Section 1 - Static Members

Introduction to Week Seven

This week we will complete our introduction to ***classes*** and ***OOP*** by exploring ***static members***, discussing ***member objects*** and introducing the **this** keyword (among other things). You will start to think like a professional OOP programmer in the next few days.

Reading

The optional textbook reading is handled as we have been doing all along: look up the terms with which you want more coverage in the index and see where it leads.

7A.1.1 Instance and Class Data

We said that a ***static class member*** is data defined in the class (but outside the method definitions) that is qualified with the **static** keyword. ***Instance members*** are those that do not have the **static** keyword. Let's learn the difference between these two kinds of member data.

Since a class is an abstract blueprint for a hand-rolled data type, we know that nothing really happens until we ***declare*** and ***instantiate*** one or more ***objects*** of that ***class***. Each time we ***instantiate*** an object, a new set of ***instance members*** is associated with that object, *distinct from the instance members of other objects* of the class. If you have two **Galaxy** objects, you have two separate sets of data, one for each object

In contrast, with ***static class members***, there is only one datum associated with the entire ***class***. If a member of a class is qualified by the **static** keyword, then no matter how many objects we instantiate, there is only one variable for that member, and it is shared by all the objects of the class.

Why would we want such a communistic variable to be in our class? There could be many reasons. Here is an example of just one:

```
public class Dog
{
   static long population = 0;
   long licenseNumber;

   // other members ...
}

// later on in the client class method ...
Dog fido = new Dog(),
    lucy = new Dog(),
    watson = new Dog();
```

We have declared three new **Dog** objects - in other words, we ***instantiated 3 dogs***. **fido, lucy** and **watson** each has its own personal copy of the **licenseNumber** data because **licenseNumber** is an ***instance*** member. However, they all share the one and only ***static member,* population**. This might be useful if, for instance, we wanted the **population** to be incremented each time a **Dog** was born, and decremented each time a **Dog** died. It is a value that is common to all dogs in the **Dog** class. But the **licenseNumber** is unique to each instance.

Exploring this a little further, we look back to our **Employee** example at the line:

```
   walter.age = 61;
```

which clearly makes sense to us. It is saying that the **age *field*** (I'll switch terminology to make sure you get used to both terms: ***member*** and ***field***) in the **walter *instance*** is being set to **61.** Similarly with a **Dog**:

```
   watson.licenseNumber = 999332222;
```

is how to modify **watson's** own **licenseNumber**. But what does:

```
   watson.population++;
```

do? Well it clearly increments **population**, but since **population** is a ***static class member,*** there is only one such ***field*** for the entire class, and we are modifying that field. Since there is no more reason to access this **population** field through use of the **watson** reference than there would be to access it through the **fido** or **lucy** reference, we are offered a preferred method of access to ***static class members***, namely, through the ***class name:***

```
   Dog.population++;
```

Here we see more clearly what we are doing. Usually, you will want to use the <u>**class name**</u> when dereferencing a ***static class field***.

7A.1.2 Initializing Static Members

As you know, we have a special method whose job it is to initialize all the ***instance member data*** for us. That method is called a ***constructor***, and it always has the same name as that of the class. This guarantees that whenever we instantiate an object of that class, its member data are all initialized to something reasonable.

```
   Galaxy gal1 = new Galaxy(), gal2 = new Galaxy();
```

These two objects now have their private data set because of the **Galaxy *constructor*** (go back and see what that constructor did, if you forgot).   Also, you may have followed my advice and created a constructor for the **Employee** class in the previous section.

Now that we have this concept of a ***static member,*** however, we have to figure out how to initialize it. Imagine the **population** member outlined above. We want to initialize it to 0. Fine.

Since it is a public member, we could have our client do it with a simple assignment statement, **Dog.population = 0;** But we should really make population **private**. Private instance data is usually initialized in the ***constructor***, but we can't use a constructor for private static data: *If, each time a **Dog** was born, the member **population** were set to **0**, we would always have a population of **0**!*

Instead, static members are normally initialized when they are declared. You see this with the population member here:

```
public class Dog
{
   static long population = 0;
   long licenseNumber;

   // other members ...
}
```

Section 2 - Static Methods

7A.2.1 Instance and Static Class Methods

This same ***static concept*** can be applied to ***class methods***. That is, we can place the **static** keyword in front of a class method to make it a **static** *class* method, in contrast to an *instance* method for that class. What does the **static** keyword mean in this case?

Actually, we've used **static *methods*** of other classes without knowing about it. **pow()** and **showMessageDialog()** are ***methods*** that belong to the **Math** and **JOptionPane** classes, respectively. We dereference them through the class names, not through any particular object of the class: **Math.pow()** and **JOptionPane.showMessageDialog(),** for example.

Let's see how this works in one of our own classes.

For this I choose a class called **Rectangle** which consists of the two numbers **width** and **height**. In addition to those two numbers there will be several ***methods***. Most of these ***methods*** will *not* have the **static** modifier in front of them, which makes them ordinary ***instance methods***, the kind that we have been defining this week. ***Instance methods*** must be ***dereferenced*** through a particular ***object instance***, and they usually operate on the private data of that one instance which is used to call them. Thus, when we called

```
gal1.setMagnitude(13.5);
```

we were asking the ***instance method*** **setMagnitude()** to operate on the private data of the Galaxy **gal1**.

In class **Rectangle** we will also introduce a **static *class method***, identified by the **static** modifier. This method is not meant to be called by using any particular **Rectangle *object***, but is ***dereferenced*** using the **Rectangle** class name, itself.

