Section 1 - Methods Defined in Classes

6B.1.1 Instance Methods.

The ***classes*** we have created so far have contained only data. ***Classes*** would not be very useful if they did not also ***encapsulate*** the ***behavior*** of that data in the form of ***methods***. To that end, ***classes*** are allowed to declare their own ***member methods***. ***Classes*** can have many different kinds of ***member methods***. Some might be meant to be called from *outside* our ***class*** (i.e., from ***methods*** of *other* ***classes***). Some might be intended only for use *inside* our class (i.e., invoked from other ***methods*** in the *same* ***class***). However, all the ***methods*** of a given ***class*** should be somehow related to the data of that ***class***.

Furthermore, we are going to do something we have not seen before. When we declare our methods in these new user-defined classes, we are going to do so without the keyword ***static*** in front of them. By omitting the word ***static*** in the ***method header***, we will be creating what are called ***instance methods***. ***Instance methods*** have benefits that our previous ***static class methods*** did not, and we will see, shortly, what these benefits are.

The first kinds of class methods we will consider are ***accessor*** and ***mutator*** methods. These are methods that our client (any part of our program outside the class) uses to manipulate our class's member data.

6B.1.2 Defining Accessor and Mutator Methods

The two class examples we have seen so far were fun, but dangerous. By this I mean that the client (the **main()** ***method*** or whoever was using the class to declare ***objects***) could access -- that is "get at" -- the ***member data*** of the class by simply ***dereferencing*** any ***field*** using the object reference.

```
garfield.petsName = "odie";
```

That is because we declared the member data using no modifier (default access), rather than declaring it using the **private** keyword. That could be bad. You have to think of it this way. You are designing your class, and another member of your team is writing the **main()** ***method*** which uses your class. This other person might not understand the issues involved in your various member data. Maybe you don't want to allow weights greater than 250 pounds because this program is for a petting zoo, and we don't want large animals sitting on the children. Or, more fundamentally, we might want all pet names to be at least three characters long (and less than 30).

Currently there is no protection. If you wrote that class and gave it to your colleague to use you would deserve every abuse he or she heaped on it because you took no protective measures. I know you don't really want that to be the case, which is why we are now going to do two things:

1. Declare all of our member data to be **private**.
2. Supply ***accessor/mutator*** methods in our class that can be used to access or modify the data

Why will supplying ***accessor/mutator methods*** help? Just watch. Here is the definition of the **Galaxy** class. Unlike the **Pet** class it not only has ***member data***, but also ***member methods***. This is the first time we have seen a **method** defined inside one of our new classes (a class that isn't a main class).

```
class Galaxy
{
   // member data
   private String name;
   private double magnitude;

   // default constructor
   Galaxy()
   {
      name = "undefined";
      magnitude = 0.0;
   }

   // accessor "set" method ------------------------------
   public boolean setMagnitude(double mag)
   {
      if (mag < -3 || mag > 30)
         return false;
      // else
      magnitude = mag;
      return true;
   }

   public boolean setName(String theName)
   {
      if (theName == null || theName.length() < 2)
         return false;
      // else we fall through
      name = theName;
      return true;
   }

   // accessor "get" methods -----------------------------
   public String getName()
   {
     return name;
   }
   public double getMagnitude()
   {
      return magnitude;
   }
}
```

We usually call the **set()** methods ***mutator*** methods, and the **get()** methods ***accessor*** methods. However, I usually use the single term ***accessor*** for both types of methods.

6B.1.3 Invoking Accessor/Mutator Methods

We now have a class called **Galaxy** which will be used by our local astronomer-programmers to do a variety of things from controlling a remote telescope to archiving newly discovered galaxies. Currently there are only two data in the **Galaxy**, the **name** and the **magnitude**. **Magnitudes** are numbers that represent the apparent brightness of the galaxy and typically have values in the range 5 to 15 (but for our purposes will always lie between -3 and 30). So we want to protect the class data from getting incorrect values. Likewise, we want to protect against invalid names being assigned to the private name member.

That's where the methods **setName()** and **setMagnitude()** come in. Notice how they take formal parameters from some unknown client and assume that the client is sloppy. It will not set the private data until after the values passed in to these methods have been checked for "reasonableness."

We could place this class in our .java file and start using it. Here is how a main class might use this **Galaxy** class:

```
public class Foothill
{
   public static void main(String[] args) throws Exception
   {
      // declare the references
      Galaxy gal1, gal2;

      // instantiate the objects
      gal1 = new Galaxy();
      gal2 = new Galaxy();

      // try to set the data
      gal1.setName("X");
      gal1.setMagnitude(100);
      gal2.setName("Stephan's Third");
      gal2.setMagnitude(13.2);

      // let's see what happened
      System.out.println("Gal #1 name: " +  gal1.getName() );
      System.out.println("Gal #1 mag: " +  gal1.getMagnitude() );
      System.out.println("Gal #2 name: " +  gal2.getName() );
      System.out.println("Gal #2 mag: " +  gal2.getMagnitude() );
   }
}
```

There are two things we need to do at once.

- Understand the mechanics of defining and using class methods.
- Understand how these methods protect the member data of the class.

