

---

## Chapter 5

# Proper Coding Style Using MVC

---

### What is in This Chapter ?

In this chapter, we will discuss proper coding style. We begin with an explanation of the **Model/View/Controller** paradigm and then show how these can be incorporated together cleanly with a modular coding style. We discuss in detail how we can perform **encapsulation** to protect and simplify the **model** classes that we use. We will then discuss how the **view** and **controller** code of the user interface can be separated cleanly. As a result, this chapter gives a template that you should follow for all of your Graphical User Interface applications.



## 5.1 Separate Model, View and Controller Components

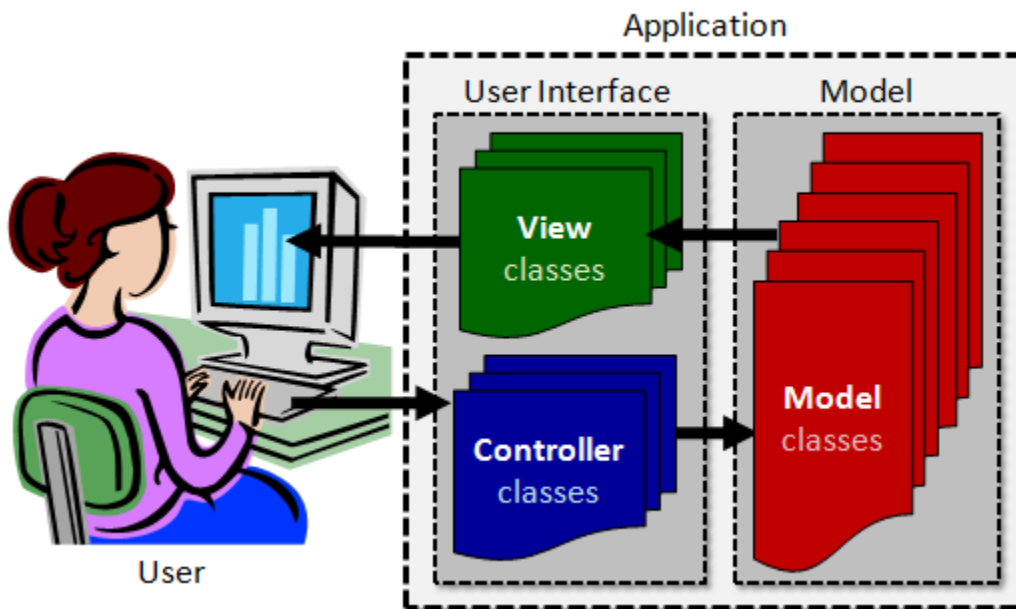
In the previous chapter, we discussed the difference between **model** classes and those classes that are part of the **user interface**. The model classes deal with the *business logic* aspects of the application and the user interface is the "front end" which allows the user to interact with the model classes.

We can further split the user interface classes into two portions called the **view** and the **controller**.

*The **view** displays the necessary information from the model into a form suitable for interaction, typically a user interface element.*

*The **controller** accepts input from the user, modifies the model accordingly.*

The idea is depicted in the picture below. The user sees the **view** of the application and then interacts with the **controller**. Such interaction usually results in the **model** being modified in some way. Then these **model** changes are reflected back to the **view** of the user interface and the user often gets visual feedback that the **model** has changed:



So far, in our examples in the last chapter, we did not have a really useful **model** and the notion of a **view** and a **controller** was not identifiable as all our GUI code was lumped together into one **JFrame** class, with perhaps an extra **JPanel** class.

Now we will discuss the "proper" way of splitting up the **model**, the **view** and the **controller** into a nice & clean modular style that allows us to modify and replace any of the three components cleanly. This arrangement represents what is called the **MVC** software architecture and sometimes referred to as a software **design pattern**. There are 3 main advantages of using the MVC architecture:

1. it decouples the models and views,
2. it reduces the complexity of the overall architectural design and
3. it increases flexibility and maintainability of code.

There are many ways to implement the MVC architecture in your programs. In this course we will consider just one specific way of writing the code. In industry, however, you will see various other ways of implementing the same architecture.

In order to create a well-structured, stable, reliable and maintainable application ... it is necessary to have a properly working model that is designed nicely so that the user interface can connect to it in a simple and safe way. In the next section, we will discuss what is necessary to create this "proper" kind of model.

## 5.2 Developing a *Proper* Model

When creating and defining an object it is a good idea to keep it simple so that anybody who uses that object in the future (including yourself) can remember how to use it. Often, there are details about an object that we don't need to know about in order to use the object. For example, when we drive a car, we need to know simple things such as:

- starting/ stopping
- steering
- changing gears
- braking, etc..



However, we do not need to worry about things such as:

- assembling the carburetor
- adjusting the spark plug timing
- installing gas lines
- changing the muffler, etc..



Cars are clearly designed to be easy to drive, requiring a simple and easy-to-understand interface. Similarly, it is important that we make our code easy to use and easy to understand. Otherwise, making changes to the code, debugging it and extending it with new features can quickly become very difficult and time consuming.

In order to keep our code simple, we need to make the interface (or "outside view") of our objects as simple as possible. That means, we need to **"hide the details"** of our object that most people would not need to worry about. That is, we need to hide some of the attributes (complicated parts) and methods (complicated procedures) for our object "under the hood", so to speak.





In addition to simplicity, there is another reason to hide some of the details of our object. We would like to prevent outsiders from "messing around with" the inner details of an object. For example we lock our car doors and trunk so that people don't get in there and take things away or damage them etc.. Similarly, for example, if we allow anyone to access the attributes of our object and perform behaviors on it in the wrong order, then this could lead to corrupt data and/or various types of errors in our code.

The idea of hiding the unnecessary details of an object and protecting inner parts of that object from general users is called *encapsulation*:

**Encapsulation** involves enclosing an object with a kind of "protective bubble" so that it cannot be accessed or modified without proper permission.



In JAVA, we protect and hide attributes and behaviors by using something called an **access modifier**.

An **access modifier** is a permission setting for our attributes and methods so that they will be visible/modifiable/usable from some places in our code but not from other places.

Access modifiers are like access levels in a high security building (e.g., no access, level 1 access, level 2 access, etc..)



By using access modifiers properly, when working with a team of software developers on a large program, some developers will have the freedom to access or modify attributes or methods from various objects, while others will not be allowed such freedom to view or change portions the objects as they would like to.

## Hiding Complex Behavior

We have already been using an access modifier called **public** when we wrote our classes, constructors and various methods:

```
public class Person { ... }
public Person(String firstName, ...) { ... }
public static void main(String[] args) { ... }
public int computeDiscount() { ... }
public void deposit() { ... }
```

The keyword **public** at the front of a method declaration means that the method is publicly available to everyone, so that these methods may be called from anywhere.

For most classes, constructors and methods, we do not need to write **public**. If we leave off this access modifier, then the class/constructor/method will have what is known as **default access** ... meaning that the methods may be called from any code that is in the same package or folder that this method's class is saved into. If we write all of our code in the same folder, then **default** and **public** access means the same thing.

There are two other access modifier options available called **private** and **protected**. When we declare a method as **private**, we would not be able to use this method from any class other than the class in which it is defined. **Protected** methods are methods that may be called from the method's own class or from one of its subclasses. So here is a summary of the access modifiers for methods:

- none - can be called from any class in the same folder
- **public** - can be called from anywhere
- **private** - can only be called from this class
- **protected** - can be called from this class or any subclasses

In this course, most of the methods that we write are **public** methods which allows the most freedom to access and modify our objects. Usually, **private** methods are known as **helper methods** since they are often created for the purpose of helping to reduce the size of a larger **public** method or when a piece of code is shared by several methods.