Before giving the full definition of the **Rectangle** class, let's just see how two of the ***method headers*** look and how they are used. First an ***instance method:***

```
   // -------  setWidthHeight Definition --------------
   public boolean setWidthHeight(double w, double h)
   {
      // definition omitted for now - sets private data to w and h
   }
```

Next a ***static class method:***

```
   // -------  whichIsBiggerRef --------------
   // returns a reference to the larger Rectangle of r1 and r2
   static public Rectangle whichIsBiggerRef(
         Rectangle r1, Rectangle r2)
   {
      // definition omitted - returns rectangle with larger area
   }
```

The function **setWidthHeight()** will be used to assign values to the data members **width** and **height**. It makes sense that this is an instance method since it needs an object on which to work. **setWidthHeight()** is a kind of ***accessor*** method because it accesses two of the private data of the class, **width** and **height**.

whichIsBiggerRef() takes two **Rectangle *objects*** as ***arguments*** and returns a ***reference*** to the one that has the larger area. Since all the data on which it needs to "think" is passed in as ***arguments***, there is no logical reason why this should be an instance method. We can sensibly use the class name, **Rectangle**, to ***dereference*** this ***method***. That's why we make it a ***static method***, rather than an ***instance method.***

As before, we can (probably in the ***main()*** method of a different class) ***instantiate*** a **Rectangle** object thusly:

```
   Rectangle r1;
   r1 = new Rectangle();
```

or, using the shorter notation:

```
   Rectangle r1 = new Rectangle();
```

then we may ***call*** the **setWidthHeight()** function by using this object as a ***handle***:

```
   r1.setWidthHeight(2, 5);
```

Now what do you suppose this function call does? Well that depends on how we define it, but you can guess that if we design it to behave rationally, it will set the **width** member and the **height** member to the values **2** and **5**, respectively.

Which **width** and **height** members? The ones belonging to the data object **r1**. Why **r1**? Because we used **r1** as a ***handle*** to ***call*** (or ***dereference***) the member function **setWidthHeight().**

Next we can do the same thing for a second rectangle:

```
   Rectangle r2 = new Rectangle();
```

then we may ***call*** the **setWidthHeight()** function by using this object as a ***handle***:

```
   r2.setWidthHeight(3, 4);
```

Now that we have two **Rectangle *instances***, we can compare them with our static method **whichIsBiggerRef()**. We'll need a **Rectangle** reference to capture the ***functional return***:

```
   Rectangle r3;

   r3 = Rectangle.whichIsBiggerRef(r1, r2);
```

Make sense? It probably would help if you saw the entire program and class definition.   We will do that soon. First a little diversion on a new topic.

Section 3 - This

7A.3.1 The Implied "this"

Pretty soon we are going to give the full definition of **Rectangle**. Let's consider the definition of the instance method **setWidthHeight().** We might try it like so:

```
   // -------  setWidthHeight Definition --------------
   public boolean setWidthHeight(double w, double h)
   {
      if ( w <= 0 || h <= 0 )
         // don't allow negative width or height
         return false;
      width = w;
      height = h;
      return true;
   }
```

We called the formal parameters **w** and **h**. Not very informative (I deduct points for this sort of poor variable or object naming). Really, what we are passing in is the **width** and **height** of the rectangle. So why don't we call these parameters **width** and **height**? We could try:

```
// -------  setWidthHeight Definition --------------
public boolean setWidthHeight(double width, double height)
{
   if ( width <= 0 || height <= 0 )
      // don't allow negative width or height
      return false;
   width = width;  // oops ... confusion!
   height = height;
   return true;
}
```

See what happened? We named the ***formal parameters*** the same as the ***instance variables***. Before, when we used the term **width** inside an ***instance method*** we meant the private data **width** of the calling object. But now we have a parameter with the same name, so there is ambiguity.

Recall that ***parameters*** are really just ordinary ***local variables*** or ***object references***. They are very similar to the variables or objects declared inside the function, the only difference being that ***parameters*** get assigned values automatically when the function is called (via the ***arguments*** passed down through the function call). Now, let's recall a fact about *any* local variable (be it a parameter or ordinary variable defined in the function).

When we declare a local variable, if it has the same name as an **instance** member, this local variable **hides** or **shadows** the instance member throughout the method definition.

What this means is that if we use the term **width** inside our newly formed method, we would be referring to the *local* **width** (the parameter), not the *member* **width**.

But what if we *want* to name the ***formal parameter*** using the word **width** and also refer to the ***member*** using the same word, **width,** later in the method? There is a special reference identifier called "***this***" that all instance methods can use. ***This*** can be used to access any of the ***instance members*** by dereference. So, even though the ***formal parameter*** **width** shadows the ***member*** **width**, making it seem inaccessible, we can still "get at" that member using the **"this."** notation:

```
this.width = width;
this.height = height;
```

Now we are safe. The LHS, **this.width,** refers to the ***member*** **width** of the calling object. The RHS, **width**, refers to the ***formal parameter*** **width** of that same name.

Now, we can name formal parameters the same as members <u>*if it makes sense to do so*</u> and still have a way to get at the members.

Here is how the method will look, once we have used this improved notation:

```
// -------  setWidthHeight Definition --------------
public boolean setWidthHeight(double width, double height)
{
   if ( width <= 0 || height <= 0 )
      // don't allow negative width or height
      return false;
   this.width = width;
   this.height = height;
   return true;
}
```

We recently learned that when we enter any ***instance method***, some ***object*** must have been used to get us there, and that ***object*** was called the ***"this" object***. In **main()**, for example, any hypothetical member *function-calls* we might invoke, like **fido**.bark() or **account_213**.Balance() or **maryann**.ChangeZipCode(), all required objects to call these methods. The word **fido** or **account_213**or **maryann** each represents the ***"this"*** object that initiated the function call. Well, that is the same ***"this"*** that I'm talking about here! When you get inside the method definition, you can't know which object did the call exactly. It might be **fido**, or **maryann** or **account_213**. So we use the keyword ***"this."*** inside that method definition if we want access to the calling object for some reason without actually knowing that object's specific name. Now I'm telling you that you can use the ***this*** notation to get at the object's members whether or not they are hidden by local variables.