We can gain this understanding by answering the following questions:

1. Why did I call one of the methods a ***constructor***, and why does it have the same name as the name of the class?
2. Why do the methods get to use the member data without ***dereferencing*** them through objects?
3. Why are some methods referred to as ***accessor*** methods, and why do they seem to be in pairs, **setNNN()** and **getNNN()**?
4. What exactly is the meaning of private and public?

In the next sections, we answer these questions.

6B.1.4 The Full Galaxy Class and Client/main() Program

Before we go on to answer these important questions, let's show the full file that contains everything, including a second constructor that we have not yet discussed (but will soon). Notice how the program is organized: the ***main class*** (***client***) is defined first and declared **public**. Next, the ***target*** class, **Galaxy** is defined (but *without* the keyword **public** in front of its name) and, finally, a sample run. Your assignments should be in the same format. However, don't forget that we have a lot to learn about classes yet, so this example will not contain *all* the essential ingredients. We'll cover the rest soon. Meanwhile, here is what we have so far. It works and it is in the correct form:

```
public class Foothill
{
   public static void main(String[] args) throws Exception
   {
      // declare the references
      Galaxy gal1, gal2;

      // instantiate the objects
      gal1 = new Galaxy();
      gal2 = new Galaxy();

      // try to set the data
      gal1.setName("X");
      gal1.setMagnitude(100);
      gal2.setName("Stephan's Third");
      gal2.setMagnitude(13.2);

      // let's see what happened
      System.out.println("Gal #1 name: " +  gal1.getName() );
      System.out.println("Gal #1 mag: " +  gal1.getMagnitude() );
      System.out.println("Gal #2 name: " +  gal2.getName() );
      System.out.println("Gal #2 mag: " +  gal2.getMagnitude() );
   }
}

class Galaxy
{
   // member data
   private String name;
   private double magnitude;

   // default constructor
   Galaxy()
   {
      name = "undefined";
      magnitude = 0.0;
   }
```

```
   // 2-parameter constructor (to be discussed)
   Galaxy(String myName, double myMag)
   {
      if (myName.length() > 2)
         name = myName;
      else
         name = "undefined";
      if (myMag >= -3 && myMag <= 30)
         magnitude = myMag;
      else
         magnitude = 0.0;
   }

   // accessor "set" method -------------------------------
   public boolean setMagnitude(double mag)
   {
      if (mag < -3 || mag > 30)
         return false;
      // else
      magnitude = mag;
      return true;
   }

   public boolean setName(String theName)
   {
      if (theName == null || theName.length() < 2)
         return false;
      // else we fall through
      name = theName;
      return true;
   }

   // accessor "get" methods ------------------------------
   public String getName()
   {
     return name;
   }
   public double getMagnitude()
   {
      return magnitude;
   }
}

/* ------------------ Paste of Run from Above Program ----------

Gal #1 name: undefined
Gal #1 mag: 0.0
Gal #2 name: Stephan's Third
Gal #2 mag: 13.2

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

Section 2 - Understanding Constructors

6B.2.1 The Class Constructor

In the **main()** method of our main class, **Foothill**, we declare two **Galaxy** ***references*** and immediately ***instantiate objects*** for them:

```
Galaxy gal1, gal2;

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

At this point, what values are stored in the private data **name** and **magnitude**? Since the client, **main()**, has not manually set these yet, it is unclear what would happen if we tried to print out the **names** and **magnitudes** of **gal1** and **gal2**.

Initializing the member data of our class is the job of the class constructor.

The *constructor* is a *method* that

- bears the same name as that of the class, and
- has no return type (it is not **void**, **int**, **double**, etc.)

The ***constructor*** method is never called explicitly but is always called automatically for you when a new ***object*** is ***instantiated***. In other words the statement:

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

causes the **Galaxy** ***constructor*** to be ***invoked***. After that, we can be sure that **gal1**'s fields have some minimum, usable data in them. If you look at the constructor, you can see the reasonable values that we use to initialize the two members.

```
// default constructor
Galaxy()
{
   name = "undefined";
   magnitude = 0.0;
}
```

Now, if the ***accessor*** methods **setName()** or **setMagnitude()** fail to set the private members **name** and **magnitude** due to poor candidates passed in by the client, these fields will at least contain their default values.

This constructor is called a ***default constructor*** because it takes no parameters. However, you can give your constructor formal parameters if you wish. For example, you might create a constructor for **Galaxy** that takes two parameters:

```
// 2-parameter constructor
Galaxy(String myName, double myMag)
{
   if (myName == null || myName.length() < 2)
      name = "undefined";
   else
      name = myName;

   if (myMag < -3 || myMag > 30)
      magnitude = 0.0;
   else
      magnitude = myMag;
}
```

