Section 1 - Class Variables in Methods

5B.1.1 Static Class Variables

Thus far we left an item unresolved - how one transfers data from the client to the methods. So far, we have only one way: the ***functional return*** value. And we can only get a single value that way, so that is a limited modality.

A second method for exchanging data among different functions is through the use of ***class member data***, of which there are two types:

- ***static class variables***
- ***instance variables***.

Although a ***static class variable*** is distinct from an ***instance variable***, we will only look at ***static class variables*** for now and not concern ourselves with the differences until a future lecture.

We saw that ***local variables*** are defined inside ***methods***, at the top, below the ***function header***, but usually above the executable statements. These ***locals*** were not shared between distinct methods.

All ***class member data***, including ***static class variables,*** are defined outside, and usually above any of the ***methods***, but inside the ***class*** (of which we still have only the one). ***Static class variables*** are known to all methods in the class. That's the whole point.

Just as we have used the **static** keyword in front of all of our ***method definitions***, for now we will use it in front of all of our ***class member data***, as well. In fact, the word **static** tells the compiler that these variables, declared outside any methods, are ***static class variables***. If we omitted the **static** keyword they would then be called ***instance variables***, which, as I just said, we will not learn this lesson.

5B.1.2 Using Static Class Variables to Share Data

Now we think back to our mortgage calculator and ask what was one of the remaining weaknesses in this program? The answer is that one of the methods was too long and did too much. We were getting data from the user and also computing the monthly payment in the same method. This is bad not only because it gives one method too much responsibility, but more importantly, because we are mixing I/O with calculations. We can now fix this.

So we will want one method to get the input from the user, and a second method to do the calculations. We will break up **getInputAndComputeMonthlyPayment()** into two smaller methods:

- **getInput()**
- **computeMonthlyPayment()**

Even the names are more palatable.

Somehow we have to transfer the information that we receive from **getInput()** into the method **computeMonthlyPayment()**. We will use ***static class variables*** to do this. But which variables are needed by both method and which can we declare as local in each? After all, we do not want to declare variables as ***static class variables*** if they are only needed by one method.

Let me repeat that last statement since it is so important that points will come off if you don't obey it. If only one method uses a variable you do not want to declare it as a static class variable. You should instead, declare it to be a ***local variable*** (as we've been doing up to this point) to the one method that uses it.

So which variables of the mortgage calculator should be declared as ***static class data***?

These are the variables that are needed by both methods:

1. **dblPrincipal**
2. **dblRate**
3. **dblYears**

These are the variables that the user gives us (or will give us) in **getInput()** and they are needed to compute the monthly payment. So we declare them as ***static member data***:

```
public class Foothill
{

   // class variables shared by more than one method
   static double dblPrincipal, dblRate, dblYears;

   // main method
   public static void main (String[] args)
   {
      // content of main() omitted
   }
     // gives an overview to user
   public static void stateInstructions()
   {
      // content of method omitted
   }

   // gets input, stores in static class variables
   public static void getInput()
   {
      // content of method omitted
   }

   // etc ...
}
```

As you can see, we have declared these three variables inside the class and above all of the methods. We also used the **static** keyword. Now, any of the methods in the class can use these variables without re-declaring them (in fact re-declaring them inside a method would be a major error).

After you understand the concept of data sharing and the syntax for declaring static class variables, go on to the next section where we apply it to the mortgage calculator.