Even if there is no local variable hiding an instance member, we can optionally place the ***"this."*** keyword in front of the instance member (or instance method) while inside an ***instance method definition***. This is a complex sentence, so read it again, then look at the following example in which we first refer to an instance method **clearSocSec()** without the ***"this."*** in front (old way), and next see it used with the ***"this."*** attached (new way) .

Recall this ***instance method:***

```
public boolean getOlder()
{
   if (age++ > 147 )
   {
      clearSocSec();
      return false;
   }
   else
      return true;
}
```

Above, we are implicitly calling the **clearSocSec()** method using the ***this*** object that got us into **getOlder()** in the first place. We could use the alternate syntax if it makes things clearer:

```
   public boolean getOlder()
   {
      if (age++ > 147 )
      {
         this.clearSocSec();
         return false;
      }
      else
         return true;
   }
```

I'll take this opportunity to point out that **getOlder()** is a method that modifies private data. This makes it a ***mutator*** even if it does not bear the time-honored name **setSomething()**. As a ***mutator***, it must not do input or output, and so it returns a **boolean** that the client can inspect and, based on the outcome, do any desired output. See the full example from last time to reinforce how that was handled.

Section 4 - Returning Objects

7A.4.1 Assignments with Objects

When we make an ***assignment*** between two **ints**:

```
   int x = 19, y;
   y = x;
```

There is little doubt what gets assigned. The value stored in **x** is copied into the variable location **y**. After this assignment statement, **x** and **y** can have separate and independent futures What happens to one will not affect the other. This is true because they are both ***primitive*** data types.

With ***objects*** of ***classes***, the same is not true. Let's say **r1** and **r2** are **Rectangle** references. ((Since shortly, I'm going to add a **label** member to the **Rectangle *class***, the code below will anticipate this addition.) Consider the following:

```
// create a new rectangle, r1, with dimensions 5 x 8
r1 = new Rectangle();
r1.label = 'a';
r1.setWidthHeight(5, 8);

// create a new rectangle, r2, with dimensions 2 x 2
r2 = new Rectangle();
r2.label = 'b';
r2.setWidthHeight(2, 2);

// now assign r1 to r2:
r2 = r1;
```

What just happened? We created two separate objects that were referenced by **r1** and **r2**, but then along came the troublesome assignment statement **r2 = r1.** Why was it troublesome?

There are two reasons.

1. Because the assignment is one between ***references***, not between ***objects***, all we really did was have **r2** point away from its old object and point, instead, to the object controlled by **r1**. Thus we lost (forever) the rectangle to which **r2** was pointing.
2. Now that we have two references pointing to the same object, if we change the object using one reference, say **r1**, the change will be felt if we later refer to the object using the other, say **r2**. We might expect **r2** and **r1** to have different futures, but this is no longer the case.

We might avoid problem 1 above by never ***instantiating* r2** in the first place. But what about the second issue? This problem persists.

There are ways to copy data from one ***object's fields*** into another ***object's fields*** without changing the ***references***. We won't go too deeply into this topic for now, but you already know how to do it the long-winded way if you wish. It would go something like this:

```
r2.width = r1.width;
r2.height = r1.height;
r2.label = r1.label;
```

etc. Of course, if the data were private, we would use ***accessor methods:***

```
r2.setWidth( r1.getWidth() );
r2.setHeight( r1.getHeight() );
r2.setLabel( r1.getLabel() );
```

What is important here is that you realize what assignment statements mean when we are dealing with ***object references***. Assigning the references, **r2 = r1**, is completely different from assigning the data in the members, one-by-one, as we are doing in the box immediately above.

7A.4.2 Functions that Return Objects

This same sort of analysis applies to ***methods*** that have ***return types*** that are class names, as in:

```
// -------  whichIsBiggerRef --------------
// returns a reference to the larger Rectangle of r1 and r2
static public Rectangle whichIsBiggerRef(
      Rectangle r1, Rectangle r2)
{
   // definition omitted - returns rectangle with larger area
}
```

As you see, we are going to ***return*** the larger **Rectangle**. But what, exactly are we returning, and how would we capture the result in the client?

The answer is, we are returning a ***reference***, and we would capture it usually with an ***assignment statement***. And now that we know an assignment statement only copies the ***reference***, this means that the data inside the object is not being returned. Only the reference is being returned.

So we had better look at the ***definition*** of this ***method*** to see what the heck is going on.

```
static public Rectangle whichIsBiggerRef(
   Rectangle r1, Rectangle r2)
{
   if ( r1.width * r1.height > r2.width * r2.height )
      return r1;
   else
      return r2;
}
```

So you see, we are taking two object references into the function and returning whichever one is larger. That's fine, as long as we realize this. If we call:

```
r3 = Rectangle.whichIsBiggerRef(r1, r2);
```

then by the end of the call, **r3** will be pointing to the same object as either **r1** or **r2**, depending on which is larger. If we later modify **r3** we will be also modifying **r2** or **r1**. That's probably not what we want. Is there a way around this? Yes. We can manufacture a fresh new object inside the method **whichIsBigger(),** different from both **r1** and **r2**, that has the same data as the larger of the two. Then, when the function returns, it will return a ***reference*** to an autonomous ***object*** whose future is not tied up with either **r1** or **r2**.

This warrants a separate ***method*** with a separate name, ***whichIsBiggerClone()***, since we are going to be cloning the larger of the two **Rectangles** in the process,. It is very simple. Here is the definition:

```
   // -------  whichIsBiggerClone --------------
   // returns a clone of the larger Rectangle of r1 and r2
   static public Rectangle whichIsBiggerClone(
         Rectangle r1, Rectangle r2)
   {
      Rectangle returnRect = new Rectangle();
      Rectangle r;

      r = whichIsBiggerRef(r1, r2);
      returnRect.width = r.width;
      returnRect.height = r.height;
      returnRect.label = '*';  // comparison results in new rect
                               // with label '*'
      return returnRect;
   }
```

Whenever we invoke this method, we will be creating a new **Rectangle**.

