
Chapter 6

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 how to prepare your model classes for use in a GUI. 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.



6.1 Separating Model, View and Controller Components

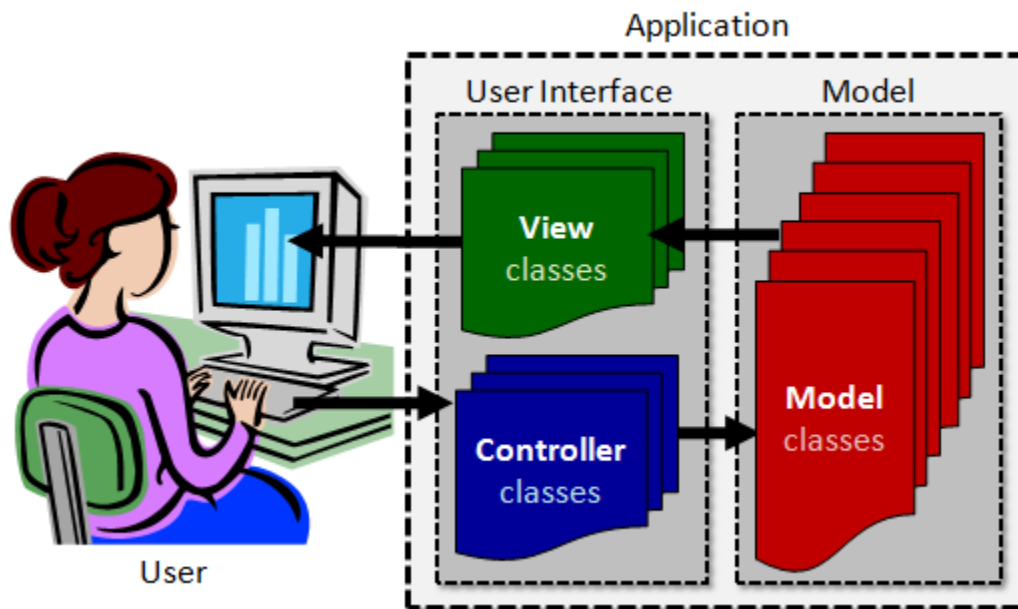
In the previous chapter, we discussed the difference between **model** classes and those classes that are part of the **user interface**. The model classes deal with the *business logic* aspects of the application and the user interface is the "front end" which allows the user to interact with the model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

6.2 Preparing Your Model Classes for the GUI

You already know how to build model classes ... they are the classes that make up your application apart from the user interface components. It is a good idea to prepare your model so that you can interact with it in a simple and clean manner from your user interface. A bank machine, for example, is somewhat pointless unless the underlying **Bank** model is fully operational.

To finalize our model classes, 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 ?

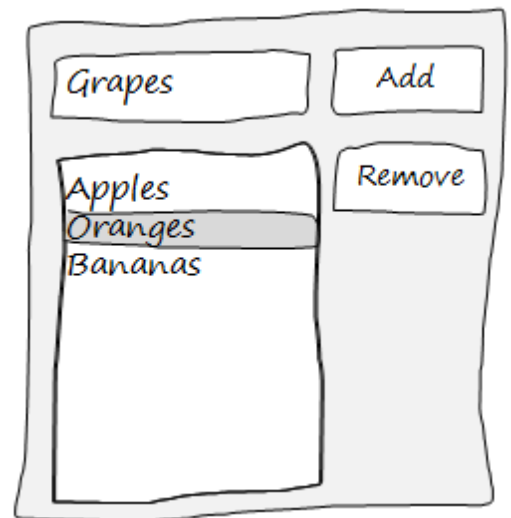
To figure this out, we just have to understand what "lies beneath" the user interface.

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 code for this simple 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; }
}
```

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 (int 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
        }
    }
}