Section 2 - Scanner and Mortgage Calculator Version 3

5B.2.1 The Program Listing

This latest version of the mortgage calculator demonstrates the use of ***class variables***:

```
import java.util.Scanner;

public class Foothill
{

   // class variables shared by more than one method
   static double dblPrincipal, dblRate, dblYears;


   // main method
   public static void main (String[] args)
   {
      double answer;

      stateInstructions();
      getInput();
      answer = computeMonthlyPayment();
      System.out.println("\n\nMonthly Payment: "
            + answer);
      sayGoodbye();
   }

   // gives an overview to user
   public static void stateInstructions()
   {
      String instructions;

      instructions =
         "The following program will calculate the \n"
         + "monthly payment  required for a loan of D dollars \n"
         + "over a period of Y years at an annual \n"
         + "interest rate of R%.";
      System.out.println(instructions);
      System.out.println("---------------\n");
   }

   // gets input, stores in static class variables
   public static void getInput()
   {
      Scanner input = new Scanner(System.in);
      String prompt, strUserResponse;
```

```
        // get principal
        prompt = "\nEnter amount of the loan. (only use numbers, \n"
           + "please, no commas or characters like '$')\n"
           + "Your loan amount: ";
        System.out.print(prompt);
        strUserResponse = input.nextLine();
        dblPrincipal = Double.parseDouble(strUserResponse);

        // get interest
        prompt = "\nNow enter the interest rate (If the quoted rate is "
           + " 6.5%, \nfor example, enter 6.5 without the %.)\n"
           + "Your annual interest rate: ";
        System.out.print(prompt);
        strUserResponse = input.nextLine();
        dblRate = Double.parseDouble(strUserResponse);

        // get length of loan
        prompt = "\nEnter term of the loan in years: ";
        System.out.print(prompt);
        strUserResponse = input.nextLine();
        dblYears = Double.parseDouble(strUserResponse);

        input.close();
    }

    // computes and returns answer
    public static double computeMonthlyPayment()
    {
        // local variables needed only in this method
        double dblTemp, dblPmt, dblMonths, dblMoRt;

        // convert years to months
        dblMonths = dblYears * 12;

        // convert rate to decimal and months
        dblMoRt = dblRate / (100 * 12);

        // use formula to get result
        dblTemp = Math.pow(1 + dblMoRt, dblMonths);
        dblPmt = dblPrincipal * dblMoRt * dblTemp
           / ( dblTemp - 1 );

        // now that we have computed the payment, return it
        return dblPmt;
    }

    // sign off
    public static void sayGoodbye()
    {
        String signoff;
        signoff =
           "\nThanks for using the Foothill Mortgage Calculator. \n"
           + "We hope you'll come back and see us again, soon.";
        System.out.println(signoff);
    }
}
```

Notice that **getInput()** is using only two local objects, **prompt** and **strUserResponse**. Other than those, it is using the ***static class variables*** (of course, without declaring them).

I said that if you define a ***static class variable***, you are not going to define it again within a ***method***. Why? If you did, you would ***"hide"*** the ***static class variable*** from that method and create a new, ***local variable*** only known to the method. The same is true of the other type of member data, the ***instance variable*** that we will learn about later. So ... when you define ***member data***, whether it be a ***static class variable*** or an ***instance variable***, don't define ***local variables*** in functions that have the same name. If you do, you will be rendering the ***member data*** unreachable.

Sometimes we see methods that have no ***local variables*** defined at the top because they are going to use only ***static*** or ***instance*** variables of their class.

5B.2.2 When You Should *and Should Not* Use Class Member Data

Static class variables are obviously an efficient means to exchange information because they are instantly known to all methods of the class, but often this is not the way to exchange information. That's because of their great weakness: exposure.

Static class variables are seen by all ***methods*** of the class, even ones that have nothing to do with their processing. Any ***method*** could inadvertently modify them, and so these static members are vulnerable at all times. If a ***static member*** gets a bad value, we have to check all ***methods*** to see who is to blame because we can't narrow the search to any subset. Therefore, it is normally not a good idea to use static class variables solely for the purpose of transferring values between methods.

This brings up two interesting questions:

- If this isn't the real purpose of static class variables, what is?
- Is there a different way to transfer data between methods that is more acceptable?

Good questions. I will answer them soon.