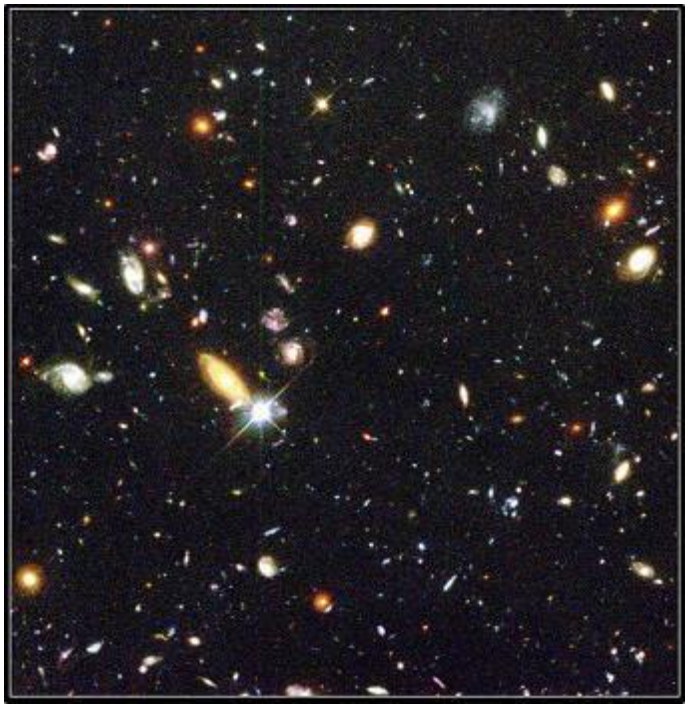
When a constructor takes parameters, it, like the mutator, should check for bad values before setting the private data. That's why we have the tests inside the constructor above. However, in a couple pages we'll see a better way for the constructor to do this. Meanwhile, the above example adequately demonstrates one way that a constructor might protect the private data from bad user-passed values. *In your homework, you will not do it this way. Instead you will use the technique demonstrated in a couple pages using mutators.*

Also, we can have both default constructors and constructors that take parameters. Just like we have seen overloaded methods in the past, there is nothing preventing us from ***overloading*** constructors (review method overloading if you forgot that concept). Multiple constructors, each taking a different number and/or types of parameters, is nothing more than method overloading applied to a constructor.

If we provide the constructor, above, that takes two parameters, we might ***instantiate*** **Galaxy** ***objects*** like so:

```
Galaxy gal1, gal2;

gal1 = new Galaxy("M31", 8.9);
gal2 = new Galaxy("Andromeda", 3.5);
```



Section 3 - Understanding Accessors

6B.3.1 Accessor/Mutator Methods Explained

Because the fields of the **Galaxy** class are declared **private**, the **main()** ***method*** cannot modify them directly. Try this from ***main()***:

```
gal1.magnitude = 13.2;
```

What happened? You can't get away with it, can you? That's what **private** does. We like that.

Instead, the client can use the ***method*** **setMagnitude()** which it *can* access, because it is declared **public**. The client (**main()**) ***dereferences*** the **setMagnitude()** method using an ***object*** of the **Galaxy** ***class***. Here are two examples:

```
gal1.setMagnitude(100);
gal2.setMagnitude(13.2);
```

If we want to change the private data of the object **gal1** from **main()** or some other client outside the class Galaxy, then we use **gal1** to ***dereference*** the **setMagnitude()** ***method*** and pass in an **int** argument - the value we want to assign. Similarly, with **gal2**.

Each of these is an attempt to set the **magnitude** of the respective **Galaxy *object*** to some value. Notice I say "attempt." Since the **setMagnitude() *mutator method*** has a built in filter to prevent out-of-range values from being assigned to the private data, there is no guarantee that either attempt will succeed. For instance, the first of these two methods calls will not cause the **magnitude** of **gal1** to change.

To see why, just look at the definition of **setMagnitude():**

```
// accessor "set" method ------------------------------
public boolean setMagnitude(double mag)
{
   if (mag < -3 || mag > 30)
      return false;
   // else
   magnitude = mag;
   return true;
}
```

The value of **100** that is passed in (when we call **gal1.setMagnitude(100)**) is not in the legal range. The ***if*** statement will evaluate to **true** (which is bad in this case) and the **return false** statement will be executed. This will cause the method to return, skipping the remaining statements.

Therefore:

1. No value is assigned to **magnitude**, and
2. The method ***returns*** a ***boolean*** value of **false** to the client (which did not, by the way, actually make use of that information -- but could have if we had designed the client differently).

If you run the program you will see that, in fact, **gal1** never has any of its private data modified because both ***accessor* set() *methods*** were called with bad (out-of-range) data. This is evident when we display the data for both **Galaxy *objects*** at the end of the program. You also notice that **gal2** *does* receive the assigned data because, for that ***object***, we passed in legal values.

This, in a nutshell, is how a ***mutator*** (or ***"setter") method*** is defined and used. The principle is very simple as long as you make sure to disallow those values that you deem to be nonsensical for the data you are representing.

Important

Your mutator methods are there to protect the private data. If you somehow don't get this, you will lose lots of points on your assignments, so please do get it. And ask if you have any doubt about it.

What about the ***accessor methods***, **getMagnitude()** and **getName()**?

These are needed because, without them, our **main()** could never print out, display or even find out what is stored in the private data **magnitude** and **name**. Again, the **private** nature of these two fields requires that we have such a method. It is standard practice to include a public ***accessor get*** method for any class field from which we want our client to be able to read.