For example, consider bringing in your car for repair. The publicly available method would be called to **repair()** the car. However, many smaller sub-routines are performed as part of the repair process (e.g., **runDiagnostics()**, **disassembleEngine()** etc...). From the point of view of the user of the class, there is no need to understand the inner workings of the repair process. The user of the class may simply need to know that the car can be repaired, regardless of how it is done. Here is an example of breaking up the repair problem into *helper* methods that do the sub-routines as part of the repair ...

```
public class Car {  
    public void repair() {  
        this.runDiagnostics();  
        this.disassembleEngine();  
        this.repairBrokenParts();  
        this.reassembleEngine();  
        this.runDiagnostics();  
    }  
    private void runDiagnostics() { /*...*/ }  
    private void disassembleEngine() { /*...*/ }  
    private void repairBrokenParts() { /*...*/ }  
    private void reassembleEngine() { /*...*/ }  
}
```

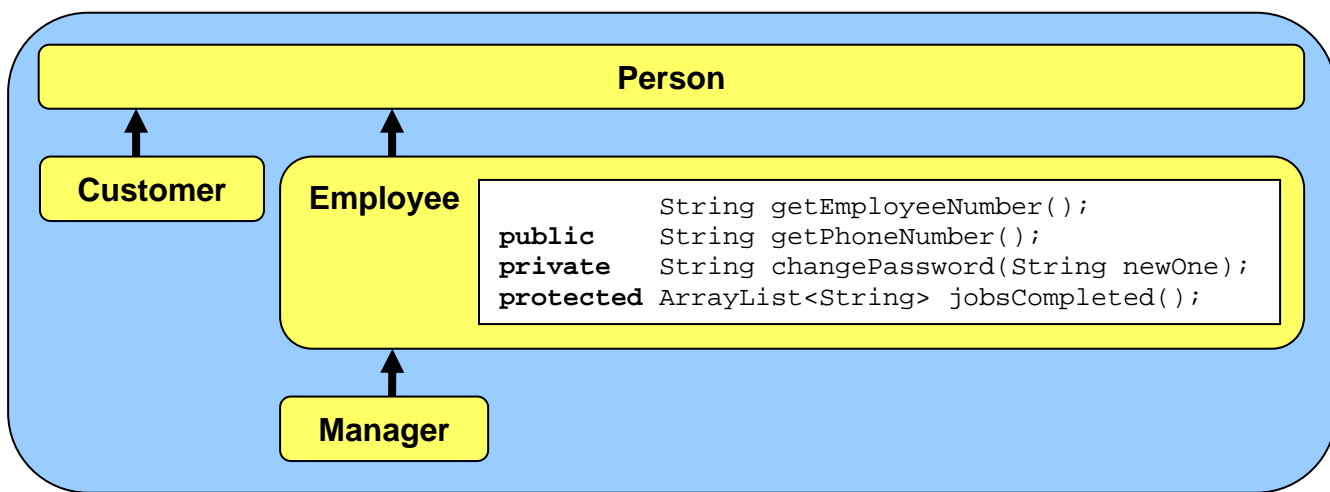
Notice that the helper methods are **private** since users of this class probably do not need to call them. Here is an example showing how we might *attempt* to call these methods from some other class:

```

public class SomeCarApplicationProgram {
    public static void main(String[] args) {
        Car c = new Car();
        c.repair();           // OK to call this method
        c.disassembleEngine(); // Won't compile, since it is private
        c.repairBrokenParts(); // Won't compile, since it is private
    }
}

```

Now, to understand the **protected** modifier, we need to consider a class hierarchy. Recall the **Person/Employee/Manager/Customer** example. Consider four methods within the **Employee** class with various access modifiers as follows:



Now consider some code within the **Manager** class that attempts to access these methods:

```

public class Manager extends Employee {
    public void tryThingsOut() {
        System.out.println(this.getEmployeeNumber()); // access allowed
        System.out.println(this.getPhoneNumber());    // access allowed
        System.out.println(this.changePassword("12345678")); // compile error
        System.out.println(this.jobsCompleted());      // access allowed
    }
}

```

Notice that the only method not allowed to be accessed is the **private** method, since the **tryThingsOut()** method is written in the **Manager** class, not in **Employee**.

Consider now the **Customer** class restrictions:

```

public class Customer extends Person {
    public void buyFrom(Employee emp) {
        System.out.println(emp.getEmployeeNumber()); // access allowed
        System.out.println(emp.getPhoneNumber()); // access allowed
        System.out.println(emp.changePassword("12345678")); // compile error
        System.out.println(emp.jobsCompleted()); // compile error
    }
}

```

Now we can no longer call the **jobsCompleted()** method, since it has been declared **protected** and **Customer** is not a subclass of **Employee**.

There really is not much more to the access modifiers when it comes to methods. However, there is one more protective keyword that can be used with methods. We can declare a method as **final** to prevent subclasses from modifying the behavior. That is, when we declare a method as being final, JAVA prevents anyone from *overriding* that method. Hence no subclasses can have a method with that same name and signature:

```

public final void withdraw(float amount) {
    ...
}

```

Why would we want to do this ? Perhaps the behavior defined in the method is very critical and overriding this behavior "improperly" may cause problems with the rest of the program.

## Hiding and Protecting Object Attributes

Now what about protecting an object's attributes ? Well, the **public/private/protected** and *default* modifiers all work the same way as with behaviors. When used on instance variables, it allows others to be able to access/modify them according to the specified restrictions.

So far, we have never specified any modifiers for our attributes, allowing them all *default* access from classes within the same package or folder.

However, in real world situations, it is often best NOT to allow outside users to modify the internal private parts of your object. The reason is that results can often be disastrous. It is easy to relate to this because we well understand how we hide our own private parts ☺.



As an example, consider the following code, which may appear in any class. It shows that we can directly access the **balance** of a **BankAccount**. This is clearly undesirable since there is little protection. Could you imagine if anyone could modify the balance of your bank account directly ?




```

BankAccount    myAccount = new BankAccount( "Mine" );

myAccount.balance = 1000000.00f;  // YAY
myAccount.balance = -1000000.00f; // WHY ...

```



In order to prevent direct access to important information we would need to prevent the code above from compiling/running. If we were to declare the **balance** instance variable as **private** within the **BankAccount** class, then the above code would not compile, thus solving the issue.

In general, while freedom to access/modify anything from anywhere seems like a friendly thing to do, it is certainly dangerous. Anyone could "stomp" all over our instance variables changing them at will. A general "rule-of-thumb" that should be followed is to declare ALL of your instance variables as **private** as follows:

```

public class Patient {
    private String    name;
    private int       age;
    private float     height;
    private char      gender;
    private boolean   retired;

    ...
}

```

Once we do this, then the following code will not work (when written in a class other than the **Patient** class):

```

public class SomeApplicationProgram {
    public static void main(String[] args) {
        Patient p = new Patient();
        p.name = "Sandy Beach";    // will NOT compile
        p.age = 15;                // will NOT compile
        p.height = 5.85f;          // will NOT compile
        p.gender = 'M';            // will NOT compile
        p.retired = false;         // will NOT compile
        System.out.println(p.name); // will NOT compile
        System.out.println(p.age);  // will NOT compile
        System.out.println(p.height); // will NOT compile
        System.out.println(p.gender); // will NOT compile
        System.out.println(p.retired); // will NOT compile
    }
}

```



What we have essentially done is to erect a wall around the object ... like the wall around a city. We have encapsulated it with a protective bubble. Although we are still able to create the object, we are prevented from accessing or modifying its internals now from outside the class. By doing this, we have protected the object so much that we cannot get information neither into it nor out from it. We have kind of secluded the object from the rest of the world by doing this. However, just as a walled city has gates or doors to allow access, we too have a form of gated access by means of any publicly available methods.