5B.2.3 Dealing with Scanner and Methods

If you attempt to use a **Scanner** for the console in more than one method, you will run into problems if multiple **Scanners** are declared and instantiated local to the methods in which each is used. The reason is that once **System.in** is closed for one instance of **Scanner**, it cannot be reopened, even if you try to redeclare a new **Scanner** object for **System.in**. Here's an example of the problem:

```
// inside first method ------------------------
Scanner input1 = new Scanner(System.in);
input1.nextLine();  // okay

...

input1.close();    // sets up problem for next attempt



// inside later method ------------------------
Scanner input2 = new Scanner(System.in);
input2.nextLine();  // run-time error
```

Even if there is only one input method that reads from the console it will be a problem if that method is called more than once.

Possible solutions:

1. (Good) Do not close the console in the first method. This is not optimal because you will get a compiler warning which you'll have to ignore.
2. (Better) Make the **Scanner** object **static** in the main class. It will then be "global" and accessible to all methods of that class. Open it once at the start of **main()** and then close it later in **main()** when you are done with it.

Here's how to implement solution #2:

```
import java.util.Scanner;

public class Foothill
{
   // shared Scanner reference for all methods in main Foothill class
   static Scanner input;

   public static void main(String[] args)
   {
      ...

      // open scanner once near the top of main
      input = new Scanner(System.in);

      // get some input
      userInFirst = getSomeInput();

      // get some more input
      userInSecond = getMoreInput();

      // done with all input so close now
      input.close();

      ...
   }

   // method definitions:
   public static String getSomeInput()
   {
      ...

      // no need to instantiate input.  it is static.  just use it
      theString = input.nextLine();

      ...
   }

   public static String getMoreInput()
   {
      ...

      // no need to instantiate input.  it is static.  just use it
      theString = input.nextLine();

      ...
   }
}
```

Section 3 - Parameters and Arguments

5B.3.1 Passing Arguments to Methods

There is still a little ugliness in our mortgage calculator. Look at **main()**. There is one statement that is ugly.

```
// main method
public static void main (String[] args)
{
   double answer;

   stateInstructions();
   getInput();
   answer = computeMonthlyPayment();
   System.out.println("\n\nMonthly Payment: "
         + answer);
   sayGoodbye();
}
```

We don't like to see main micromanaging. After all, it hired the services of **getInput()** to gather data from the user, so why should it dirty its hands with the output? It seems the **sayGoodbye()** method is a better place to report the results since it is already doing output anyway. But how do we get the answer into that method?

Use ***static class variables***? We could, but we hate to use them unless we are forced to. Remember my admonitions about only using member data for really important information and when we must. There is a better way.

We have used ***class variables*** and ***functional return*** values to pass information back and forth. Now we introduce ***arguments*** and formal ***parameters.***

Let's say we want a ***method*** that adds two numbers and returns the sum to the ***client***. Instead of using ***class variables***, we can pass the two numbers down to the method through the ***method call*** by placing them inside the ***functional parentheses***:

```
y = adder(3,5);
```

The values placed inside the parentheses are called ***arguments to the method,*** or just ***arguments***. ***Arguments*** can be constants, objects or expressions:

```
y = adder(x,z);
y = adder( x*3-1, 5/z);
y = adder(y,y);
```

When ***defining*** the ***method***, the ***method header*** must declare that it is going to accept two ***arguments***, and specify what their types are:

```
   int adder(int a, int b)
   {
      // definition of adder
   }
```

The a and b are called ***formal parameters.*** They are nothing more than ***local variables*** used inside the ***method*** whose values are obtained from the ***arguments*** passed down from the client.

In the ***method call***:

```
   y = adder(3,5);
```

the value of **3** is passed to the **a** (equivalent to **a=3** inside the method), and **5** to the **b**.

