram {
	public static void displayMenu() {
		System.out.println("Luigi's Pizza                                    ");
		System.out.println("-----------------------------------------------");
		System.out.println("                      S(SML)  M(MED)  L(LRG)");
		System.out.println("1. Cheese              5.00    7.50   10.00 ");
		System.out.println("2. Pepperoni           5.75    8.63   11.50 ");
		System.out.println("3. Combination         6.50    9.75   13.00 ");
		System.out.println("4. Vegetarian          7.25   10.88   14.50 ");
		System.out.println("5. Meat Lovers         8.00   12.00   16.00 ");
	}

	public static int getKind(Scanner keyboard) {
		System.out.println("What kind of pizza do you want (1-5) ?");
		return keyboard.nextInt();
	}

	public static char getSize(Scanner keyboard) {
		System.out.println("What size of pizza do you want (S, M, L) ?");
		return keyboard.next().charAt(0);
	}

	public static float computeCost(int kind, char size) {
		float cost = 4.25f + (kind * 0.75f);

		if (size == 'M')
			return cost * 1.5f;
		else if (size == 'L')
			return cost * 2;
		return cost;
	}

	public static void displayResults(float cost) {
		System.out.println("The cost of the pizza is: " +
				String.format("$%,1.2f", cost));
		System.out.println("The price with tax is: " +
				String.format("$%,1.2f", cost * 1.13));
	}

	public static void main(String[] args) {
		Scanner keyboard = new Scanner(System.in);
		displayMenu();
		displayResults(computeCost(getKind(keyboard), getSize(keyboard)));
	}
}
```

Notice the last line of the **main()** method. It has 4 function calls. **displayResults()** takes as a parameter, the return value from the **computeCost()** method. But before that, the **getKind()** and **getSize()** methods each return their user inputs, which become parameters to the **computeCost()** method. Make sure that you understand this last line. It is important to understand that the order of the method calls is as follows:



The rule is that a method can only be called if its parameters are available. So, for example, even though the **displayResults()** was written before the **computeCost()** method, JAVA is unable to call that method until the value of the **computeCost()** method call is returned so that it can be used as the parameter to the **displayResults()** method.

Just a few last points about methods. Methods are uniquely identified by their **name** and their **parameter types**. So, in the same program, you cannot define two methods with the same name and parameter lists. Consider our **minimum()** method. Here are examples of other methods that are allowed and are not allowed to be defined within the same program:

```
public static int Minimum(int first, int second, int third) { ... }
```

`public static int Minimum(int a, int b, int c) { ... }` `// Not allowed since name and parameter types are all the same`	
`public static int Min(int first, int second, int third) { ... }` `// Allowed since name is different`	
`public static int Minimum(float first, float second, float third) { ... }` `// Allowed since parameter types are different`	
`public static int Minimum(int first, int second) { ... }` `// Allowed since number of parameters are different`	
`public static void Minimum(int first, int second, int third) { ... }` `// Not allowed since name is the same but return type is different`