We will grant access to "some" of our object's attributes (i.e., instance variables) by creating methods known as **get** and **set** methods (also called **getters** and **setters**). The idea of creating these gateways to our object's data is common practice and is considered to be a robust strategy when creating classes to be used in a large software application. In this course, since we are only creating a few classes and since we are the only code writers, we may not immediately see the benefits of declaring **private** attributes and then creating these methods. However, in a larger/complicated system with hundreds of classes, the benefits become quite clear:

- object attributes are easier to understand and use
- attributes are protected from external/unknown changes
- we are following proper and robust coding style

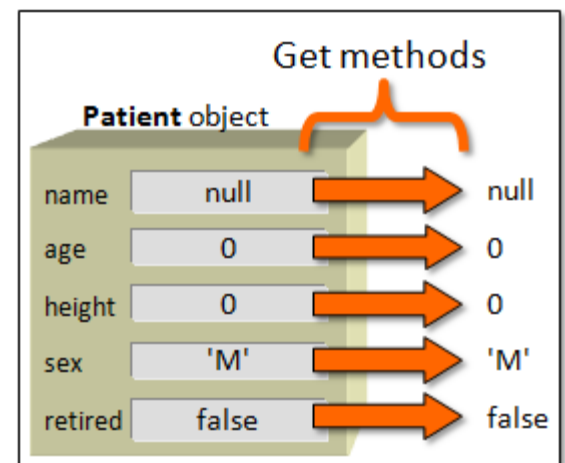
Let us first consider **get** methods. They let us look at information that is within the object by getting the object's attribute values (i.e., get the values of the instance variables). **Get** methods have:

- **public** access
- return type matching attribute's type
- name matching attribute's name
- code returning attribute's value

Here is how we would write the standard **get** methods for a **Patient** class:

```
public class Patient {
    private String name;
    private int age;
    private float height;
    private char gender;
    private boolean retired;

    // Get methods for name, age, height, gender and retired attributes
    public String getName() { return this.name; }
    public int getAge() { return this.age; }
    public float getHeight() { return this.height; }
    public char getGender() { return this.gender; }
    public boolean isRetired() { return this.retired; }
}
```



Notice that all the methods look the same in structure. They are all **public**, all have return types and names that match the attribute type, all have no parameters and all are one line long.

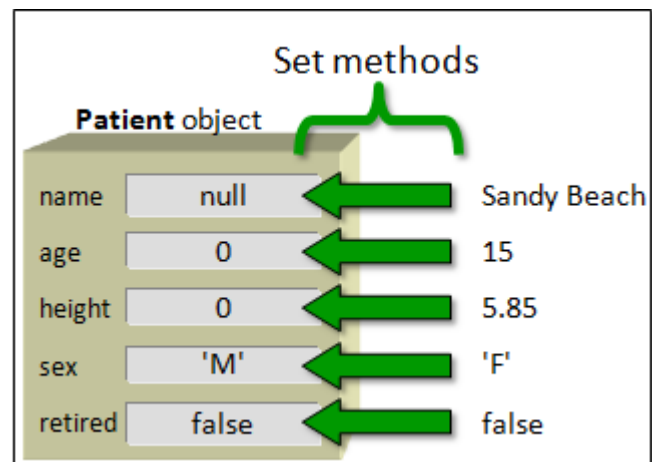
When we call the method to get the attribute value, the method simply returns the attribute value to us. It's quite simple. By convention, all get methods start with “**get**” followed by the attribute name, with the exception of attributes that are of type **boolean**. In that case, we usually use “**is**” followed by the attribute name, as it makes the method call more natural.

Now let us examine the **set** methods. **Set** methods allow us to put values into the instance variables (i.e., to set the object's attributes). **Set** methods have:

- **public** access
- **void** return type
- name matching attribute's name
- a parameter matching attribute's type
- code giving the attribute a value

Here is how we would write the standard **set** methods for the **Patient** class:

```
// Set method for name attribute
public void setName(String n) {
    this.name = n;
}
// Set method for age attribute
public void setAge(int a) {
    this.age = a;
}
// Set method for height attribute
public void setHeight(float h) {
    this.height = h;
}
// Set method for gender attribute
public void setGender(char g) {
    this.gender = g;
}
// Set method for retired attribute
public void setRetired(boolean r) {
    this.retired = r;
}
```



The single line of code in a **set** method is quite simple also.

When we call the method to give the attribute a new value (i.e., we supply the new value as a parameter to the method), the method simply takes that new attribute value and sets the attribute to it by using the **=** operator.