Okay, we're ready to see the entire **Rectangle** example. We don't even have to discuss it because we have already done so!

Section 5 - The Rectangle Class

7A.5.1 The Program Listing

The full listing of the **Rectangle** class and client main class that tests it, follows. Notice that the main class is called **Boxcar**, so you'll have to make sure that you either change your main **.java** file to be named **Boxcar.java**, or you'll have to change the class **Boxcar** in the listing to the name of your main **.java** file.

```
// class Rectangle --------------------
class Rectangle
{
   // member data for the class -- some private
   // and some public
   private double width, height;
   public char label;

   // A couple of class methods:
   // -------  whichIsBiggerRef --------------
   // returns a reference to the larger Rectangle of r1 and r2
   static public Rectangle whichIsBiggerRef(
         Rectangle r1, Rectangle r2)
```

```
{
   if ( r1.width * r1.height > r2.width * r2.height )
      return r1;
   else
      return r2;
}

// -------  whichIsBiggerClone --------------
// returns a clone of the larger Rectangle of r1 and r2
static public Rectangle whichIsBiggerClone(
      Rectangle r1, Rectangle r2)
{
   Rectangle returnRect = new Rectangle();
   Rectangle r;

   r = whichIsBiggerRef(r1, r2);
   returnRect.width = r.width;
   returnRect.height = r.height;
   returnRect.label = '*';  // comparison results in a new rect
                            // with label '*'
   return returnRect;
}

// Now, the instance methods:
// -------  setWidthHeight Definition --------------
public boolean setWidthHeight(double width, double height)
{
   if ( width <= 0 || height <= 0 )
      return false;
   this.width = width;
   this.height = height;
   return true;
}

// -------  getWidth Definition --------------
public double getWidth()
{
   return width;
}

// -------  getHeight Definition --------------
public double getHeight()
{
   return height;
}

// -------  getArea Definition --------------
public double getArea()
{
   return width*height;
}

// -------  getPerimeter Definition --------------
public double getPerimeter()
{
   return 2*(width + height);
}
```

```
   // -------  showAllData Definition --------------
   public void showAllData()
   {
      System.out.println( "Label=" + label + " Width="
           + width + " Height=" + height );
   }

} // ----------------------- end Rectangle


//main class Boxcar --------------------
public class Boxcar {
   public static void main (String[] args)
   {
      Rectangle r1, r2, r3;

      // allocate two Rectangle objects
      r1 = new Rectangle();
      r2 = new Rectangle();

      // create a new rectangle, r1, with dimensions 5 x 8
      r1.label = 'a';
      r1.setWidthHeight(5, 8);
      r1.showAllData();
      System.out.println( "Rectangle with label " + r1.label
           + " has area " + r1.getArea()
           + " and perimeter " + r1.getPerimeter() + "\n");

      // create a new rectangle, r2, with dimensions 6 x 6
      // based on r1's dimensions:
      r2.label = 'b';
      r2.setWidthHeight( r1.getWidth()+1, r1.getHeight()-2);
      r2.showAllData();
      System.out.println( "Rectangle with label " + r2.label
           + " has area " + r2.getArea()
           + " and perimeter " + r2.getPerimeter() + "\n");

      // find out which is bigger - result (reference) is r3:
      r3 = Rectangle.whichIsBiggerRef(r1, r2);

      // at this point r3 and r1 (because r1 was bigger) point
      // to the same object.  If we changed r1, we would also be
      // changing r3.  How can you verify this? (informal hwk)
      System.out.println( "Rectangle with label " + r3.label
           + " is larger, having area " + r3.getArea() );

      // find out which is bigger - result (object) is r3:
      r3 = Rectangle.whichIsBiggerClone(r1, r2);

      // now r3 is decoupled from r1 and r2.  It is its own
      // object.  However the output looks identical to above.
      System.out.println( "Rectangle with label " + r3.label
           + " is larger, having area " + r3.getArea() );
```

```
       // finally, a demonstration of how to use the boolean return value
      if ( ! r3.setWidthHeight(-3, 40) )
         System.out.println("ERROR in setWidthHeight()!");
      else
         System.out.println("OK Return setWidthHeight()!");

      if ( ! r3.setWidthHeight(21, 40) )
         System.out.println("ERROR in setWidthHeight()!");
      else
         System.out.println("OK Return setWidthHeight()!");
   }

} // ----------------------- end Boxcar
```

7A.5.2 Remarks on Static Class Methods

To repeat, ***static class methods*** are ***not*** associated with any particular ***object*** of the ***class***, while ***instance methods*** are. ***Instance methods,*** being called through an ***object dereference*** as in:

```
   obj.instMethod();
```

automatically have access to the "***this***" reference, either explicitly, as in **this.aMember = something**, or implicitly, as in **aMember = something.** All the ***instance variables*** of the class can be tinkered with inside ***instance methods,*** and we saw that this modifies the object that was used to make the method call.

In ***static class methods***, we are simply putting some functionality of the ***class*** into it in the form of a ***method*** that can be called *without a class object.* Often, as we have done here, ***static class methods*** take ***objects*** of the class as ***arguments***, but not always. For instance, in the **Math** class, functions like **Math.random()** don't take any parameters, much less **Math** class objects (there is no such thing as a Math class object anyway!).

Static class methods are just methods that you think best belong inside your class because they pertain to aspects of that ***class*** or modify the objects' private data that might be passed in as ***arguments*** to the ***methods***.