I often use the term ***accessor*** to mean both ***mutator*** methods (set methods) and ***accessor*** methods (get methods).

Section 4 - Instance Members and Methods

6B.4.1 General Member Methods

Besides ***constructors*** and ***accessor*** methods, there are other types of ***member methods***. If these ***methods*** do not have the keyword **static** in front of them, they are called ***instance methods*** and are called through an ***object dereference***. If they do have the keyword **static** in front of their declaration, they are called ***static class method*** (or sometimes just ***class methods***) and are called through a ***dereference*** to the ***class name*** (but no special object). We will talk about static methods next time, but now let's look deeper into instance methods.

An ***instance method*** is called using an ***object*** to dereference it, as in

```
someObject.instanceMethod();
```

Once inside the definition of this ***method***, ***instanceMethod()***, we can manipulate the private fields of that particular object. A good example of an instance method is the **Galaxy *mutator* setMagnitude(),** since it is called using an ***object*** (**gal1** in this case) and is intended to modify the private data of that object (the **magnitude** field)::

```
gal1.setMagnitude(100);
```

In contrast, if we have a ***static class method***, it is called like so:

```
ClassName.classMethod();
```

Since there is no ***object*** used to call ***classMethod()***, we cannot, inside the definition of ***classMethod(),*** refer to any of the class's fields. That makes sense because there is no object lying around to which that instance member data could belong. A good example of a static class method of a Java-defined class is **Math.pow()** because it does not require any objects to dereference the call. It merely uses the ***class name***, **Math**. Although we can't see the definition of **Math** class, if we could, we would see that **pow()** was declared with the modifier **static** in front of it.

This is a little dizzying, I know.

Let me try to summarize:

- ***Instance member data*** (or instance fields), are created every time a new object is instantiated. These instance members can be manipulated by ***Instance methods*** of the same class. When an ***instance method*** manipulates ***instance data*** inside its definition, it is manipulating the data of the ***object*** that called it.
- ***Static member data*** (or static fields) are shared by all objects of the class. Similarly, static class methods are called using the class name and no particular object. ***Static methods*** *may not* manipulate **instance data** in their definitions but they can manipulate ***static member data***.

6B.4.2 Accessing Member Data With and Without Object References

In the **Pet** and **Employee** examples from last week we declared all the data using default access (which for Java is the same as **public** if all of your classes are in the same file). This allowed **main()** to ***dereference*** the member data, directly:

```
birkoff.age = 26;
```

or

```
mikesDog.petsName = "Jerry";
```

We could do that last week because we made all member data **public**. This week, however (and from now on!) we are declaring all the member data to be **private**, so **main()** cannot get to the data directly. Instead it must use objects to dereference mutator methods like **setAge()** or **setPetsName()** . Further up you saw me do that with **gal1.setMagnitude(100)**.

In all cases, though, **main()** needs to use *some* object to make the change. Whether the data was **public** (and we were setting the data manually from **main()**) or it was **private** (and we had to use a ***mutator***). Either way, we needed a specific object to do something. This makes sense, since we need to know *which* **Employee's age** we are talking about or *which* **Galaxy**'s **magnitude** we want to set. We obviously need a specific **Employee** or **Galaxy** ***object*** in our hand: **birkoff**.age = ..., or **gal1**.setMagnitude(...).

So far, so good. But in the **Galaxy** example, we notice something odd. Look at the definition of any of our ***methods*** inside the **Galaxy** class. Take the **getName()** ***method*** as an example:

```
public String getName()
{
   return name;
}
```

Sure, we are returning the value stored in the **name** field, but ...which **name**??!! **gal1's name**? **gal2's name**? How do we know which one to use? Another example is in the ***constructor***:

```
Galaxy()
{
   name = "undefined";
   magnitude = 0.0;
}
```

To which **name** and **magnitude** are we referring in there?

Stay calm, stand back, look at the big picture, and all will be revealed.

Here are the clues:

- These ***methods*** are *not* declared using the keyword **static** in front of them, which means they are ***instance methods***, not ***static class methods***.
- While ***static class methods*** are invoked using the ***class name*** to ***dereference*** them (i.e., **Math.pow()**), ***instance methods*** require an ***object*** -- an actual, individual, instance of the class -- to invoke them (i.e., **gal2.setMagnitude(13.5)**;
- Since we cannot call any ***instance method***, including a ***constructor***, unless our client is using a specific ***object***, then it is *that object's member data* to which all un-dereferenced fields, inside the method definitions, refer.

So in **getName( )**, the **name** referred to inside the ***method definition*** is the **name** of whatever object did the calling (i.e., whatever object was used to ***dereference*** the method call).

Same with the ***constructor***. We would only get inside the ***constructor*** method if we were sent there by the client ***instantiating*** a particular ***object***. When the client ***instantiates*** **gal1**, then it is **gal1's** data that is initialized inside the ***constructor***. When **gal2** is being instantiated, **gal2's** data is initialized.