In the ***call***

```
   y = adder(x,z);
```

the contents of **x** are copied into the ***formal parameter*** **a**, and the contents of **z** are copied into **b**.

```
   y = adder( x*3-1, 5/z);
```

In the above, the two expressions are evaluated, and their results are copied into **a** and **b** respectively. Here is a **double** version of **adder**():

```
import java.lang.*;

public class Foothill
{
   public static void main (String[] args)
   {
      double x = 10.2, y = 35.9, z;

      z = adder(x,y);
      System.out.println( x + " + " + y + " = " + z);
   }

   // method adder ------------------
   static double adder(double x, double y)
   {
      double temp;

      temp = x+y;
      return temp;
   }
}

/* ------ a sample run for this program --------

10.2 + 35.9 = 46.099999999999994

-----------------------------------------------  */
```

```
(Really ugly version of 46.1, eh?  Computers!)
```

5B.3.2 Matching Types between Arguments and Parameters

The ***data type*** of the ***arguments*** must match, *or at least be compatible with* the type of the ***formal parameter***. If you were to think of the ***formal parameter*** as the left hand side (***LHS***) of an ***assignment statement***, and its incoming ***argument*** as the right hand side (***RHS***), then the compiler will not complain about the method call if it wouldn't complain about that assignment statement.

```
int x = 10, y = 35;
double z;

z = adder(x, y);
z = adder('V', '?');
```

are okay because:

```
a = x;
b = y;
```

and

```
a = 'V';
b = '?';
```

are acceptable - the implicit numeric conversions of Java kick in. (Can you remember why 'V' and '?' can be widened implicitly to ints or doubles?). But if **adder()** were defined, instead, to take **float** arguments:

```
static double adder(float x, float y)
```

then this would generate a compiler error:

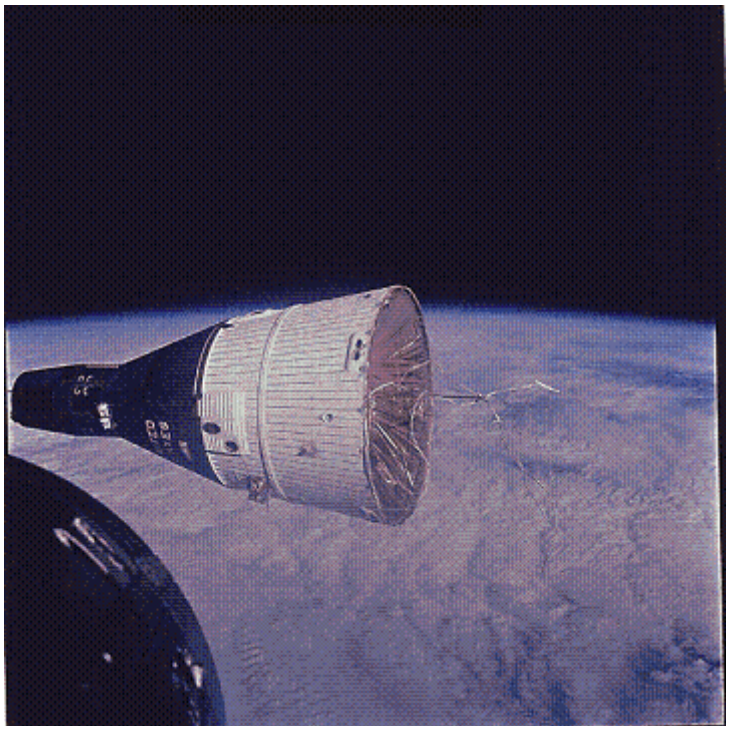
```
double x = 10, y = 35;
double z;

z = adder(x, y);
```

for the same reason that this would:

```
a = x;
```

You can't assign a **double** to a **float** without explicit ***type cast***.

Before we apply all of this to our mortgage calculator, let's continue talking about parameters and methods.



Section 4 - Formal Parameters and Overloading

5B.4.1 Parameters vs. Class Variables

Many students ask why use ***parameters*** at all, since we can do the same thing with ***class variables*** more easily.

The answer is that the question contains a fallacy. We can't do the same thing, and, besides, it wouldn't be any easier if we could. Here's why.

