Section 1 - Objects as Parameters to Methods

7B.1.1 Modifying Parameters in Methods

We've already seen that when we pass ***primitive data*** as ***arguments*** to ***methods***, that data cannot be modified by the method in such a way that it changes after the method call is complete. Whatever value it had before, it still has. Here is an example:

```
public class Foothill
{
   public static void main (String[] args) throws Exception
   {
      int chloesWagFrequency = 30;

      System.out.println("Chloe's wag frequency is "
         + chloesWagFrequency + " wags per minute.");
      promisePieceOfCarrot( chloesWagFrequency );
      System.out.println("Chloe's wag frequency is "
         + chloesWagFrequency + " wags per minute.");
   }

   // a would-be modifying method
   static void promisePieceOfCarrot( int freq)
   {
      freq *= 3;
   }
}
```

Here's the run:

```
Chloe's wag frequency is 30 wags per minute.
Chloe's wag frequency is 30 wags per minute.
```

No surprise there.

7B.1.2 Passing Objects to Methods

Now we consider ***objects*** and ***object references.*** We already know that when we declare an ***object reference***, we have no data yet. We have to ***instantiate*** the object with a second statement (using the **new** keyword). Similarly, when we pass ***objects*** to ***methods*** or ***assign objects*** to each other, we are only ***passing*** or ***assigning*** the ***references***. The actual object doesn't get copied or moved.

This actually works to our favor in methods. If we pass an ***object reference*** to a ***method***, then modify that ***object*** inside the ***method***, the results are felt permanently back in **main()**. Here is an example:

```
public class Foothill
{
   public static void main (String[] args) throws Exception
   {
      Dog chloe = new Dog();

      // public access not usually good, but do it for this example
      chloe.name = "Chloe of Hood Canal";
      chloe.wagsPerMinute = 30;

      System.out.println(chloe.name + "'s wag frequency is "
         + chloe.wagsPerMinute);
      promisePieceOfCarrot( chloe );
      System.out.println(chloe.name + "'s wag frequency is "
         + chloe.wagsPerMinute);
   }

   // hold out a carrot
   static void promisePieceOfCarrot( Dog dog )
   {
      dog.wagsPerMinute *= 3;
   }
}

class Dog
{
   // non-private access not usually good, but do it for this example
   public String name;
   public int wagsPerMinute;
}
```

Caution

In this example, I have declared the data using the **public** keyword to allow the client access to the data. This is rarely good, and I only did it here to keep the code short. Normally, in a real program or assignment, we would not think of making member data explicitly **public** (or even **default** , meaning that there is no access modifier) -- which gives other classes ill-advised access to this data.)

Here is a copy of the run:

```
Chloe of Hood Canal's wag frequency is 30
Chloe of Hood Canal's wag frequency is 90
```

So this works! We now have a way, if we want, of sending data into a method and returning from the method with the data modified.

This is not to be abused, however. It is still preferred to pass modified data back, not through one of the formal parameters, but by a ***functional return***. This way, we can leave the passed-in arguments unchanged in case the client wanted them left as is.

Summary: When passing a ***primitive*** to a ***method***, its value *cannot be modified* upon return to the client. When passing a class ***reference*** to a ***method***, the ***member data*** of that ***object*** (i.e. ***fields*** of the object pointed to by that ***reference***) *can be modified permanently* by the ***method***.

Just remember that **Chloe's** wagging tail goes faster after she sees the carrot.

7B.1.3 Instance Methods That Take Objects as Parameters

We showed a static method that takes **Dog** objects as parameters (see above). Can we pass an object to an ***instance method*** of a class? Sure, if it makes sense to do so. If we have a class **Dog**, we normally would not design an instance method in that class that takes a single **Dog** object as a parameter, because there is automatically one object you get for free with every instance method invocation, namely the "***this***" object, inherent in every instance call. So it usually doesn't make sense to pass an object of the same class to an instance method of that class for this very reason. You don't need to. You get one object for free without passing any parameters!