When called from inside the class (i.e., from a **Rectangle** method), you can do so with no ***dereference*** whatsoever. This is done in **whichIsBiggerClone()**. When called from another ***class***, you would use the ***class*** name to ***dereference*** the method rather than a particular object. This is done from **main()** inside **Boxcar**.

Class methods go a little counter to the ***object-oriented*** approach because they allow functions to be called with no real object doing the calling. However, there is no way to avoid it: all languages need such ***global methods***, and by bundling lots of related class methods into a single ***class***, like the **Math** ***class*** in which all methods are **static**, it gives these ***ordinary functions*** some basic organization.

The ***static class method*** vehicle is a notational way to manufacture ordinary functions (and, as we shall see soon, ***ordinary global data***). So if you were worried that you were stuck with a language that had no ***globals***, don't worry -- Java's got 'em. Nevertheless, using anything that resembles a global is always best avoided, and I do not encourage you to abuse this functionality.

Section 6 - Knowledge Check

7A.6.1 Concept Check

We have covered a lot in the past two weeks. Before we start with the analysis of a typical OOP project solution, let's pause to make sure that you have read and understood all the requisite vocabulary. If even one of these terms or concepts is not clear, then you should go back and read the relevant page in the last two or three modules. If, after rereading the modules, the concept still is not clear, you should ask the instructor in the public forums. There we will dissect the module in question and clarify the issues for you.

At this point, you should know what these concepts mean and be able to recognize and use them in programs.

OOP Concepts from Recent Modules

1. Class
2. Object
3. Class member
4. The difference between private and public member data
5. The difference between instance members and static members
6. Instance method
7. Static method
8. Accessor
9. Mutator
10. Constructor
11. Default constructors
12. Protecting private data in mutator and constructor methods by filtering out bad values

Method Concepts from Earlier Modules

1. Function (or method) signature
2. Function (or method) return type
3. Function (or method) formal parameters
4. Local variables

Again, if you think that you will get clarity by continuing to read beyond this point without solidifying these concepts, that is not a realistic expectation. Therefore, I want to urge you to go back and reread the recent sections if you need to, so that you can truthfully say that you get all the above concepts. Ask for help if you need it.

Now that you understand these concepts, you are ready to go through the analysis of an OOP problem solution. That is the goal of the next sections.

Section 7 - Analysis of an OOP Project

7A.7.1 General Strategy: Class vs. Test Main

When we are given a project description in this course, there are two aspects to consider. The first is the ***class*** specification. The class is intended for many different users, not just you. So you must define it independently of the actual ***client*** application (i.e., independently of **main()** or the **Foothill** class) and only include data and methods useful to the wide range of potential applications. The second is the ***client***, which usually means the **main()**. The client is really only a test of the class, but I do give you requirements for its behavior, as well, so you must design the **main()** carefully and according to spec. Think of these as two distinct parts of the task.

Our approach will be to first build the ***class*** and test it in a small client (**main()**). In fact, we won't even try to build the entire class at once. We will do it in stages, completing and testing each stage before we move on to the next.

7A.7.2 The Example Project Specification

You are to define a class called **Patient**.

Patient will have the following instance members:

- **name**, a **String**
- **id**, an **int**
- **temperature**, a **double**

The **id** is meant to represent a patient ID that might be used to track a patient in an E.R. or hospital. The **temperature** is the Fahrenheit temperature that the patient presents upon admission.

The class should only allow **String** **name**s that have between **2** and **40** characters. It will only allow **id**s that are between **0** and **9999**. It will only allow **temperature**s between **88** and **111** degrees. Any other value will be considered a user error and we will not allow the client to set such values. This is true of all applications that use the class, so these limits are built-into the ***class***, not the ***client***. These limits, which are part of the class, should be ***public static*** members that are ***constants***. Also, there should be an *alarm temperature* of **103.5** (**static**) that will be used to set off an "alarm". I will describe the alarm in a moment.

You should provide the following instance methods:

- Default and parameter-taking ***constructors***.
- A ***mutator*** and ***accessor*** for each instance member.
- A **display()** method that clearly shows the patient's data. If the patient's temperature is above the alarm temperature, we will add a line *"*** urgent: attend immediately ***"*.

Provide a test main that instantiates two **Patient** objects and gets the data for each of them from the user, interactively. It then compares the **temperature**s of the two **Patient** objects and displays the **name**s of the patients, making sure that the **name** of the patient with the higher **temperature** is listed first (under the assumption that a patient with a higher **temperature** requires more immediate attention than a patient with a lower one). Test this in multiple runs and show that the patient with the higher **temperature** is always displayed first.

7A.7.3 Constructor and display() - *The Framework Phase*

The first step is to get the framework of the ***class*** and a small ***main()*** designed and debugged. We will do the minimum necessary to create a working program, then build from there. We will call this the ***Framework Phase***. Often, if you are stuck deep in the implementation of an assignment, I will ask to see your ***Framework Phase*** so save a copy of it in a separate file before you move on to the next phases.

***In the* Framework Phase *we create* only:**

- The default ***constructor***
- The **display()** method
- A ***test main()*** that instantiates two objects and displays them, without any user input

We are not doing any ***mutators***. We are not doing anything with ***static members***. One step at a time, thank you very much.

You should be able to do this small part in about 15 minutes. Try it, and after you get it working, look at my solution:

```
public class Foothill
{
   public static void main(String[] args)
   {
      // instantiate two Patients
      Patient person1, person2;
      person1 = new Patient();
      person2 = new Patient();

      // display both
      person1.display();
      person2.display();
   }
}

class Patient
{
   private String name;
   private int id;
   private double temperature;

   Patient()
   {
      name = "nobody";
      temperature = 98.6;
      id = 0;
   }

   public void display()
   {
      System.out.println(  "Patient: "
         + "\n  Name: " + name
         + "\n  ID: " + id
         + "\n  Body Temperature: " + temperature + " (F)" );
   }
}
```

```
/* ------------- RUN (OUTPUT) ----------------------
Patient:
  Name: nobody
  ID: 0
  Body Temperature: 98.6 (F)
Patient:
  Name: nobody
  ID: 0
  Body Temperature: 98.6 (F)
--------------- END OF RUN ------------------------- */
```

This was easy, but you have to actually ***do it*** to reap the benefit. Also, you can't just write it. You have to fully debug and run it.