6B.4.3 Instance Methods Calling Instance Methods

Here is a dumb little program, but one that makes a point.

```
// class Employee --------------------
class Employee
{
   // member data for the class -- no keyword
   // static in front, so these are called
   // "instance members"

   private int socSec;
   private int age;

   // -------  clearSocSec Definition --------------
   private void clearSocSec()
   {
       socSec = 000000000;
   }

   // -------  setAge Definition --------------
   public boolean setAge(int a)
   {
      if (a < 0 )
         // don't allow negative age
         return false;
      else
      {
         age = a;
         return true;
      }
   }

   // -------  setSS Definition --------------
   public boolean setSS(int ss)
   {
      if (ss < 0 )
         // don't allow negative ss
         return false;
      else
      {
         socSec = ss;
         return true;
      }
   }

   // ------- Employee::getOlder -------
   public boolean getOlder()
   {
      if (age++ > 147 )
      {
         clearSocSec();
         return false;
      }
      else
         return true;
   }

} // ----------------------- end Employee
```

```
// main class Company --------------------
public class Foothill
{
   public static void main (String[] args)
   {
      Employee walter;
      int k;

      walter = new Employee();

      walter.setSS(123456789);
      walter.setAge(140);

      for (k = 0; k < 14; k++)
         if ( walter.getOlder() )
            System.out.println( "Happy Birthday!" );
         else
            System.out.println( "You must be dead.  I have given"
               + " your social security number"
               + " to someone else.");
   }
}
```

Notice that this is similar to the first **Employee** example we had, but

1. now the data inside the **Employee *class*** is **private**, and
2. there are two **public *methods*** dedicated to setting this private data, as well as a **private *instance method***.

Notice that one of these ***methods*** calls another, and it does so without using an ***object*** to ***dereference*** the called ***method***. That is, inside **getOlder()** we see a call to **clearSocSec()** without the expected **obj.clearSocSec()**. Not only that, but **clearSocSec()** is ***private***, so we wonder if it can be called at all! Confused?

I'll try to straighten you out.

The reason that the call **clearSocSec()** inside the function **getOlder()** doesn't need to be ***dereferenced*** by an object was actually answered earlier. Since some ***object*** called **getOlder()** originally, there *already* is an ***object*** which owns all of the unattached data members referenced inside the ***method***. This goes for ***member method calls*** inside the ***method*** as well.

Note:

From an instance method, any further calls to other instance methods are understood to be owned by the original object.

Thus, **walter.getOlder()** inside **main()** will send control to **getOlder().** From there, the unqualified call to **clearSocSec()** is understood to also come through the ***object* walter**. We say that we are accessing the ***object*** called ***this*** (the implicit reference to the ***calling object***). Now, the whole concept of one member function calling another member function should make sense. The

original call establishes the ***"this"*** object through which all lower level calls (if any) are to be made. Of course once the original call returns back to **main()**, we would need another object to get into an ***instance method.***

As for the fact that **clearSocSec()** is ***private***, and **getOlder()** is able to call it -- well that, again, is what member functions do. They can not only change private data, but they can call private methods. There may be methods that we do not want our client to be able to call -- they are convenience functions for our other instance methods. We make those methods ***private*** by using the keyword of the same name: **private**.

6B.4.4 Public, Private and Default

Finally, we wrap up our questions with the explanation of the ***access modifiers,*** **public** and **private.**

- If **public** is used to modify a ***class member*** or ***method,*** then that ***member*** or ***method*** can be ***dereferenced*** from anywhere in the program.
- If **private** is used to modify a ***class member*** or ***method,*** then that ***member*** or ***method*** cannot be ***dereferenced*** from anywhere in the program, *except other methods of the same class.*
- If no keyword is used to modify a ***class member*** or ***method,*** then that ***member*** or ***method*** is said to have default access and can be ***dereferenced*** from anywhere in the same .java file (or package) but cannot be accessed from methods outside that .java file (or package). Since we haven't really studied packages yet, just think of this as a file issue for now.

The fact that all ***member data,*** including **private** member data, can be accessed inside ***methods*** of the same class, is important. We have seen this already in our ***class methods*** above. The ***accessor methods*** had no problem accessing the name or magnitude data of the **Galaxy** ***class*** even though those data were **private**. That's because those ***methods*** are, themselves, part of the same ***class***.

Section 5 - Essential Ingredients in All Class Designs

You now know what a ***class*** is, what ***instantiating an object*** of the class means, what ***private member data*** means, what ***public instance methods*** are, what ***mutators***, ***accessors*** and ***constructors*** are.

Reality Check

If you don't, then you need to stop reading and go back a few pages or ask a question in the forums. You will not understand this section until you first have that vocabulary mastered, which we just covered.