But sometimes we have a function that wants a *second* object, and if so, we may pass that second object as a parameter. For example, maybe we want to see the *difference* in wag rates between a one dog and another. We could write this as a ***static function*** that takes two **Dogs**, but here's a another idea: we can make it an ***instance method*** of the **Dog** class, with one **Dog** being the object doing the calling (***this***) and the other **Dog** being the parameter.

From the client point of view the call would look like this (perhaps):

```
System.out.println("The difference in wag frequency between "
   + chloe.name + " and " + lily.name
   + " is " + chloe.differenceInWags( lily ) );
```

Let's see how this works in the instance method definition:

```
public int differenceInWags( Dog otherDog )
{
   return wagsPerMinute - otherDog.wagsPerMinute;
}
```

As you see, it is straightforward -- the this object's member is accessed, as we already learned, without any dereference inside the method; **wagsPerMinute**, stark naked in the method definition, can mean only one thing: the ***this*** object's **wagsPerMinute**. The parameter's **wagsPerMinute**, on the other hand, is distinguished from that because we use the formal parameter, **otherDog**, to dereference it: **otherDog.wagsPerMinute**.

Observation

Even if **wagsPerMinute** had been **private**, we could still have accessed both the **this** object's member and the parameter's member without using accessor methods. We would just refer to the members as we did in the above method. The private data of the class is available to any class method (in the same class), even if we are accessing some non-**this** object's member, as we did above with the parameter **otherDog**.

Here is the full program from which the above were taken:

```
public class Foothill
{
   public static void main (String[] args) throws Exception
   {
      Dog chloe = new Dog();
      Dog lily = new Dog();

      // public access not usually good, but do it anyway
      chloe.name = "Chloe of Hood Canal";
      chloe.wagsPerMinute = 30;
      lily.name = "Lily the Waggiest";
      lily.wagsPerMinute = 75;

      System.out.println("The difference in wag frequency between "
         + chloe.name + " and " + lily.name
         + " is " + chloe.differenceInWags( lily ) );
   }
}

class Dog
{
   public String name;
   public int wagsPerMinute;

   public int differenceInWags( Dog otherDog )
   {
      return wagsPerMinute - otherDog.wagsPerMinute;
   }
}
/* ---------------- Paste of Run --------------------

The difference in wag frequency between Chloe of Hood Canal and Lily the
Waggiest is -45

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

Section 2 - Building on the Past

7B.2.1 Mortgage, Resurrected

We left a thread hanging back when we did our mortgage calculator. All of the individual tasks were neatly tucked away in their own methods, but there was one irritating detail. The **input()** method was not capable of returning its input data through either ***functional returns*** or ***parameters*** (do you remember why neither of these modes was capable of transporting the three input values back to main()?). We had to resort to ***static class variables*** (***dblPrincipal, dblRate, dblYears***) as a means of communicating the info among our various methods. And, as I said at the time, ***static class variables*** are not really meant to be used for this purpose. Instead, they should have a special meaning for our class (like the **population** variable in a **Dog** class, which kept a count of the number of **Dogs** instantiated at any given time).

Classes and objects give us a way around this dilemma. Before we present this solution, let's repeat the mortgage calculator as we left it. Review it briefly to orient yourself for the discussion that follows:

```
import java.util.Scanner;
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) throws Exception
   {
      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);
   }
}

/* ----------------- Sample Run --------------------------

The following program will calculate the
monthly payment  required for a loan of D dollars
over a period of Y years at an annual
interest rate of R%.
---------------


Enter amount of the loan. (only use numbers,
please, no commas or characters like '$')
Your loan amount: 200000

Now enter the interest rate (If the quoted rate is  6.5%,
for example, enter 6.5 without the %.)
Your annual interest rate: 4.3

Enter term of the loan in years: 15
Monthly Payment: $1,509.62

Thanks for using the Foothill Mortgage Calculator.
We hope you'll come back and see us again, soon.