Normally, we write all the **get** and **set** methods together, and sometimes shorten them onto one line. Also, they are often listed in the code right after the **public** constructors as follows:

```

public class Patient {
    private String    name;
    private int       age;
    private float     height;
    private char      gender;
    private boolean   retired;

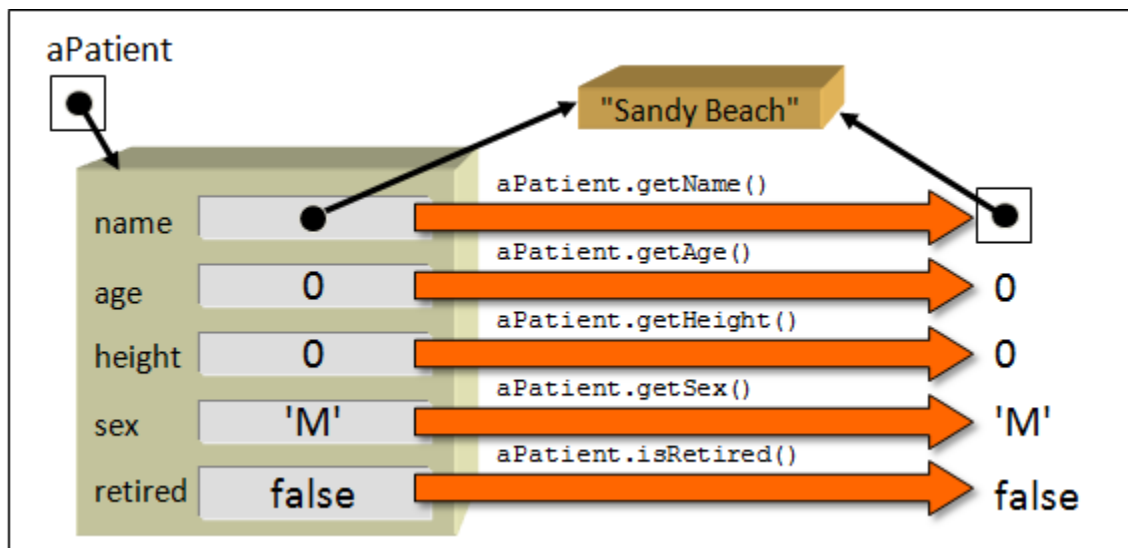
    // Constructor
    public Patient() {
        name = "Unknown";
        age = 0;
        height = 0;
        gender = '?';
        retired = false;
    }

    // Get methods
    public String getName() { return this.name; }
    public int  getAge()   { return this.age; }
    public float getHeight() { return this.height; }
    public char getGender() { return this.gender; }
    public boolean isRetired() { return this.retired; }

    // Set methods
    public void setName(String n) { this.name = n; }
    public void setAge(int a) { this.age = a; }
    public void setHeight(float h) { this.height = h; }
    public void setGender(char g) { this.gender = g; }
    public void setRetired(boolean r) { this.retired = r; }
}

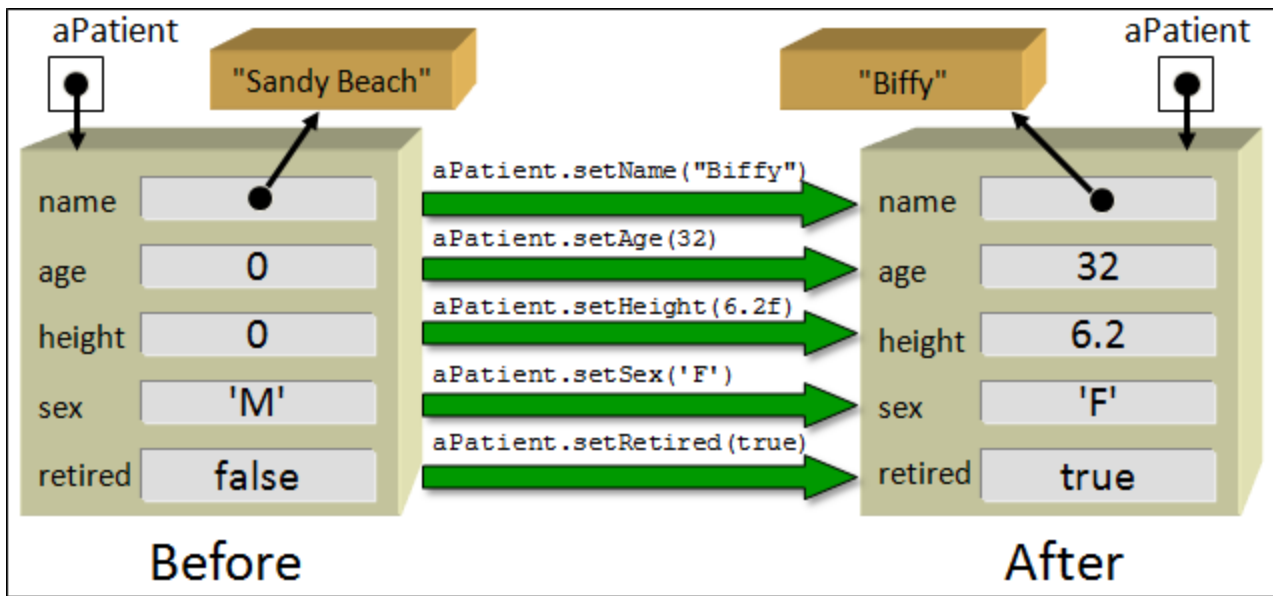
```

Here is how the **get** method works:



Notice that primitive attribute values are returned as simple values but *object* attribute values are returned as pointers/references to the object. The **Patient** object remains unchanged as a result of a get method call.

Here is how the **set** method works:



Notice that primitive attribute values are simply replaced with the new value. For object attribute values, after the **set** call, the attribute will point to the new object. The previous object that the attribute used to point to is discarded (i.e., garbage collected) if no other objects are holding on to it. Once we create these **get/set** methods, we can then access and modify the object from anywhere in our program as before:

```
public class TestPatientProgram {
    public static void main(String[] args) {
        Patient p = new Patient();

        System.out.println("Before Setting ...");
        System.out.println(p.getName()); // was println(p.name);
        System.out.println(p.getAge()); // was println(p.age);
        System.out.println(p.getHeight()); // was println(p.height);
        System.out.println(p.getGender()); // was println(p.gender);
        System.out.println(p.isRetired()); // was println(p.retired);

        p.setName("Sandy Beach"); // was p.name = "Sandy Beach";
        p.setAge(15); // was p.age = 15;
        p.setHeight(5.85f); // was p.height = 5.85f;
        p.setGender('F'); // was p.gender = 'F';
        p.setRetired(true); // was p.retired = true;

        System.out.println("\nAfter Setting ...");
        System.out.println(p.getName()); // was println(p.name);
        System.out.println(p.getAge()); // was println(p.age);
        System.out.println(p.getHeight()); // was println(p.height);
        System.out.println(p.getGender()); // was println(p.gender);
        System.out.println(p.isRetired()); // was println(p.retired);
    }
}
```

Here is what the output would be (however, initial values depend on the **Patient** constructor):

```
Before Setting ...
Unknown
0
0.0
?
false

After Setting ...
Sandy Beach
15
5.85
F
true
```

Now if we think for a moment ... what did we really do by making all the **get** and **set** methods ? Really, we wrote a lot of code (e.g., 5 **get** methods and 5 **set** methods for the **Patient** class) but did not gain anything new. The code does the same thing as before. In fact, the test code seems longer and perhaps slower (since we are calling a method to get/set the instance variables for us instead of accessing them directly). So why did we do this ? Let us review the advantages again:

1. First, **get/set** methods actually make life simpler for users of your class because the user does not have to understand the “guts” of the object being used. It allows them to treat the object as a “black box”. The user does not need to know about all the instance variables. Some are used to hold data that is temporary or private. You should only create public **get** methods for the instance variables that the user of the class would need to know about.
2. Second, it prevents the users of a class from directly modifying the object's internals. Recall, for example, that we should never be able to directly change the balance of our bank account without going through the proper transaction procedures such as depositing and withdrawal. Of course, if we always create **public get/set** methods for all our attributes, then we still would have no such protection. So, it is important to create **set** methods **only** for the attributes that you want the user of the class to be able to change directly. Therefore, you do not always need to make **set** methods.



---

## Restricting Class Access

In regards to class definitions, we are also allowed to indicate either *default* or **public** access to the class. So far, all of our classes have had **public** access, but we can have default access by leaving off the keyword **public**:

```
public class Manager { // public access from classes anywhere
    ...
}
```

```
class Employee { // default access from classes within package/folder
    ...
}
```

When we created inner classes for the purposes of event handling (e.g., **new ActionListener() {...}**), these had default access.

Interestingly, we can also declare a class as **final**. This means that it CANNOT have subclasses:

```
public final class Manager {
    ...
}
```

Why would we want to do this ? Perhaps the class has very weird code that the author does not want you to inherit ... maybe because it is too complicated and may easily be misused. Many of the JAVA classes (e.g., **ArrayList**) are declared as **final** which means that we cannot make any subclasses of them. It is a kind of security issue to prevent us from "messing up" the way those classes are meant to be used. It's a shame, because often we would like to have special types of **ArrayLists** and other similar objects ☹.