Consider our friend **adder**():

```
double adder(double a, double b)
{
   return a + b;
}
```

(I got rid of **temp**, because it was not really needed.) Using ***parameters*** (as we should) we can call **adder**() with many different arguments:

```
y = adder(3,5);
y = adder(x-1, z+3);
y = adder(a,b); // a, b local to main,
                // different from adder's params
```

However, what if we removed the ***formal parameters*** from the definition of **adder()** and instead tried to use ***static class variables***, **a** and **b** to pass the information?

```
public class Foothill
{
   // member data
   static double a, b;

   static double adder()
   {
      return a + b;
   }

   // rest of class ...
```

The method calls would look very different and much more confusing. We could no longer pass **adder()** arguments. Instead we would have to "prep" the **static class members** prior to the method call:

```
   a = 3;
   b = 5;
   y = adder();
```

or

```
   a = x - 1;
   b = z + 3;
   y = adder();
```

While this is bad enough, if we were to try to translate this call:

```
   y = adder(a,b); // a, b local to main,
                   // different from adder's params
```

into a similar parameter-less function call of three statements, we would have a real problem. I know this might be confusing to you, and it should be - this sort of mind bending confusion is caused when ***local variables*** have the same names as ***class variables*** making the parameter-less version of this call impossible. (If you are not confused by all this, you can see what you would have to do to make it work. But don't try too hard - the whole point is that using ***parameters*** rather than static ***class variables*** is the right thing to do.)

All of this busy nonsense defeats the purpose of ***methods***, which is to hide the messy details from the client and tuck them away into modular functions.

A second reason, equally important, is the "visibility" of ***class members***. As you know, all methods in the class can use and modify them. The more members you use, the more prone your program is to having one of the ***methods*** accidentally modify them.

5B.4.2 The Local Behavior of Formal Parameters

Look at this hypothetical method:

```
float adder(float a, float b)
{
   return a + (b++);
}
```

If it is called by:

```
y = adder(one, two);
```

is the value of **two** changed in the ***client*** when the call is complete?

No! If you go back to my description of how ***arguments*** are passed to ***methods***, you see that the ***value*** of the ***argument*** is copied down into the ***formal parameter***. That ***parameter*** is ***local***, and is only a *copy* of the client's object. This is a good thing. It prevents the method from modifying the client's data.

If the client wants a value changed it can go back to using ***class variables***, but we now know that ***class variables*** are not meant for this purpose. Or it can copy the ***return*** value into one of its objects:

```
y = f(a, b, c);
```

But what if we don't want to use ***class variables***, and what if we need more than one value modified by the method? This actually happened to us in the **getInput()** method of our mortgage calculator. We needed three objects returned to the client and since functional returns can give us only one, we were forced to use static class variables. We will have to leave that imperfection as is this week, and come back to that question when we have learned more about ***classes***.

5B.4.3 An Important Summary

This last discussion was so important it is worth summarizing without all the excess verbiage.

Changes to Parameters from inside Methods

When you pass primitive data (like **ints, doubles,** etc.) into a ***method*** through the argument mechanism (i.e., by placing variables inside the parentheses), the ***method cannot make any permanent change to those arguments***. When the method returns to the client, whatever values those variables had before are still there.

Notice that in the above statement I said this applies to ***primitive data***. That's because when we get to more complex data types, specifically ***classes***, we will see that ***objects*** of these ***classes***,

which are often passed into ***methods*** as ***arguments***, can and do get modified by the ***method***. Keep this in mind as we move forward.

Consider the method call:

```
myMethod( cat );
```

If **cat** is a ***primitive data type***, its value will never be changed by a call to **myMethod()**. If **cat** is an ***object*** (a more complex data type of some ***class***) then its value *can* be changed by the ***method***.

Since we are not creating or using ***classes*** or ***objects*** much yet, this distinction will remain unimportant for the time being.