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

Section 3 - Client View of the Class

7B.3.1 main()'s View of the MortgageData Class

We need a way to neatly encapsulate all the data of our mortgage loan into a single object. Once we do that, we have options for solving the problem of transferring that data between methods. We can ***return*** an object as a ***functional return*** or we can pass the ***object*** as a ***parameter*** to a method. Each of these two new alternatives works since, unlike primitive data (**ints, doubles,** etc.) ***object references*** allow their objects to "remember" changes made to the object after they are passed back to the client. And that, after all, is the point.

The solution is going to be simple. But it is not going to be short. Even the simplest class, when equipped with all of the necessary ***accessor*** and ***mutator methods***, ***constructors*** and data tends to get long, fast. But the content of the class is very repetitive, so once you see how one method works, the other methods will be clear.

Let's start with the client that we would like to be able to write, and work backwards from there.

```
public class Sample
{
   public static void main (String[] args)
   {
      double answer;
      MortgageData loan;

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

   //  ... rest of main class
}
```

Notice two things about this main():

1. There are no class variables in the main class **Sample**. The data is now completely described and passed as objects which are local to main().
2. There is a new data type called **MortgageData**. Without seeing the details of this type, we fully understand its purpose. It holds the information that is returned by **getInput()** and is used to transport that information to **computeMonthlyPayment().**

You have to appreciate the beauty and simplicity of this **main()** and what it represents. We often write such client methods before defining the data that it uses. This will help us understand the class **MortgageData** from the "class user" point of view, which later leads to the proper definition of the class from the "class designer" perspective.

7B.3.2 getInput()'s View of the MortgageData Class

Main() is not the only place, of course, that the **MortgageData** objects will be used. As you see, **getInput()** returns such an object, so it must be ***instantiating*** one such object to return and also filling that object with useful data in the interim. Let's examine a small portion of the new version of **getInput()**, looking at the **MortgageData** class from a class user (not class designer) perspective:

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

   // get principal
   do
   {
      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();
      dblResponse = Double.parseDouble(strUserResponse);
   }
   while ( !userData.setPrincipal(dblResponse) );

   // continue on with rest of getInput() definition ...

}
```

First look at the *method header* (aka. the "*method signature*"):

```
public static MortgageData getInput()
```

The fact that a class name **MortgageData** is directly in front of (i.e., to the left of) the method name **getInput()** tells us that this method is going to *return* a **MortgageData** object.

We are ***instantiating*** a **MortgageData** object with the intention of returning it as a functional return to our client (or more accurately, returning its reference to the client). Once created, we can use the ***accessor*** method **setPrincipal()** to attempt to set the principal of the loan.

As you can imagine, this object will have not only a principal member, but also ***interest rate*** and ***loan term*** members. These three **doubles** will be the primary data of our class. Without seeing the definition at this point, we can still

anticipate how the class is to be used. Since setting class data with an ***accessor*** (***mutator***) method should filter out bad data, it is common for such a method to return a **boolean**: **true** if the assignment was successful and **false** if the value was out-of-range. We are going to make our methods behave exactly this way, and as you see, we can use the **boolean** return value to help us control an input ***do/while loop***. Study the above ***while*** phrase until you fully understand why it works.

Next, we have to explain the **MIN_LOAN** and **MAX_LOAN** members. We use our sleuthing skills and knowledge of classes to figure out what these are. If a class member is accessible from outside the class then it is **public**. So this is a **public *member***. Also, if a member is ***dereferenced*** through an ***object*** it is an ***instance variable,*** while if it is ***dereferenced*** through the ***class name*** it is a ***static class variable***. Since we are using the class name, **MIN_LOAN** and **MAX_LOAN** must be ***public static members.*** Their name tells us what they mean: These are two **doubles** that determine the range of values our class will accept for its principal loan amount.

This is another example of why we sometimes declare members to be ***static class members***. These min and max values are obviously the same for every object in the class.

7B.3.3 computeMonthlyPayment()'s View of the MortgageData Class

We continue to look at how our client is going to use this new class. **computeMonthlyPayment()** takes an object of this class as a ***formal parameter***. It uses that object to extract the three primitive **doubles** that it needs for the computation. Here is a code fragment that shows this, focusing on the **getYears()** ***accessor*** method of the **MortgageData** class:

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

   // convert years to months
   dblMonths = loan.getYears() * 12;

   // continue with the rest of the definition of the method ...

}
```

This gives us clues about how we should write our **MortgageData** class.