Now we have a framework on which we can bang, cut, drill and craft.

7A.7.4 Adding the Statics and Mutators - *The Class Phase(s)*

We are going to add the rest of the class members but not do the full **main()** yet. We will still keep **main()** very simple, to test out the class and debug it if necessary.

We add the static members:

- **The limits for the string length.** The spec did not say what to call them, so we can pick our own names. If it had specified the names of these statics, we would be bound to name them as described. We'll call them **MIN_LENGTH** and **MAX_LENGTH.**
- **The limits for the id.** Same thing: we'll name this **MIN_ID** and **MAX_ID**.
- **The limits for the temperature.** You guessed it, **MIN_TEMP** and **MAX_TEMP.**
- **The alarm value**, which we will call **ALARM_TEMP.**

We also add:

- ***Accessors*** for each member
- ***Mutators*** for each member
- The ***parameter-taking constructor***

We will write a test main that does the following:

- *Instantiates* ***three*** **Patient** objects, one using the parameter-taking constructor.
- Manually *sets* patient #1 to **"Fred"**, **1234** and **100.3** and patient #2 to **"Janis"**, **5555**, **103.7.**
- *Displays* all patients.
- Calls a mutator with an *illegal* value and confirm that it returns false.
- Uses a couple *accessors* to print data from one of the objects to the screen.

This is the ***Class Phase***, and sometimes it is broken into sub-phases. For instance, if there are a lot of class methods, you might do a few mutators, get those working, add a few more, get that working, then add some more elaborate instance methods that might not be mutators or accessors. As with the ***Framework Phase***, I expect that you will have done this if you are asking for help to debug the way the ***assigned client*** interacts with the assigned class. You would not be attempting to write the ***assigned client*** if you had not first done the ***Class Phase***, with a ***simple client*** and debugged that. (Of course, you can always ask for help if you are working on the ***Class Phase*** and get stuck.)

In our current effort, if we have done everything correctly, we'll see the alarm on **Janis**, but not on **Fred**. Again, please try it yourself. It is better for you to get stuck and ask questions on *this* in the public forums rather than to wait and be overwhelmed in an assignment.

```
public class Foothill
{
   public static void main(String[] args)
   {
      // instantiate three Patients
      Patient person1, person2, person3;
      person1 = new Patient();
      person2 = new Patient();
      person3 = new Patient("Racha", 98.7, 32);

      // make changes to two
      person1.setName("Fred");
      person1.setID(1234);
      person1.setTemperature(100.5);

      person2.setName("Janis");
      person2.setID(5555);
      person2.setTemperature(103.7);

      // display all
      person1.display();
      person2.display();
      person3.display();

      System.out.println("\nMore tests ----------- :\n");

      // test a couple mutators for data filtering
      if ( !person1.setTemperature(333) )
         System.out.println( "Unable to set temperature to 333." );
      else
         System.out.println( "Temp set to 333." );

      if ( !person2.setID(-44) )
         System.out.println( "Unable to set ID to -44." );
      else
         System.out.println( "ID set to -44." );
      System.out.println();

      // test a few accessors
      System.out.println( "Patient #1's name is " + person1.getName() );
      System.out.println( "The minimum valid temperature is " +
Patient.MIN_TEMP );
   }
}
```

```
class Patient
{
   public static final int MIN_LENGTH = 2;
   public static final int MAX_LENGTH = 40;
   public static final int MIN_ID = 0;
   public static final int MAX_ID = 9999;
   public static final double MIN_TEMP = 88.;
   public static final double MAX_TEMP = 111.;
   public static final double ALARM_TEMP = 103.5;
   public static final double DEFAULT_TEMP = 98.6;
   public static final String DEFAULT_NAME = "nobody";
   public static final int DEFAULT_ID = 0;

   private String name;
   private int id;
   private double temperature;

   Patient()
   {
      name = DEFAULT_NAME;
      temperature = DEFAULT_TEMP;
      id = DEFAULT_ID;
   }

   Patient(String name, double temperature, int id)
   {
      if ( !setName(name) )
         this.name = DEFAULT_NAME;
      if ( !setTemperature(temperature) )
         this.temperature = DEFAULT_TEMP;
      if ( !setID(id) )
         this.id = DEFAULT_ID;
   }

   public void display()
   {
      System.out.println(  "Patient: "
         + "\n  Name: " + name
         + "\n  ID: " + id
         + "\n  Body Temperature: " + temperature + " (F)" );
      if (temperature > ALARM_TEMP)
         System.out.println( "*** urgent: attend immediately ***");
   }

   // Accessors can be done in line:
   double getTemperature() { return temperature; }
   int getID() { return id; }
   String getName() { return name; }
// mutators
   Boolean setTemperature(double temperature)
   {
      if (temperature < MIN_TEMP || temperature > MAX_TEMP)
         return false;
      this.temperature = temperature;
      return true;

   }
   Boolean setID(int id)
   {
      if (id < MIN_ID || id > MAX_ID)
         return false;
      this.id = id;
      return true;
   }
   Boolean setName(String name)
   {
      if (name.length() < MIN_LENGTH || name.length() > MAX_LENGTH)
         return false;
      this.name = name;
      return true;
   }
}

/* ------------- RUN (OUTPUT) ----------------------
Patient:
  Name: Fred
  ID: 1234
  Body Temperature: 100.5 (F)
Patient:
  Name: Janis
  ID: 5555
  Body Temperature: 103.7 (F)
*** urgent: attend immediately ***
Patient:
  Name: Racha
  ID: 32
  Body Temperature: 98.7 (F)

More tests ----------- :

Unable to set temperature to 333.
Unable to set ID to -44.

Patient #1's name is Fred
The minimum valid temperature is 88.0
--------------- END OF RUN ------------------------- */
```

Notice that I defined the parameter-taking ***constructor*** so that it calls the ***mutators***. Once you have ***mutators***, you should use them from that ***constructor*** so that all changes to members go through the ***mutator*** methods and nowhere else. This is more important in ***constructors*** that take parameters, and it is optional in default constructors since you may assign values directly there.