6B.5.1 Essential Idea #1: Constructors with Parameters Should Call Mutators

Let's take another look at the **Galaxy *constructor*** that takes two parameters:

```
// 2-parameter constructor
Galaxy(String myName, double myMag)
{
   if (myName == null || myName.length() < 2)
      name = "undefined";
   else
      name = myName;

   if (myMag < -3 || myMag > 30)
      magnitude = 0.0;
   else
      magnitude = myMag;
}
```

Also, in this same class, we've seen the mutators **setMagnitude()** and **setName():**

```
// accessor "set" method ------------------------------
public boolean setMagnitude(double mag)
{
   if (mag < -3 || mag > 30)
      return false;
   // else
   magnitude = mag;
   return true;
}

public boolean setName(String theName)
{
   if (theName == null || theName.length() < 2)
      return false;
   // else we fall through
   name = theName;
   return true;
}
```

Do you notice that we are duplicating code? This is bad. We can make the code more efficient and less prone to error if we make use of the ***mutator*** from the ***constructor***. The first attempt at doing this is not going to be correct, but it is a good guess:

```
// 2-parameter constructor
Galaxy(String myName, double myMag)
{
   setName(myName);
   setMagnitude(myMag);
}
```

What's wrong with this? It looks pretty good, right? Instead of doing the testing manually, it calls the fellow instance methods, **setName()** and **setMagnitude()** which will take the client's data (passed in through **myName** and **myMag**) and passes them along to these mutators which will filter out any bad values. This is *almost* correct. The problem is that if values passed in are bad, then the mutator will not only reject them, but will not assign anything to the private data. So if **myMag = -234**, then the call **setMagnitude(myMag)** will fail, return **false** (which we are not testing as you see) and leave **magnitude** undefined! Constructors should never leave member data undefined.

If you think that the way to fix this is by forcing the ***mutators*** to always set private data, even if bad values are passed in, then you need to think a little further. I'll show you a better solution that allows the ***mutators*** to (correctly) not make any assignment if a bad value is passed in:

```
// 2-parameter constructor
Galaxy(String myName, double myMag)
{
   if ( !setName(myName) )
      name = "undefined";
   if ( !setMagnitude(myMag) )
      magnitude = 0.0;
}
```

Aha! We are using, rather than throwing away, the return values of the mutators. If they are true, no problem. If they are false, we manually put a default value in the member(s). Problem solved.

Note that for default constructors, there's no need to call the mutators because you, the class designer, have full control over the default value you assign. There is no point in testing something for being within-range if you know what the value is ("undefined" and 0.0). So we leave the default constructor as it is.

6B.5.2 Essential Idea #2: Class Methods Must Protect Private Data and Not Rely on Clients

Here is a common error that beginning programmers make:

A client (e.g., **main()**) method call:

```
if ( userName != null && userName.length() < 2 )
   gal1.setName( userName );
```

The definition of **setName()** *WRONG*:

```
boolean setName(String theName)
{
   name = theName;
   return true;
}
```

The beginner says "I am protecting my data in the client so I don't have to worry about it in the mutator."

Here is the concept I want you to learn today:

Key Concept

As class programmers, we are designing our classes for a thousand other clients that we have never seen, nor will we ever see. Our test client is only one sample main(). There will be thousands others to come.

Therefore, there is no way we can assume anything about our client. If the client tests for bad input, that's great. But what if it doesn't? If our class can crash a program because an evil client is passing it bad data, it is not the client's fault, it is *our* fault as class designers. Our classes should never fail. We must always test every instance method which takes client parameters for bad data before we do anything.

6B.5.3 Essential Idea #3: Any Constant that Relates to the Whole Class Should be Given a Symbolic Name

Literals like **"undefined"** and **0.0** create maintenance problems. Every time we change them in one place, we have to change them everywhere. This can be avoided by making such values ***static finals*** in our class. Let's do that now, improving further our **Galaxy** class.

```
class Galaxy
{
  ...

   // static constants
   public static final double DEFAULT_MAG = 0.0;
   public static final String DEFAULT_NAME = "undefined";

   ...

   // default constructor
   Galaxy()
   {
      name = DEFAULT_NAME;
      magnitude = DEFAULT_MAG;
   }

   // 2-parameter constructor
   Galaxy(String myName, double myMag)
   {
      if ( !setName(myName) )
         name = DEFAULT_NAME;
      if ( !setMagnitude(myMag) )
         magnitude = DEFAULT_MAG;
   }

   ...
}
```

With this improvement, we have one place where we would change **DEFAULT_NAME** or **DEFAULT_MAG**, and when done, all references later in the class are automatically fresh. If there is even one literal (like **"undefined"** and **0.0**) used where these symbolic names could have been used, the entire class is broken (and points will be deducted). Unless you use the symbolic names everywhere in the class, they will not serve their purpose.

Caution

A loop counter or other trivial helper variable for a method *should not be defined as a class member* like this. The symbolic constants described above are values which are important to the whole class, independent of implementation (a ***minimum magnitude*** has meaning even if we are not programming anything). But a loop counter or other helper variable only has meaning as a local variable for the method in which it is used. Even if you use the same variable name in multiple methods, do not make such variables members. Furthermore, it's especially important not to make such variables class members for the purpose of communicating their values between methods. That violates the rule about not using "globals." In this section, I am only referring to innate class constants.