Section 4 - Designing the Class

7B.4.1 The Member Data

There are two types of member data we need:

1. ***Instance members*** for the 3 doubles that describe a loan.
2. ***Static class members*** that hold the extreme values of our data

We will add a new modifier, **final**, to our vocabulary. Since these static class variables are going to be constant, we will ensure this by declaring them to be so. This will prevent any inadvertent re-assignment or change to those static members. That is what the keyword **final** means.

Here is the portion of the **MortgageData** class that covers the definition of both the ***instance data*** and the ***static data:***

```
class MortgageData
{
   private double dblPrincipal;
   private double dblRate;
   private double dblYears;

   // class constants
   static final double MIN_LOAN = 1;
   static final double MAX_LOAN = 100000000;
   static final double MIN_RATE = .00001;
   static final double MAX_RATE = 25;
   static final double MIN_YRS = 1;
   static final double MAX_YRS = 100;

   // ... continue on, defining class methods ...

}
```

This is very common in a class. The important data is kept **private**. Data meant to be constants are **public** or **default** and are declared as **final**. They are commonly capitalized to emphasize their meaning as constants.

7B.4.2 Accessor Methods of the Class

Here are two class *accessor* (meaning *accessor* and/or *mutator*) methods, **setRate()** and **getRate()**. The remaining accessor methods **setPrincipal(), getPrincipal(), setYears()** and **getYears()** are defined similarly.

```
class MortgageData
{
   // data definition and constructors skipped, and then:

   public boolean setRate (double rt)
   {
      if (rt < MIN_RATE || rt > MAX_RATE)
         return false;
      dblRate = rt;
      return true;
   }

   public double getRate()
   {
      return dblRate;
   }

   // other methods defined ...

}
```

This is a good time to emphasize that the class methods do not have to use the class name to dereference the class variables. **MIN_RATE** and **MAX_RATE** in the above method demonstrate this point.

7B.4.3 Two Overloaded Constructors

We complete the picture by presenting the ***constructors*** for the class.

We have seen examples of ***constructor overloading*** with our **Galaxy** class. We do that again here. We define two separate ***constructors***, one that takes no parameters (a ***default constructor)*** and one that takes three parameters. If the client passes parameters to the constructor in the instantiation line, then the 3-parameter constructor gets invoked. If not, the default constructor is called.

This can be done with any ***method***, of course, not just ***constructor methods.*** But overloading constructors is a very powerful and common technique.

```
class MortgageData
{
   // data definition skipped, and then:

   public MortgageData()
   {
      // default values (or assign directly if you prefer)
      setPrincipal(MIN_LOAN);
      setRate(MIN_RATE);
      setYears(MIN_YRS);
   }

   public MortgageData(double prin, double rt, double yr)
   {
      if (!setPrincipal(prin))
         setPrincipal(MIN_LOAN);
      if (!setRate(rt))
         setRate(MIN_RATE);
      if (!setYears(yr))
         setYears(MIN_YRS);
   }

   // proceed to define other methods ...

}
```

So we can instantiate a **MortgageData** object either this way:

```
   MortgageData loan1 = new MortgageData();
```

or this way:

```
   MortgageData loan2 = new MortgageData(200000, 6.125, 30);
```

Also, note that the constructor makes use of the other ***mutator*** functions **setPrincipal(),** etc. There is no need to set the data of the class manually if we can use a ***mutator*** method that does the work of filtering the data. We don't have to test the data for validity again since it is already done for us in these ***mutators***.