```

Once we have the model implemented with useful methods, 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("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("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("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("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("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]);
}
}
```

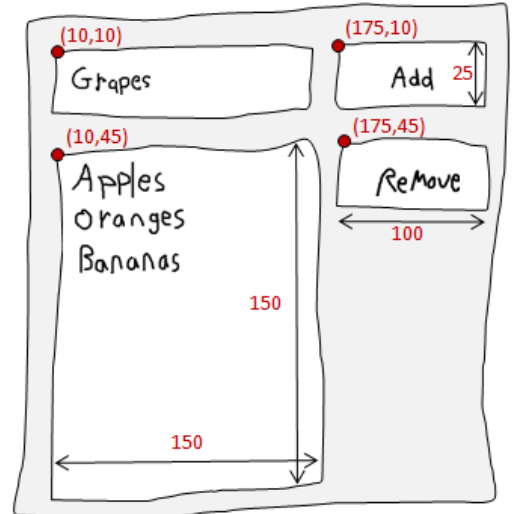
Notice how the test program is nicely formatted with comments indicating what is being tested. Also, notice that there are descriptive print statements that explains what is happening. You should follow a similar style and approach when writing your test programs.

6.3 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);

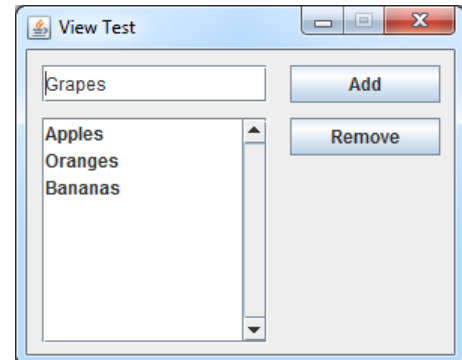
        // Add the JList
        JList aList = new JList();
        JScrollPane scrollPane = new JScrollPane(aList,
            JScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
```

```

        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

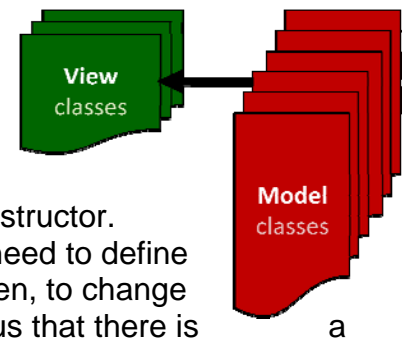
    public GroceryListView(ItemList m) {
        model = m; // Store the model for access later

        ...
    }
}

```

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 **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...

```

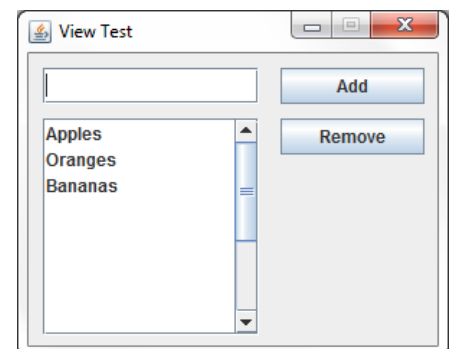
import javax.swing.*.*;

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);
    }
}

```

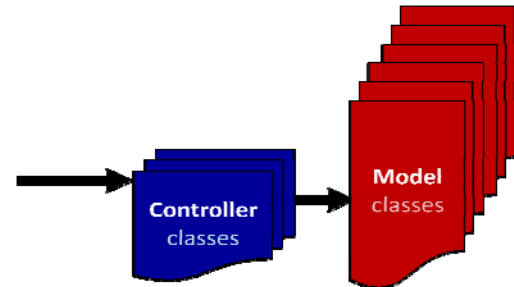


The window should come up now with some fruits listed in the list. We will add some more functionality to the **View** later, but for now, we need to understand how to complete the final portion of our MVC application... the controller ...

6.4 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

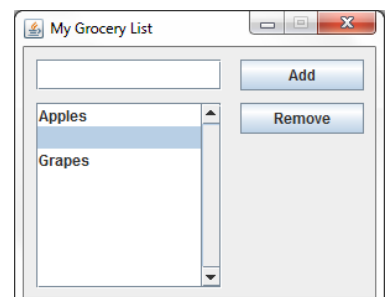


Now, we need to "plug it in" to the **JButton**. We will need to add an **ActionListener** to the button. Insert the following into the controller's constructor:

```
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 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();
    }
}
```

Of course, to get it to work, we need to insert the following into the controller's constructor:

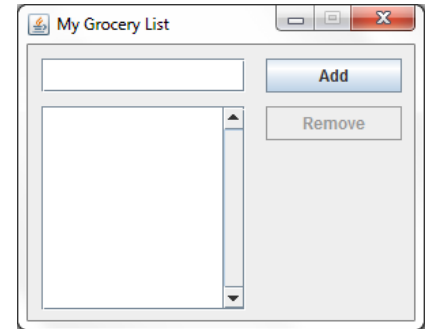
```
view.getRemoveButton().addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        handleRemoveButtonPress();
    }
});
```

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 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);

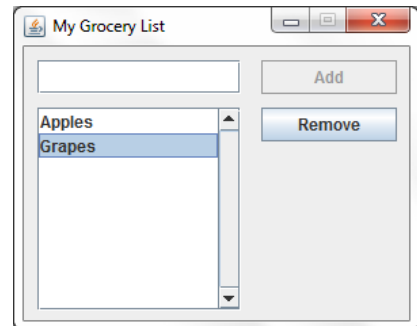
    // Reselecte the selected item
    aList.setSelectedIndex(selectedItem);

    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 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.

This page was intentionally left blank.