Here is the entire program, with all improvements. This would get full credit if submitted as an assignment.

```
public class Foothill
{
   public static void main(String[] args) throws Exception
   {
      // declare the references
      Galaxy gal1, gal2;

      // instantiate the objects
      gal1 = new Galaxy();
      gal2 = new Galaxy();

      // try to set the data
      gal1.setName("X");
      gal1.setMagnitude(100);
      gal2.setName("Stephan's Third");
      gal2.setMagnitude(13.2);

      // let's see what happened
      System.out.println("Gal #1 name: " +  gal1.getName() );
      System.out.println("Gal #1 mag: " +  gal1.getMagnitude() );
      System.out.println("Gal #2 name: " +  gal2.getName() );
      System.out.println("Gal #2 mag: " +  gal2.getMagnitude() );
   }
}

class Galaxy
{
   // member data
   private String name;
   private double magnitude;

   // static constants
   public static final double DEFAULT_MAG = 0.0;
   public static final String DEFAULT_NAME = "undefined";
   public static final double MIN_MAG = -3.;
   public static final double MAX_MAG = 30.;
   public static final int MIN_STR_LEN = 2;
```

```
   // default constructor
   Galaxy()
   {
      name = DEFAULT_NAME;
      magnitude = DEFAULT_MAG;
   }

   // 2-parameter constructor
   Galaxy(String myName, double myMag)
   {
      if ( !setName(myName) )
         name = DEFAULT_NAME;
      if ( !setMagnitude(myMag) )
         magnitude = DEFAULT_MAG;
   }

   // accessor "set" method ------------------------------
   public boolean setMagnitude(double mag)
   {
      if (mag < MIN_MAG || mag > MAX_MAG)
         return false;
      // else
      magnitude = mag;
      return true;
   }

   public boolean setName(String theName)
   {
      if (theName == null || theName.length() < MIN_STR_LEN)
         return false;
      // else we fall through
      name = theName;
      return true;
   }

   // accessor "get" methods -----------------------------
   public String getName()
   {
     return name;
   }
   public double getMagnitude()
   {
      return magnitude;
   }
}

/* ------------------ Paste of Run from Above Program ----------

Gal #1 name: undefined
Gal #1 mag: 0.0
Gal #2 name: Stephan's Third
Gal #2 mag: 13.2

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

You will probably see examples in the modules that follow which violate this rule. Call any such instances to my attention in the ***typos forum*** so I can fix them. The original modules were written under some time pressure, and I did not fully vet all the examples. My apology in advance.

Section 6 - Inner Classes and the String Class

This section contains material that you may not use immediately. You can make a note to come back to it as you encounter **Strings** and inner classes in future reading. This is a good time to see these topics, however, so I present them to you now.

However, everyone is required to read the usual Prevent Point Loss paragraph at the bottom of this page.

6B.6.1 Inner Classes

Thus far all of our classes have been siblings to one another. That is, we complete the definition of one class, before we begin the definition of another one. On occasion, we may want to create a new class inside another class, that is, start the new class definition before we have reached the closing brace of the previous class definition. If one class is defined completely within another class, it is called an ***inner class***. The inner class can be used in the methods of the outer (i.e. containing) class. If it is declared **public**, it can also be used from client classes.

Inner classes are of very limited use, and their main application is in something having to do with GUI programming called ***event listeners***. We will come to event driven programming and these inner listener classes in a moment, but first let's see an example of an inner class in a simple, non-GUI program.

6B.6.2 A Contact List with an Inner Phone Class

Our example uses a class called **Contact** which is used to instantiate client contact objects consisting of:

- a name
- a phone number

The **Contact** class serves as an ***outer class*** for an ***inner class*** called **Phone**. **Phone** is used by the **Contact** class to sanitize and format the phone number **Strings** for the outer **Contact** class.

In this example we use ***constructors that take parameters***. We also make use of the **Character** class method **isDigit()**. The inner class **Phone** constructs new sanitized **Strings** from the messy **Strings** that are passed into the ***constructors*** by the client. The new, properly formatted, **Strings** are built up inside the **Phone** ***constructor*** using ***for loops*** and ***concatenation***. In other

words, while the client may supply a sloppy string such as "650 - 915-3 06-1", the Phone constructor will convert and store this number internally as "6509153061". Then, when asked to convert this **String** to a human-readable phone number for output, the class enlists an instance method called **toString(),** which we define below. The format of phone numbers, after preparation by toString is "(650)916-3061". The class also handles numbers without area codes and badly formed numbers.

Please read this example not only for its demonstration of inner classes, but also as a study in string processing, constructors, for loop logic, and use of local variables.

```
public class Foothill
{
   public static void main (String[] args) throws Exception
   {
      Contact client1
            = new Contact("Evariste Galois", "650-949-7000");
      Contact client2
            = new Contact("Michael Faraday", "949-1234");
      Contact client3
            = new Contact("Clerk Maxwell", "(213)555123");

      System.out.println("Dial " + client1.getPhone()
            + " to reach " + client1.getName());
      System.out.println("Dial " + client2.getPhone()
            + " to reach " + client2.getName());
      System.out.println("Dial " + client3.getPhone()
            + " to reach " + client3.getName());
   }
}

// class Contact
class Contact
{
   private String name;
   private Phone phone;