This probably took about an hour or two to write and debug. You could have broken this into steps and done only one ***mutator*** and ***accessor*** first, say, the ***mutator*** and ***accessor*** associated with the **name** member, debugged it, then went on to do the others.

Note that this **main()** did not completely test everything. Ideally, you would do a thorough check of your class methods before proceeding to the next phase. Keep a copy of the final ***Class Phase*** and client so you can show them to me if you need help with something further down stream. I'll ask to see them to help you isolate your problem.

At this point our class is about done and all we need to do is design and write our sample **main()**. Next page.

Section 8 - Completion of the OOP Project

7A.8.1 What Goes Into main()?

Any details of the output or processing that are associated with the sample main must stay out of the class, and instead be done in **main()** or in **non-Patient** methods that **main()** calls. The user interaction, for example, does not go into the **Patient** class. This would not make sense, since virtually no one but us, in our testing, will ever use these methods. Only put methods in the **Patient** class that are likely to be used by others and that pertain to the basic **Patient** data.

7A.8.2 Test main() Without the Patient Class - *The Naked Main*

We will do **main()** in two steps.

First, let's get the UI (User Interface) working.

- Ask for patient data.
- Each time we get data from the user, echo it to see if we have read it in correctly.

We'll call this the ***Naked Main*** phase, since it involves writing a main() without any of the target class equipment included.   This is only about establishing that we are getting good data from the user. This phase will look very different for every student, and I probably won't require that you show me much of this.

Designing the ***Naked Main*** for the current example is straightforward. Note that we will be mixing strings and numbers during input. The best way to do this is by using **nextLine()** for *everything* and converting the **id** and **temperature** to numeric data types as needed. We don't want to mix **nextLine()** with **nextInt() or nextDouble()**.

```
// ---------------- SOURCE ------------------------
import java.util.Scanner;

public class Foothill
{
   public static void main(String[] args)
   {
      // variables to capture input
      String userName;
      double userTemp;
      int userId;

      // get the info for patient #1:
      System.out.println(  "Patient #1 ---");

      // we built three helper methods for this
      userName = getPatientName();
      userId = getPatientID();
      userTemp = getPatientTemp();

      // testing that we got the correct input:
      System.out.println( "Patient #1: " + userName + " id: " + userId
         + " temp: " + userTemp );

      // get the info for patient #2:
      System.out.println( "Patient #2 ---") ;

      // we built three helper methods for this
      userName = getPatientName();
      userId = getPatientID();
      userTemp = getPatientTemp();

      // testing that we got the correct input:
      System.out.println(  "Patient #2: " + userName + " id: " + userId
         + " temp: " + userTemp);
   }

   // main methods (non-Patient methods)
   static String getPatientName()
   {
      // declare an object that can be used for console input
      Scanner inputStream = new Scanner(System.in);

      String stringIn;
      System.out.println( "What's the patient's name? " );
      // get the answer in the form of a string:
      stringIn = inputStream.nextLine();

      return stringIn;
   }
   static int getPatientID()
   {
      // declare an object that can be used for console input
      Scanner inputStream = new Scanner(System.in);

      int id;
      String stringIn;

      System.out.println( "What's the patient's id #? " );
      // get the answer in the form of a string, then convert to int:
      stringIn = inputStream.nextLine();
      id = Integer.parseInt(stringIn);  // convert String str to an int num
      return id;
   }

   static double getPatientTemp()
   {
      // declare an object that can be used for console input
      Scanner inputStream = new Scanner(System.in);

      double temp;
      String stringIn;

      System.out.println( "What's the patient's temperature? " );
      // get the answer in the form of a string, then convert to int:
      stringIn = inputStream.nextLine();
      temp = Double.parseDouble(stringIn);
      return temp;
   }
}

/* ------------- RUN (OUTPUT) ----------------------
Patient #1 ---
What's the patient's name?
Marsha Malone
What's the patient's id #?
111111
What's the patient's temperature?
98.5
Patient #1: Marsha Malone id: 111111 temp: 98.5
Patient #2 ---
What's the patient's name?
Frank Lewis
What's the patient's id #?
9876
What's the patient's temperature?
22.5
Patient #2: Frank Lewis id: 9876 temp: 22.5

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

Note that we haven't done any error checking. This will be handled through the ***mutators***. Right now we are only confirming that we got the correct user input. This is a very important step in interactive **main()** methods. Don't try to write lots of code without first doing a **main()** like you see above. Now we know we can trust the information we are getting from the user.

7A.8.3 Combining the Real main() with the Target Class - *Synthesis*

We will proceed slowly. The next step is to take the data we received and place it into **Patient** objects. We will delay doing the last part of logic (testing to see which patient is sicker) until we have this portion ready.

This is called the* Synthesis Phase*.

- Instantiate both patients.
- Get info for patient #1.
- Use mutators to set the data for #1.
- Do it all now for patient #2.
- Print out both patients.