5B.4.4 Method Signatures

When we talk about a method (function) informally, we often omit the parameter details. I might say, *"**adder()** this ..."* or *"**adder()** that ... "* in a sentence, but the parentheses are placed there to emphasize that it is a method. They do not mean that the method takes no parameters. The signature of the **adder()** method is shown when we describe it fully. It is **double adder(double a, double b)**. This is the official description of the method. It is called the ***signature*** of the function or method.

It is very important that you implement methods according to the signature that is specified. Even though I might put empty *parens* after the method name when discussing it in a sentence, like **println()** or **adder()** or **setAge()**, you must look for the *official* description and implement that. If the method signature that you provide does not match 100% with the method signature that is in the program spec, it is as if you did not do that method at all. That's right -- a*n incorrect signature is worth nothing*. The reason is that the rest of your project group assumed that you were writing *to spec* and if you did not, they can't use any of your method. They have to go back and write the method correctly from scratch. It is as if you did not exist! So this is not a fussy detail, but a real requirement. Don't get creative with methods. Write *to spec*.

5B.4.5 Method Overloading

Two methods with different signatures like **GetInstructions()** and **double adder(double a, double b)** are clearly distinct - they do different things, have different names and take different parameters. But can we have two methods with the same name that take different parameter types, i.e., two methods with the same name but different signatures, like **void addTime(double minutes)** and **void addTime(String minutes)**? Yes we can. We simply define the two methods separately, as if they were completely distinct (which, in fact they are). Each function header will differ only in that the parameter list will be different. Each can behave differently.

Why would we do this?

1. If both methods do virtually the same thing, we may want to give them the same name.
2. If we want to allow clients to call the method either using one type of parameter (double) or another (String).

If you give two methods (functions) the same name but a different parameter list, then you are overloading the methods (functions)

Here is a simple example of function overloading that demonstrates the idea:

```
import java.util.Scanner;

public class Foothill
{
   // declare a (rare) global that can be used by all for console input
   static Scanner input_stream = new Scanner(System.in);
   // -------  main --------------
   public static void main(String[] args)
   {
      displayTemperature(98.8);
      System.out.println();
      displayTemperature("99.4");
      System.out.println();

      sayThanks();
      System.out.println();
      sayThanks("I just wanted to tell you how much we enjoyed the movie.");
      System.out.println();   }

   // main methods (non-Patient methods)
   static void displayTemperature(double temp)
   {
      System.out.println("The patient's temperature is: " + temp);
   }
   static void displayTemperature(String temp)
   {
      System.out.println("The patient's temperature is: " + temp);
   }

   static void sayThanks()
   {
      System.out.println("Thank you!");
   }

   static void sayThanks(String special_note)
   {
      System.out.println(special_note);
      sayThanks();
   }
}
```

```
/* ------------- RUN (OUTPUT) ----------------------
The patient's temperature is: 98.8

The patient's temperature is: 99.4

Thank you!

I just wanted to tell you how much we enjoyed the movie.
Thank you!

--------------- END OF RUN ------------------------- */
```

Notice that we can call one overloaded variant (**sayThanks()**) from another (**sayThanks(string)**), and this is often desirable. If we are going to do many of the same tasks in both methods, then we should try to concentrate those common tasks into *one* of the variants and let the *other* variant call the *first* to reduce code duplication.

Section 5 - Mortgage Take 4

5B.5.1 Cleaning It All Up

We had the following remaining problems in our mortgage calculator:

1. We were doing output directly in the **main()** rather than delegating that to the **sayGoodbye()** method
2. The result had too many fractional digits displayed to the right of the decimal point.