You should always reuse methods of classes this way. Duplicating logic leads to errors and makes your code longer than it need be.

Section 5 - Mortgage Calculator Version 5

7B.5.1 The Listing

This is a long listing, but we have already discussed every aspect of it. I put it here for you so you can copy and paste it for experimentation.

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

public class Foothill
{
   // main method
   public static void main (String[] args) throws Exception
   {
      double answer;
      MortgageData loan;

      stateInstructions();
      loan = getInput();
      answer = computeMonthlyPayment(loan);
      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 MortgageData getInput()
 {
    Scanner input = new Scanner(System.in);
    String prompt, strUserResponse;
    double dblResponse;
    MortgageData userData = new MortgageData();

    // get principal
    do
    {
       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();
       dblResponse = Double.parseDouble(strUserResponse);
    }
    while ( !userData.setPrincipal(dblResponse) );

    // get interest
    do
    {
       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();
       dblResponse  = Double.parseDouble(strUserResponse);
    }
    while ( !userData.setRate(dblResponse) );

    // get length of loan
    do
    {
       prompt = "\nEnter term of the loan in years: ";
       System.out.print(prompt);
       strUserResponse = input.nextLine();
       dblResponse = Double.parseDouble(strUserResponse);
    }
    while (!userData.setYears(dblResponse));

    return userData;
}
```

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

      // convert years to months
      dblMonths = loan.getYears() * 12;

      // convert rate to decimal and months
      dblMoRt = loan.getRate() / (100 * 12);

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

class MortgageData
{
   private double dblPrincipal;
   private double dblRate;
   private double dblYears;
```

```
// class constants
static final double MIN_LOAN = 1;
static final double MAX_LOAN = 100000000;
static final double MIN_RATE = .00001;
static final double MAX_RATE = 25;
static final double MIN_YRS = 1;
static final double MAX_YRS = 100;

public MortgageData()
{
   // default values (or assign directly if you prefer)
   setPrincipal(MIN_LOAN);
   setRate(MIN_RATE);
   setYears(MIN_YRS);
}

public MortgageData(double prin, double rt, double yr)
{
   if (!setPrincipal(prin))
      setPrincipal(MIN_LOAN);
   if (!setRate(rt))
      setRate(MIN_RATE);
   if (!setYears(yr))
      setYears(MIN_YRS);
}

public double getPrincipal()
{
   return dblPrincipal;
}

public double getRate()
{
   return dblRate;
}
public double getYears()
{
   return dblYears;
}

public boolean setPrincipal (double prin)
{
   if (prin < MIN_LOAN || prin > MAX_LOAN)
      return false;
   dblPrincipal = prin;
   return true;
}

public boolean setRate (double rt)
{
   if (rt < MIN_RATE || rt > MAX_RATE)
      return false;
   dblRate = rt;
   return true;
}
```

```
   public boolean setYears (double yr)
   {
      if (yr < MIN_YRS || yr > MAX_YRS)
         return false;
      dblYears = yr;
      return true;
   }
}

/* ----------------- Sample Run --------------------------

The following program will calculate the
monthly payment  required for a loan of D dollars
over a period of Y years at an annual
interest rate of R%.
---------------


Enter amount of the loan. (only use numbers,
please, no commas or characters like '$')
Your loan amount: 200000

Now enter the interest rate (If the quoted rate is  6.5%,
for example, enter 6.5 without the %.)
Your annual interest rate: 4.3

Enter term of the loan in years: 15
Monthly Payment: $1,509.62

Thanks for using the Foothill Mortgage Calculator.
We hope you'll come back and see us again, soon.


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

Prevent Point Loss

- **Use symbolic names, not literals.** Never use a numeric literal like 1000, 3 or 52 for the size of an array, or the maximum value of some data member in your code. Instead create a symbolic constant and use the constant. In other words, use ARRAY_SIZE or MAX_CARDS, not 1000 or 52. (1 point penalty)