```
import java.util.Scanner;

public class Foothill
{
   // declare a (rare) global that can be used by all for console input
   static Scanner inputStream = new Scanner(System.in);
   // -------  main --------------
   public static void main(String[] args)
   {
      // variables to capture input
      String userName;
      double userTemp;
      int userId;

      // Patient objects
      Patient person1 = new Patient();
      Patient person2 = new Patient();

      // get the info for patient #1:
      System.out.println("Patient #1 ---");
      // we built three helper methods for this
      userName = getPatientName();
      userId = getPatientID();
      userTemp = getPatientTemp();

      // set patient #1
      if ( !person1.setName(userName) )
         System.out.println("Error in patient name: Invalid length.");
      if ( !person1.setID(userId) )
         System.out.println("Error in patient id: out of range.");
      if ( !person1.setTemperature(userTemp) )
         System.out.println("Error in patient temperature:  out of range.");
      // get the info for patient #2:
      System.out.println("Patient #2 ---");
      userName = getPatientName();
      userId = getPatientID();
      userTemp = getPatientTemp();

      // set patient #2
      if ( !person2.setName(userName) )
         System.out.println("Error in patient name: Invalid length.");
      if ( !person2.setID(userId) )
         System.out.println("Error in patient id: out of range.");
      if ( !person2.setTemperature(userTemp) )
         System.out.println("Error in patient temperature:  out of range.");

      // display patients
      person1.display();
      person2.display();
   }

   // main methods (non-Patient methods)
   static String getPatientName()
   {
      String stringIn;

      System.out.print("What's the patient's name? ");
      stringIn = inputStream.nextLine();
      return stringIn;
   }

   static int getPatientID()
   {
      int id;
      String stringIn;

      System.out.print("What's the patient's id #? ");
      stringIn = inputStream.nextLine();
      id = Integer.parseInt(stringIn);
      return id;
   }

   static double getPatientTemp()
   {
      double temp;
      String stringIn;

      System.out.print("What's the patient's temperature? ");
      stringIn = inputStream.nextLine();
      temp = Double.parseDouble(stringIn);
      return temp;
   }
}
```

```
class Patient
{
   public static final int MIN_LENGTH = 2;
   public static final int MAX_LENGTH = 40;
   public static final int MIN_ID = 0;
   public static final int MAX_ID = 9999;
   public static final double MIN_TEMP = 88.;
   public static final double MAX_TEMP = 111.;
   public static final double ALARM_TEMP = 103.5;
   public static final double DEFAULT_TEMP = 98.6;
   public static final String DEFAULT_NAME = "nobody";
   public static final int DEFAULT_ID = 0;

   private String name;
   private int id;
   private double temperature;

   Patient()
   {
      name = DEFAULT_NAME;
      temperature = DEFAULT_TEMP;
      id = DEFAULT_ID;
   }

   Patient(String name, double temperature, int id)
   {
      if ( !setName(name) )
         this.name = DEFAULT_NAME;
      if ( !setTemperature(temperature) )
         this.temperature = DEFAULT_TEMP;
      if ( !setID(id) )
         this.id = DEFAULT_ID;
   }

   public void display()
   {
      System.out.println(  "Patient: "
         + "\n  Name: " + name
         + "\n  ID: " + id
         + "\n  Body Temperature: " + temperature + " (F)" );
      if (temperature > ALARM_TEMP)
         System.out.println( "*** urgent: attend immediately ***");
   }

   // Accessors can be done in line:
   double getTemperature() { return temperature; }
   int getID() { return id; }
   String getName() { return name; }
   // mutators
   Boolean setTemperature(double temperature)
   {
      if (temperature < MIN_TEMP || temperature > MAX_TEMP)
         return false;
      this.temperature = temperature;
      return true;

   }
   Boolean setID(int id)
   {
      if (id < MIN_ID || id > MAX_ID)
         return false;
      this.id = id;
      return true;
   }
   Boolean setName(String name)
   {
      if (name.length() < MIN_LENGTH || name.length() > MAX_LENGTH)
         return false;
      this.name = name;
      return true;
   }
}

/* ------------- RUN 1 (OUTPUT) ----------------------
Patient #1 ---
What's the patient's name? aa aa
What's the patient's id #? 111
What's the patient's temperature? 99.1
Patient #2 ---
What's the patient's name? bb bb
What's the patient's id #? 222
What's the patient's temperature? 102.5
Patient:
  Name: aa aa
  ID: 111
  Body Temperature: 99.1 (F)
Patient:
  Name: bb bb
  ID: 222
  Body Temperature: 102.5 (F)
--------------- END OF RUN 1 ------------------------- */
```

```
/* ------------- RUN 2 (OUTPUT) ----------------------
Patient #1 ---
What's the patient's name? bad bad
What's the patient's id #? 123
What's the patient's temperature? 199
Error in patient temperature:  out of range.
Patient #2 ---
What's the patient's name? hot hot
What's the patient's id #? 456
What's the patient's temperature? 104.2
Patient:
  Name: bad bad
  ID: 123
  Body Temperature: 98.6 (F)
Patient:
  Name: hot hot
  ID: 456
  Body Temperature: 104.2 (F)
*** urgent: attend immediately ***
--------------- END OF RUN 2 ------------------------- */
```

This is a big synthesis of the assigned **main()** and the ***class***. But it is not that difficult -- perhaps an hour's worth of typos or logic errors, but it should have been fun.

We ran it twice, giving bad data intentionally to see if our program dealt with it all correctly. Chances are, in a real program, you will have found lots of little bugs that had to be fixed when doing this.

Something you could have done that I didn't do: give better error messages. This would be a perfect opportunity to use the public ***static constants*** (the range limits of **id**, **name** and **temperature**) in your **main()** to produce a very precise error message. I'll leave it up to you to fix this.

7A.8.4 Finishing It Off

I am going to leave the rest to you. At this point, you only have to do some logic in your **main()** to test to see which, among the two patients, should be listed first due to higher temperatures. It can all be done directly in **main()** without any new methods.