---

## Defining a Text-based "Look" for Your Model

As just described, properly-designed model classes will use *encapsulation* to hide any unnecessary information from those who will make use of those classes and to keep things simple. Sometimes however, it is desirable to be able to visually distinguish one object from another. For example, consider the following code:

```
public class MyObjectTestProgram {
    public static void main(String[] args) {
        System.out.println(new Patient()); // a patient object
        System.out.println(new Patient()); // another patient object
    }
}
```

The result on the screen is as follows:

```
Patient@7d8a992f  
Patient@164f1d0d
```

By default, JAVA displays all of the objects that you make in this manner, showing you the type of object (i.e., the class name) followed by something that represents the object's location in memory. This format for displaying objects is not very useful for debugging. If we had a dozen or so **Patient** objects displayed in this manner, we would not be able to "pick out" one that we may be looking for. It would be more advantageous if we had something a little more descriptive ... perhaps showing the patient's name.

What JAVA happens to be doing here is converting the **Patient** object to a **String** object first and then displaying the resulting characters to the screen. In fact, every object in JAVA has, by default, a method called **toString()** which will convert the object to a **String**.



The **Strings** returned from the call to **toString()** have the exact same characters that are displayed when we just display the objects directly using **System.out.println()**. That is because whenever JAVA attempts to display anything to the console, it automatically calls the **toString()** method for the object to convert it to characters before displaying. So, the two lines shown below do exactly the same thing:

```
Patient p = new Patient();  
  
System.out.println(p);           // displays Patient@7d8a992f  
System.out.println(p.toString()); // displays Patient@7d8a992f
```

Why do we care ? Well, we can actually replace the default **toString()** behavior by writing our own **toString()** method for all of our own objects that defines exactly how to convert our object to a **String**. That is, we can control the way our object "looks" when we print it on the screen or when we display it in our User Interface.

Suppose that we want our **Patient** object to display something like this when printed:

Patient named Hank

You should notice that the first two words of this output are fixed and it is only the last part (i.e., the first name of the **Patient**) that varies from patient to patient. We can make this to be the standard output format for all **Patient** objects simply by writing the following method in the **Patient** class:

```
public String toString() {  
    return ("Patient named " + this.name);  
}
```



This method overrides the default **toString()** method, essentially replacing it. Notice that the method is called **toString()** with no parameters and that it has a return type of **String**. This is important in order for the method to properly override the one inherited from the **Object** class.

Consider the output of the following code:

```
Patient          p1, p2, p3;

p1 = new Patient(); // assume first name is set to "" within constructor
p2 = new Patient();
p2.setName("Holly");
p3 = new Patient();
p3.setName("Hank");

System.out.println(p1);
System.out.println(p2);
System.out.println(p3);
```

Here is the output ...

```
Patient named
Patient named Holly
Patient named Hank
```

Now what if we wanted the output to be in this format instead:

```
19 year old Patient named Hank
```

To write an appropriate **toString()** method, we need to understand what is fixed in this output and what will vary. The number **19** should vary for each patient as well as the **first** and **last** names. Here is how we could write the code (replacing our previous **toString()** method):

```
public String toString() {
    return (this.age + " year old Patient named " + this.name);
}
```

Notice that the basic idea behind creating a **toString()** method is to simply keep joining together **String** pieces to form the resulting **String**. Now here is a harder one. Let us see if we can make it into this format:

```
19 year old non-retired patient named Hank
```

Here we have the **age** and **names** being variable again but now we also have the added variance of their **retirement status**.

Here is one attempt:

```
public String toString() {  
    return (this.age + " year old " + this.retired + " patient named "  
        + this.name);  
}
```

However, this is not quite correct. This would be the format we would end up with:

19 year old false patient named Hank

Notice that we cannot simply display the value of the **retired** attribute but instead need to write “retired” or “non-retired” for the **retired** status.

To do this then, we will need to use an **IF** statement. However, in JAVA, we cannot write an **IF** statement in the middle of a **return** statement. So we will need to do this using more than one line of code. We can make an **answer** variable to hold the result and then break down our method into logical pieces that append to this **answer**:

```
public String toString() {  
    String answer;  
  
    answer = this.age + " year old ";  
    answer = answer + this.retired;  
    answer = answer + " patient named " + this.name);  
  
    return answer;  
}
```

Now we can insert the appropriate **IF** statements as follows:

```
public String toString() {  
    String answer;  
  
    answer = this.age + " year old ";  
  
    if (this.retired)  
        answer = answer + "retired";  
    else  
        answer = answer + "non-retired";  
    answer = answer + " patient named " + this.name;  
  
    return answer;  
}
```

The result is what we wanted. Note however, that we can simplify this code a little further:

```

public String toString() {
    String answer = this.age + " year old ";

    if (!this.retired)
        answer = answer + "non-";

    return (answer + "retired patient named " + this.name);
}

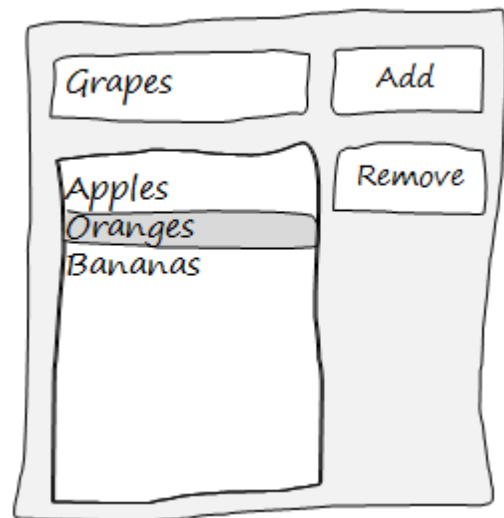
```

## Defining the Business Logic for Your Model

To finalize our model, we should decide what kinds of methods should be publically available so that the main application's user interface can access, modify and manipulate the model in meaningful ways.

For example, suppose that we wanted to develop the application that we described earlier that allowed us to make a list of things to purchase at the grocery store. What is the **model** in this application ?

To figure this out, we just have to understand what "lies beneath" the user interface. What is it that we are displaying and changing ? It is the list of items.



Let us develop a proper model for this interface. We can call it **ItemList** and it can keep track of an array of **Strings** that represent the list. Here is the basic model code:

```

public class ItemList {
    public final int    MAXIMUM_SIZE = 100;

    private String[]    items;
    private int         size;

    public ItemList() {
        items = new String[MAXIMUM_SIZE];
        size = 0;
    }

    public int getSize() { return size; }
    public String[] getItems() { return items; }
}

```

Looking back at the user interface, it is likely that we will want to add items to the list based on the text that is entered through the text field. The item to be added will likely go at the bottom of the list. So, we should make a public method to do this:

```
public void add(String item) {  
    // Make sure that we do not go past the limit  
    if (size < MAXIMUM_SIZE)  
        items[size++] = item;  
}
```

Likewise, the remove button will likely cause the currently selected item in the list to be removed from the list. We will probably remove it according to its index into the list. Here is a public method to do this:

```
public void remove(int index) {  
    // Make sure that the given index is valid  
    if ((index >= 0) && (index < size)) {  
        // Move every item after the deleted one up in the list  
        for (i=index; i<size; i++) {  
            items[i] = items[i+1];  
        }  
        // Reduce the list size by 1  
        size--;  
    }  
}
```

Therefore, here is our completed model:

```
public class ItemList {  
    public final int    MAXIMUM_SIZE = 100;  
  
    private String[]    items;  
    private int         size;  
  
    public ItemList() {  
        items = new String[MAXIMUM_SIZE];  
        size = 0;  
    }  
  
    public int getSize() { return size; }  
    public String[] getItems() { return items; }  
  
    public void add(String item) {  
        // Make sure that we do not go past the limit  
        if (size < MAXIMUM_SIZE)  
            items[size++] = item;  
    }  
    public void remove(int index) {  
        // Make sure that the given index is valid  
        if ((index >= 0) && (index < size)) {  
            // Move every item after the deleted one up in the list  
            for (int i=index; i<size; i++)  
                items[i] = items[i+1];  
            size--;    // Reduce the list size by 1  
        }  
    }  
}
```