We learned how to solve the fractional digit issue a few lessons ago using NumberFormat, and we just now learned how to pass the answer to a method rather than printing it out in main(), directly. So, we will give our final rendition of the mortgage calculator. Here it is. Look for and understand the changes:

```
import java.text.*;
import java.util.*;

public class Foothill
{

   // class variables shared by more than one method
   static double dblPrincipal, dblRate, dblYears;

   // main method
   public static void main (String[] args)
   {
      double answer;

      stateInstructions();
      getInput();
      answer = computeMonthlyPayment();
      reportResults(answer);
   }

   // gives an overview to user
   public static void stateInstructions()
   {
      String instructions;

      instructions =
         "The following program will calculate the \n"
         + "monthly payment  required for a loan of D dollars \n"
         + "over a period of Y years at an annual \n"
         + "interest rate of R%.";
      System.out.println(instructions);
      System.out.println("---------------\n");
   }

   // gets input, stores in static class variables
   public static void getInput()
   {
      Scanner input = new Scanner(System.in);
      String prompt, strUserResponse;

      // get principal
      prompt = "\nEnter amount of the loan. (only use numbers, \n"
         + "please, no commas or characters like '$')\n"
         + "Your loan amount: ";
      System.out.print(prompt);
      strUserResponse = input.nextLine();
      dblPrincipal = Double.parseDouble(strUserResponse);

      // get interest
      prompt = "\nNow enter the interest rate (If the quoted rate is "
         + " 6.5%, \nfor example, enter 6.5 without the %.)\n"
         + "Your annual interest rate: ";
      System.out.print(prompt);
      strUserResponse = input.nextLine();
      dblRate = Double.parseDouble(strUserResponse);
```

```
      // get length of loan
      prompt = "\nEnter term of the loan in years: ";
      System.out.print(prompt);
      strUserResponse = input.nextLine();
      dblYears = Double.parseDouble(strUserResponse);
   }

    // computes and returns answer
    public static double computeMonthlyPayment()
    {
       // local variables needed only in this method
       double dblTemp, dblPmt, dblMonths, dblMoRt;

       // convert years to months
       dblMonths = dblYears * 12;

       // convert rate to decimal and months
       dblMoRt = dblRate / (100 * 12);

       // use formula to get result
       dblTemp = Math.pow(1 + dblMoRt, dblMonths);
       dblPmt = dblPrincipal * dblMoRt * dblTemp
          / ( dblTemp - 1 );

       // now that we have computed the payment, return it
       return dblPmt;
   }

    // sign off
    public static void reportResults(double result)
    {
       String signoff;
       NumberFormat bucks =
          NumberFormat.getCurrencyInstance(Locale.US);

       signoff =
          "\nThanks for using the Foothill Mortgage Calculator. \n"
          + "We hope you'll come back and see us again, soon.";

       System.out.println("Monthly Payment: "
             + bucks.format(result));
       System.out.println(signoff);
    }
}
```

Look at **main().** It is almost perfect. We have renamed **sayGoodbye()** to **reportResults()** which has the goodbye chatter combined with the display of the payment. The **NumberFormat** is also included as a local declaration inside **reportResults()**. So, we have incorporated our most recent new concepts into our program.

We only wish we could get rid of the ***static class variables*** and pass them between **getInput()** and **computeMonthlyPayment()** as parameters somehow. But that won't work yet (why?). We'll have to save that for another day.

Next week we are going dive into the popular and important programming world of OOP (object-oriented programming). This will be an important step in your becoming a productive and valuable programmer. Your net worth is about to increase.

Prevent Point Loss

- **Separate output and calculation.** Methods that do calculations should not do input or output. Methods that do input or output should not calculate (except possibly, that input methods might do some range checking on the values entered by the user). (2 - 3 point penalty)

- **Do not use globals.** Whenever possible use parameter passing and functional returns to communicate data between methods; do not use global (i.e., static class) variables. In your assignments, globals (static class variables) will never be necessary as a means of passing information back-and-forth to methods, and you will always lose points if you use them to do so. (2 - 4 point penalty)

- **Make sure your method *signature* matches the *spec*.** Don't assume that the method does not take parameters just because we talk about it informally with empty parentheses, as in ***adder()***. Find the official signature, like ***double adder(double a, double b),*** and write to *that spec*. Even one missing or incorrectly typed parameter will result in a large point loss since this renders your method useless to others in the group -- none of their code will work unless your method signature matches the design spec. (5 - 10 point penalty)