   // static constants
   static final int VALID_NUM_LEN_SHORT = 7;
   static final int VALID_NUM_LEN_LONG = 10;
   static final String INVALID_STRING_MSG = "(invalid phone number)";
   static final int POS_OF_DASH_IN_PHONENUM = 3;
   static final int POS_OF_RT_PAREN_IN_AREA_CODE = 3;
   static final int START_OF_LASTFOUR_IN_PHONENUM = 4;

   public Contact(String inName, String inPhone)
   {
      name = new String(inName);
      phone = new Phone(inPhone);
   }

   public String getName()
   {
      return name;
   }
```

```
   public String getPhone()
   {
      return phone.toString();
   }

   // inner class Phone used by class Contact only
   private class Phone
   {
      String number;

      Phone(String inNum)
      {
         char nextDigit;
         int k;

         // store only digits "213-555-1212" becomes "2135551212"
         for (k = 0, number = ""; k < inNum.length(); k++)
         {
            nextDigit = inNum.charAt(k);
            if (Character.isDigit(nextDigit))
               number = number + nextDigit;
         }
      }

      // returns properly formed number (AAA)PPP-FFFF or PPP-FFFF
      public String toString()
      {
         int nextDigit, j;
         String retStr;

         if (number.length() != VALID_NUM_LEN_SHORT
               && number.length() != VALID_NUM_LEN_LONG)
            return INVALID_STRING_MSG;

         retStr = "";
         // if full 10 digit number, use parens for area code
         nextDigit = 0;
         if ( number.length() == VALID_NUM_LEN_LONG)
         {
            retStr += "(";
            for ( j = 0; j < POS_OF_RT_PAREN_IN_AREA_CODE; j++, nextDigit++ )
               retStr += number.charAt(nextDigit);
            retStr += ")";
         }
         // either way, hyphenate 7 digit number (don't reset nextDigit)
         for ( j = 0; j < POS_OF_DASH_IN_PHONENUM; j++, nextDigit++ )
            retStr += number.charAt(nextDigit);
         retStr += "-";
         for (j = 0; j < START_OF_LASTFOUR_IN_PHONENUM; j++, nextDigit++)
            retStr += number.charAt(nextDigit);

         return retStr;
      }
   }
}
```

The run is as follows:

```
Dial (650)949-7000 to reach Evariste Galois
Dial 949-1234 to reach Michael Faraday
Dial (invalid phone number) to reach Clerk Maxwell
```

6B.6.3 String Instance Methods

Once you have a **String** object, you can use it to dereference a large number of useful instance methods in the String class. The Java link in the resources for Week 2A reading takes you to the String class page where you can see descriptions of all of these. Here are some useful **String** methods.

- boolean string1.**equals**(String string2) : returns **true** if string1 and string2 store identical strings (even though they may be referencing different objects). It returns **false**, otherwise.
- boolean string1.**equalsIgnoreCase**(String string2): same as **equals()** but ignores difference in capitalization.
- int string1.**compareTo**(String string2): returns an **int** which is < 0 if string1 is before string2, > 0 if string1 is after string2, and 0 if string1 is identical to string2. This method uses normal alphabetical ordering.
- int string1.**compareToIgnoreCase**(String string2): same as **compareTo()** but ignores difference in capitalization.
- int string1.**indexOf**(String string2): Returns the index (position) within string1 of the *first* occurrence of string2 or -1 if string2 does not appear in string1.
- int string1.**indexOf**(String string2, int from_index): Same as **indexOf(string2)** but takes an **int** as a second parameter, representing the position in string1 to begin the search. (Useful for skipping known occurrences of string2 in string1).
- int string1.**lastIndexOf**(String string2): Returns the index within string1 of the *last* occurrence of string2, or -1 if string2 does not appear in string1.
- String[] string1.**split**(String split_token, int limit): Returns an ***array of* Strings** which represent pieces of the original string1, each split using the split_token as a separator. For instance, if the split token is a space, " ", then the array that **split()** returns will be the individual words in string1.
- String string1.**toUpperCase**(): returns a **String** based on string1 but with all chars up-cased.
- String string1.**toLowerCase**(): returns a **String** based on string1 but with all chars down-cased.
- String string1.**substring**(int beginning_index, int ending_index): Returns a new **String** that represents the portion of string1 starting at position beginning_index and ending at position ending_index.
- int string1.**length()**: returns the length of string1. This is an old friend that we've been using all along.

There are dozens other **String** methods and they are all very useful, so I'll let you explore them at your leisure.

Prevent Point Loss

- **All mutators return boolean.** Set methods or functions should return a boolean value, whether your client uses the return value or not. (1 - 2 point penalty)
- **Filter input.** Any mutator, constructor, or other member method (function) that uses a passed parameter as a basis for setting private data, should test for the validity of that passed parameter - that means range checking, string length testing, or any other test that makes sense for the class and private member. (2+ point penalty)
- **Call mutators from constructors that take parameters.** When a constructor takes parameters, send those parameters to the mutators for testing rather than testing directly. This avoids code duplication. (1 point penalty)
- **Do not declare loop counters or other helper variables as class members.** Most variables should be declared locally in their respective methods. Only data that is innate to the meaning of the class should be member data.
- **Use symbolic constants rather than literals.** If the value of some default or min/max for your class has a true meaning for the class (i.e., it is not about implementation but means something even before we start programming), we must make it a static constant, usually public, for potential use by our clients. (1.5 point penalty)