## Testing Your Model

Once we have the model, it is always a good idea to **test it!** The easiest way to do this is to write a simple test program to try out the various methods. We should write a test program that does a thorough testing. We should at least try to add a few items to the list, remove one, remove too many and add too many:

```
public class ItemListTestProgram {
    public static void main(String[] args) {
        // Make a new list
        ItemList groceryList = new ItemList();
        System.out.println("Grocery list has " + groceryList.getSize() + " items");

        // Add a few items
        System.out.println("\nAdding Apples, Oranges and Bananas ...");
        groceryList.add("Apples");
        groceryList.add("Oranges");
        groceryList.add("Bananas");
        System.out.println("Grocery list has " + groceryList.getSize() + " items");
        System.out.println("Here are the items in the list:");
        for (int i=0; i<groceryList.getSize(); i++)
            System.out.println(groceryList.getItems()[i]);

        // Remove an item
        System.out.println("\nRemoving Apples ...");
        groceryList.remove(0);
        System.out.println("Grocery list has " + groceryList.getSize() + " items");
        System.out.println("Here are the items in the list:");
        for (int i=0; i<groceryList.getSize(); i++)
            System.out.println(groceryList.getItems()[i]);

        // Try to remove too many items
        System.out.println("\nTrying to remove too many items ...");
        groceryList.remove(0);
        groceryList.remove(0);
        groceryList.remove(0);
        groceryList.remove(0);
        System.out.println("Grocery list has " + groceryList.getSize() + " items");
        System.out.println("Here are the items in the list:");
        for (int i=0; i<groceryList.getSize(); i++)
            System.out.println(groceryList.getItems()[i]);

        // Try to add too many items
        System.out.println("\nTrying to add too many items ...");
        for (int i=0; i<200; i++)
            groceryList.add("Item# " + i);
        System.out.println("Grocery list has " + groceryList.getSize() + " items");
        System.out.println("Here are the items in the list:");
        for (int i=0; i<groceryList.getSize(); i++)
            System.out.println(groceryList.getItems()[i]);
    }
}
```

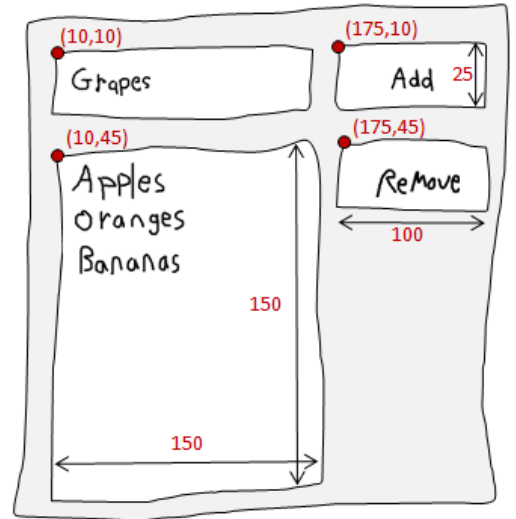
Notice how the test program is nicely formatted with comments indicating what is being tested. Also, notice that there are descriptive print statements that explain what is happening. You should follow a similar style and approach when writing your test programs.

## 5.3 Developing a *Proper View*

Once you have developed and properly tested the model for your application, you can then begin to write the user interface. Sometimes, however, it may be beneficial to have a rough idea as to what your user interface will do before you develop the model. For example, we did not write the **add()** and **remove()** methods in the **ItemList** class until we realized that we needed them based on what we want our completed application to do.

The next step will be to develop the view for the application. In general, there may be many views in an application, just as there may be many models and controllers. However, we will assume for now that we have a single view.

To keep things simple, we will develop our views as **JPanels** that can be placed onto our **JFrame** windows. Looking back at the chapter on Graphical User Interfaces, you will recall that we made a similar window in our **FruitListApp**. The following code should therefore be easily understood (refer back to chapter 4 if you do not recall):



```
import javax.swing.*;

public class GroceryListView extends JPanel {
    public GroceryListView() {
        // Choose to lay out components manually
        setLayout(null);

        // Add the text field
        JTextField newItemField = new JTextField();
        newItemField.setLocation(10,10);
        newItemField.setSize(150,25);
        add(newItemField);

        // Add the ADD button
        JButton addButton = new JButton("Add");
        addButton.setLocation(175, 10);
        addButton.setSize(100,25);
        add(addButton);

        // Add the REMOVE button
        JButton removeButton = new JButton("Remove");
        removeButton.setLocation(175,45);
        removeButton.setSize(100,25);
        add(removeButton);
    }
}
```

```

    // Add the JList
    JList aList = new JList();
    JScrollPane scrollPane = new JScrollPane(aList,
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    scrollPane.setLocation(10,45);
    scrollPane.setSize(150,150);
    add(scrollPane);

    setSize(290, 230); // manually computed sizes
}

```

For the purposes of a quick test to make sure that our view is properly formatted, we can create and run a simple program like this:

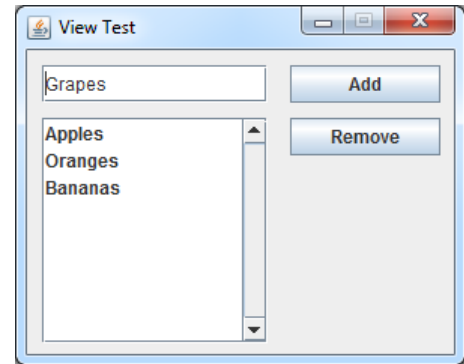
```

import javax.swing.*.*;

public class GroceryListViewTestProgram {
    public static void main(String[] args) {
        JFrame frame = new JFrame("View Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(290, 230); // manually computed sizes
        frame.setResizable(false);

        frame.getContentPane().add(new GroceryListView());
        frame.setVisible(true);
    }
}

```



Now, for the view to work properly it must refresh its look based on the most up-to-date information in the model. Therefore, we need a way of having the view update itself given a specific model. There are many ways to do this, but a simple way is to write a method called **update()** that will refresh the "look" of the view whenever it is called. Of course, to be able to update, the view must have access to the model. We can pass the model in as a parameter to the view constructor and store it as an attribute of the view:

```

public class GroceryListView extends JPanel {
    private ItemList model; // The model to which this view is attached

    public GroceryListView(ItemList m) {
        model = m; // Store the model for access later

        ...
    }
}

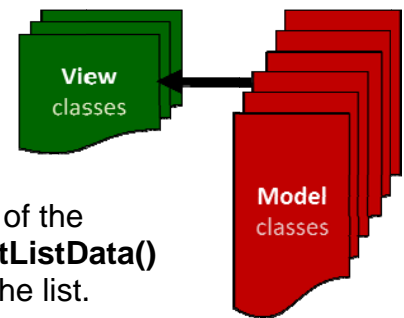
```

Since the model attribute is simply a reference (i.e., a pointer in memory) to the **ItemList**, then any changes to the **ItemList** will also affect the model stored in this **model** instance variable.

Our **update()** method may be as simple as replacing the entire **JList** with the new items currently stored in the model. So, regardless of what changes take place in the **ItemList**



model, we simply read the list of items and re-populate the **JList** with the latest items. To be able to do this, we will need access to that **JList**. However, the **JList** is defined as a local variable in the constructor. In order to be able to access it from the **update()** method, we will need to define the variable outside of the constructor as an instance variable. Then, to change the contents of the **JList**, a quick search in the JAVA API informs us that there is a **setListData()** method which will allow us to pass in an array of items to show in the list.



Here is the view code now:

```
public class GroceryListView extends JPanel {
    private ItemList model; // The model to which this view is attached
    private JList aList; // The visible list representing the model

    public GroceryListView(ItemList m) {
        model = m; // Store the model for access later

        ...
        aList = new JList();
        ...
    }
    public void update() {
        aList.setListData(model.getItems());
    }
}
```

Notice how the **JList** is now easily accessible in the **update()** method. One problem, however, is that our model's array is always of size 100 regardless of how many items have been placed in it. The **setListData()** method will end up making a list of 100 items in it ... leaving many blanks. Looking at the API again, there is another **setListData()** method ... but one that takes a **Vector** as a parameter. We will discuss this later in the course. For now, we can add additional code here to make a new array (for display purposes) which has a length exactly equal to the size of the **items** array. Change the **update()** method to this:

```
public void update() {
    // Create and return a new array with the
    // exact size of the number of items in it
    String[] exactList = new String[model.getSize()];
    for (int i=0; i<model.getSize(); i++)
        exactList[i] = model.getItems()[i];

    aList.setListData(exactList);
}
```

Thinking ahead a little, we know that eventually the application will need to respond to user input through pressing the **Add** or **Remove** button, typing in the text field or selecting from the list. This will be part of the controller. However, in order to accomplish this, as you will soon see, we need to allow the controller to access the **JButtons**, the **JList** and the **JTextField**.

We can allow such access by making instance variables for all four window components and then provide **get** methods to access them.

As a final programming aspect of the view class, it is a good idea to call the **update()** method at the end of the constructor. That way, when the view is first created, it can be refreshed right away to show the true state of the model upon startup.

You should follow this same standard approach when designing your views:

```
import javax.swing.*;

public class GroceryListView extends JPanel {
    private ItemList model; // The model to which this view is attached

    // The user interface components needed by the controller
    private JList aList;
    private JButton addButton;
    private JButton removeButton;
    private JTextField newItemField;

    // public methods to allow access to JComponents
    public JList getList() { return aList; }
    public JButton getAddButton() { return addButton; }
    public JButton getRemoveButton() { return removeButton; }
    public JTextField getNewItemField() { return newItemField; }

    public GroceryListView(ItemList m) {
        model = m; // Store the model for access later

        // Choose to lay out components manually
        setLayout(null);

        // Add the text field
        newItemField = new JTextField();
        newItemField.setLocation(10,10);
        newItemField.setSize(150,25);
        add(newItemField);

        // Add the ADD button
        addButton = new JButton("Add");
        addButton.setLocation(175, 10);
        addButton.setSize(100,25);
        add(addButton);

        // Add the REMOVE button
        removeButton = new JButton("Remove");
        removeButton.setLocation(175,45);
        removeButton.setSize(100,25);
        add(removeButton);
    }
}
```

```

        // Add the JList
        aList = new JList();
        JScrollPane scrollPane = new JScrollPane(aList,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        scrollPane.setLocation(10,45);
        scrollPane.setSize(150,150);
        add(scrollPane);

        setSize(290, 230); // manually computed sizes

        // Call update() to make sure model contents are shown
        update();
    }

    // Update the view to show the model's state
    public void update() {
        // Create and return a new array with the
        // exact size of the number of items in it
        String[] exactList = new String[model.getSize()];
        for (int i=0; i<model.getSize(); i++)
            exactList[i] = model.getItems()[i];
        aList.setListData(exactList);
    }
}

```

We can run a quick test to make sure that this is working...

```

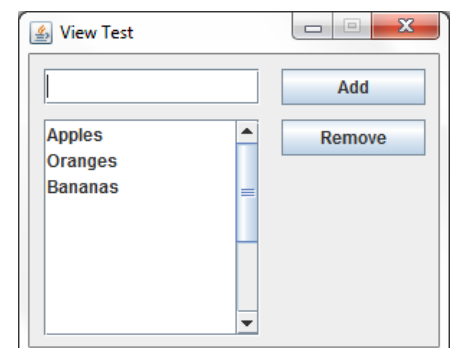
import javax.swing.*.*;

public class GroceryListViewTestProgram2 {
    public static void main(String[] args) {
        ItemList groceryList = new ItemList();
        groceryList.add("Apples");
        groceryList.add("Oranges");
        groceryList.add("Bananas");

        GroceryListView aView = new GroceryListView(groceryList);

        JFrame frame = new JFrame("View Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(290, 230); // manually computed sizes
        frame.setResizable(false);
        frame.getContentPane().add(aView);
        frame.setVisible(true);
    }
}

```

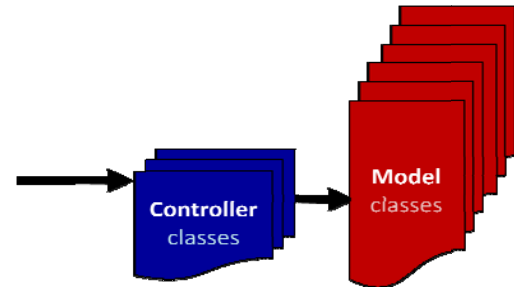


The window should come up now with some fruits listed in the list. We will add some more functionality to the **View** later, but for now, we need to understand how to complete the final portion of our MVC application... the controller ...

## 5.4 Developing a *Proper* Controller

The final piece of our application is the controller. The controller is responsible for taking in user input and making changes to the model accordingly. These changes can then be refreshed with a simple call to **update()** in the view class.

The controller will be our **JFrame** class, representing the whole application. It will tie together the model and the view. In addition the controller is where we "hook up" our event handlers to handle the user interaction. It will handle all user input and then change the model accordingly ... updating the view afterwards.



To begin, we can define the class such that it creates a new view and a new model. Here is the basic structure ... we will be adding the event handlers one by one. Take note of how cleanly separated the model and the view are ... as they are stored separately as attributes of the controller:

```

import javax.swing.*;           // Needed for JFrame
import java.awt.event.*;       // Need soon for ActionListener
import javax.swing.event.*;    // Need soon for ListSelectionListener, DocumentListener

public class GroceryListApplication extends JFrame {
    private ItemList      model; // The model to which this view is attached
    private GroceryListView view; // The view that shows the state of the model

    public GroceryListApplication(String title) {
        super(title); // Sets the title of the window

        // Create the model and view
        model = new ItemList();
        view = new GroceryListView(model);

        // Add the view
        getContentPane().add(view);

        // Add the event handlers
        // ... coming soon ...

        // Manually computed size
        setSize(290, 230);
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    // This is where the program begins
    public static void main(String[] args) {
        JFrame frame = new GroceryListApplication("My Grocery List");
        frame.setVisible(true);
    }
}
  
```

Now it is time to get everything working. We will approach this slowly by adding functionality as we go along. Make sure you understand where the new pieces of code will "fit" into the three classes that we already have.

## Adding Items

To add an item to the Grocery List, we will require the user to type the item into the text field and then press the **Add** button. Let us write an event handler for when the **Add** button is pressed. It will need to get the contents of the text field and then insert that string as a new item in the model. Then the view should be updated. So we should add this method to the controller (i.e., the main application):

```
// The Add Button event handler
private void handleAddButtonPress() {
    model.add(view.getNewItemField().getText());
    view.update();
}
```

Notice that this method is **private**, since no external classes should be calling it. The code does two main things that ALL of your event handlers should do:

1. Change the Model
2. Update the View



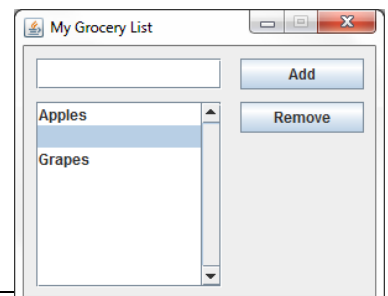
Now, we need to "plug it in" to the **JButton**. We will need to add an **ActionListener** to the button. Insert the following into the controller's constructor:

```
view.getAddButton().addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) { handleAddButtonPress(); }});
```

Notice how the **JButton** is accessed from the view. The listener is added and points to the event handler method that we just wrote. The code should now work and allow new items to be added to the list.

However, a slight problem occurs when the user does not have anything typed into the text field and then presses the **Add** button. A "blank" item is added to the list. This is not pleasant. We can ensure that no blank items are added by altering our code a little:

```
private void handleAddButtonPress() {
    String text = view.getNewItemField().getText().trim();
    if (text.length() > 0) {
        model.add(text);
        view.update();
    }
}
```



You may have noticed the **trim()** method added here. This is a **String** method that removes any leading and trailing space and tab characters. That will ensure that we do not add any items consisting of only spaces or tab characters.

## Removing Items

Similarly, to remove add an item from the Grocery List, we will require the user to select the item from the list and then press the **Remove** button. Let us write an event handler for when the **Remove** button is pressed. It will need to get the index of the selected item from the list and call the model's **remove()** method using this index. Then the view should be updated. To get the selected item from the list, we can look in the JAVA API and determine that the **JList** method we need to call is **getSelectedIndex()**. This method returns -1 if nothing is selected, so we should handle that. So we should add this method to the controller (i.e., the main application):

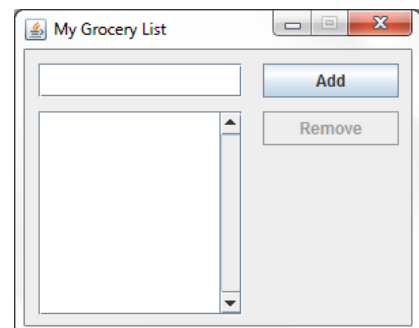
```
// The Remove Button event handler
private void handleRemoveButtonPress() {
    int index = view.getList().getSelectedIndex();
    if (index >= 0) {
        model.remove(index);
        view.update();
    }
}
```

We are now done with the basic functionality of our program. We could stop here. However, we will continue with some additional fine-tuning...

## Disabling the Remove Button

It would be a good idea to disable the **Remove** button when nothing has been selected in the list. That way the user knows visually that the **Remove** operation is now valid until something is selected. It is a good form of feedback to the user and it makes the user interface more intuitive to use.

Since this is simply a visual change, we could simply add a line to the **update()** method in the view class:



```
public void update() {
    // Create and return a new array with the
    // exact size of the number of items in it
    String[] exactList = new String[model.getSize()];
    for (int i=0; i<model.getSize(); i++)
        exactList[i] = model.getItems()[i];
    aList.setListData(exactList);

    // enable/disable the remove button accordingly
    removeButton.setEnabled(aList.getSelectedIndex() >= 0);
}
```

Notice that if the list has something in it, then the selected index will be 0 or more. We can use this boolean result to set the button to be enabled or disabled. If we start up the application, the **Remove** button will be disabled properly. When should it be re-enabled? According to our **update()** method, whenever something is selected from the list it will be enabled. However, we need to ensure that the **update()** method is called when the user selects something from the list. Whenever the user clicks in the list, we can simply update the view to ensure that the **Remove** button is re-enabled. Here is the simple event handler to put into the controller class:

```
// The List click event handler
private void handleListSelection() {
    view.update();
}
```

There are a few ways to cause this to occur. The simplest is to add a **mousePressed** event handler to the **JList**. Again, we plug it in by adding this to the constructor:

```
view.getList().addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) { handleListSelection(); });
```

A minor issue is that when we update the list, the currently selected item becomes unselected. We can fix this by remembering what is selected before we update the list and then re-select that item again afterwards. In the view, we can alter the **update()** method to do this:

```
// Update the view to show the model's state
public void update() {
    //Remember what was selected
    int selectedItem = aList.getSelectedIndex();

    // Now re-populate the list
    String[] exactList = new String[model.getSize()];
    for (int i=0; i<model.getSize(); i++)
        exactList[i] = model.getItems()[i];
    aList.setListData(exactList);

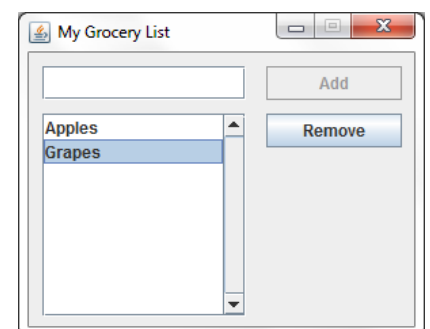
    // Reselect the selected item
    aList.setSelectedIndex(selectedItem);

    removeButton.setEnabled(aList.getSelectedIndex() >= 0);
}
```

## Disabling the Add Button

Finally, it would also be a good idea to disable the **Add** button when nothing is typed in the text field.

To do this, we need to add a single line to the **update()** method in the view that is similar to the one we added to enable/disable the **Remove** button. However, this time we look at the text in the text field:





```
public void update() {  
    ...  
    addButton.setEnabled(newItemField.getText().trim().length() > 0);  
}
```

Of course, once again we need to have the button re-enabled when the user starts typing into the text field. We need an event that occurs when the user types into a text field. Again the event handler is simple:

```
// The text field typing event handler  
private void handleTextEntry() {  
    view.update();  
}
```

Plugging this in to the controller, however, is a little trickier. Modifying the contents of the text field can be done via single character typing or pasting some text or selecting/deleting text etc... Because of the complexity, JAVA decided to create a **Document** object for each **JTextField**. So we will need to implement a **DocumentListener**. Each **JTextField** has a method called **getDocument()** that gets the document that belongs to the text field. We add the listener to that document as follows:

```
view.getNewItemField().getDocument().addDocumentListener(new DocumentListener() {  
    public void changedUpdate(DocumentEvent theEvent) { handleTextEntry(); }  
    public void insertUpdate(DocumentEvent theEvent) { handleTextEntry(); }  
    public void removeUpdate(DocumentEvent theEvent) { handleTextEntry(); } });
```

Notice that we need to implement three methods to handling inserting, removing and changing of the text in any way. In our situation, we do not care to distinguish between these three since any changes in the text field should generate an **update()**.

## Clearing the Text Field

One final alteration to the program would be to clear the text field after an item has been added. Otherwise, after each an item has been added, the user will have to delete the text before adding the next item. This can be tedious.

To accomplish this, we simply add one more line to the **Add** button event handler to clear the text:

```
// The Add Button event handler  
private void handleAddButtonPress() {  
    String text = view.getNewItemField().getText().trim();  
    if (text.length() > 0) {  
        view.getNewItemField().setText("");  
        model.add(text);  
        view.update();  
    }  
}
```

Finally, our application works as desired. We are done.