3 Events and Listeners

What's in This Set of Notes?

Now that we know how to design the "look" of a window by placing components on it, we need to make the window respond properly to the user interaction. The techniques are based on something called "Event Handling". In JAVA, we handle events by writing "Listeners" (also known as event handlers).

Here are the individual topics found in this set of notes (click on one to go there):

3.1 Events and Event Handlers

In the previous set of notes, we have seen how to create a GUI with various types of components. However, none of the components on the window seem to respond to the user interactions. In order to get the interface to "work" we must make it respond appropriately to all user input such as clicking buttons, typing in text fields, selecting items from list boxes etc... To do this, we must investigate ***events***.

What is an event ?

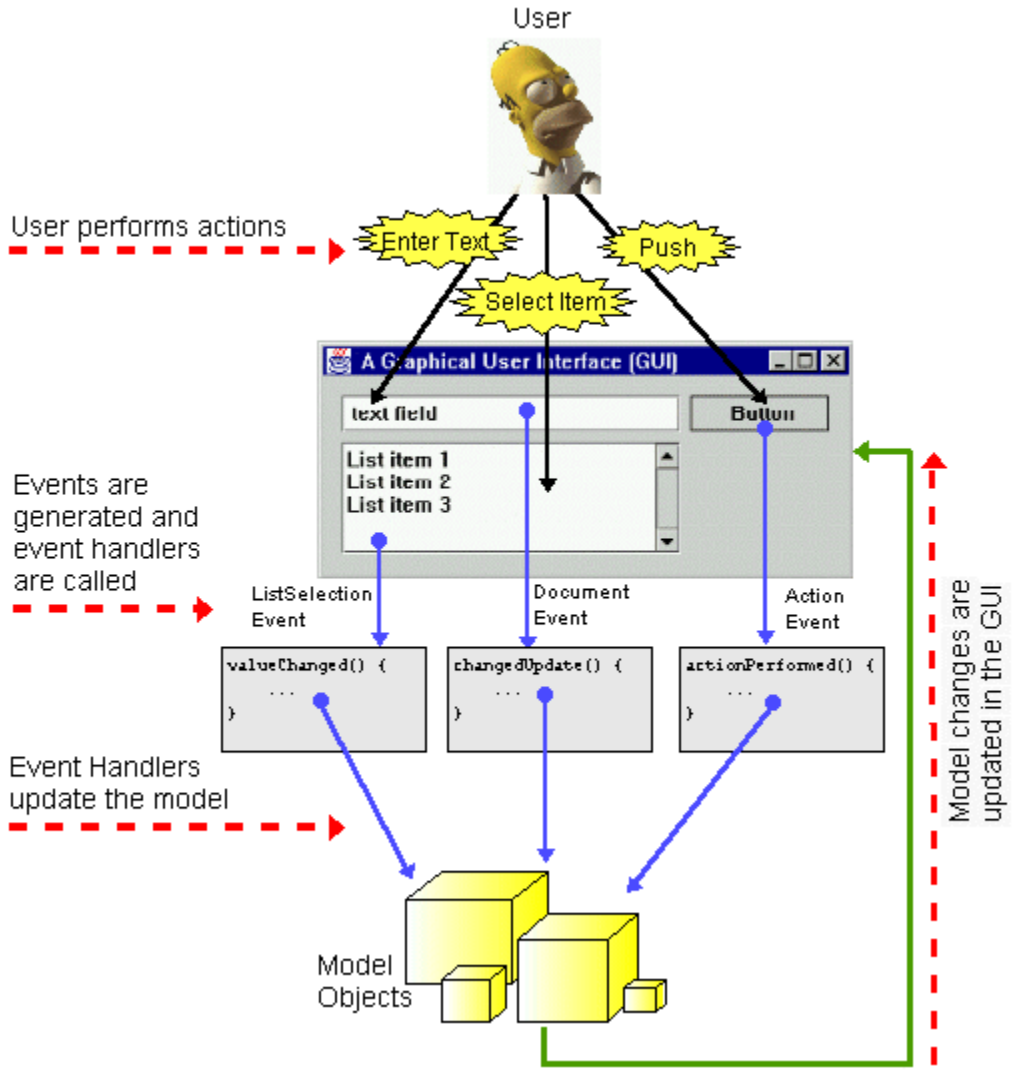
- An ***event*** is something that happens in the program based on some kind of triggering input.
- typically caused (i.e., generated) by user interaction (e.g., mouse press, button press, selecting from a list etc...)
  - the component that caused the event is called the ***source***.
- can also be generated internally by the program

How are Events Used in JAVA ?

- events are objects, so each type of event is represented by a distinct class (similar to the way exceptions are distinct classes)
- low-level events represent window-system occurrences or low-level input such as mouse and key events and component, container, focus, and window events.
- some events may be ignored, some may be ***handled***. We will write ***event handlers*** which are known as ***listeners***.

Nothing happens in your program UNLESS an event occurs. JAVA applications are thus considered to be ***event-driven***.

Here is a picture that describes the process of user interaction with a GUI through events:



Basically...here's how it works:

1. The user causes an event (e.g., click button, enter text, select list item etc...)

2. The JAVA VM invokes (i.e., triggers) the appropriate event handler (if it has been implemented and registered).
    - This invocation really means that a **method is called** to handle the event.
3. The code in the event handling method changes the model in some way.
4. Since the model has changed, the interface will probably also change and so components should be updated.

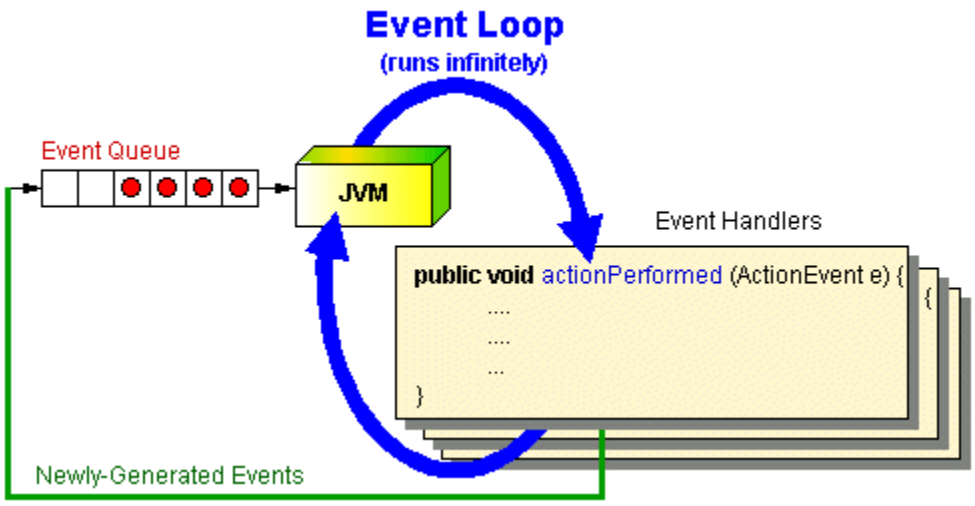
Notice that JAVA itself waits for the user to initiate an action that will generate an event.

- This is similar to the situation of a cashier waiting for customers ... the cashier does nothing unless an event occurs. Here are some events which may occur, along with how they may be handled:
    - a customer arrives - employee wakes up and looks sharp
    - a customer asks a question - employee gives an answer
    - a customer goes to the cash to buy - employee initiates sales procedure
    - time becomes 6:00pm - employee goes home

JAVA acts like this employee who waits for a customer action. JAVA does this by means of something called an ***EventLoop***. An ***Event Loop*** is an endless loop that waits for events to occur:

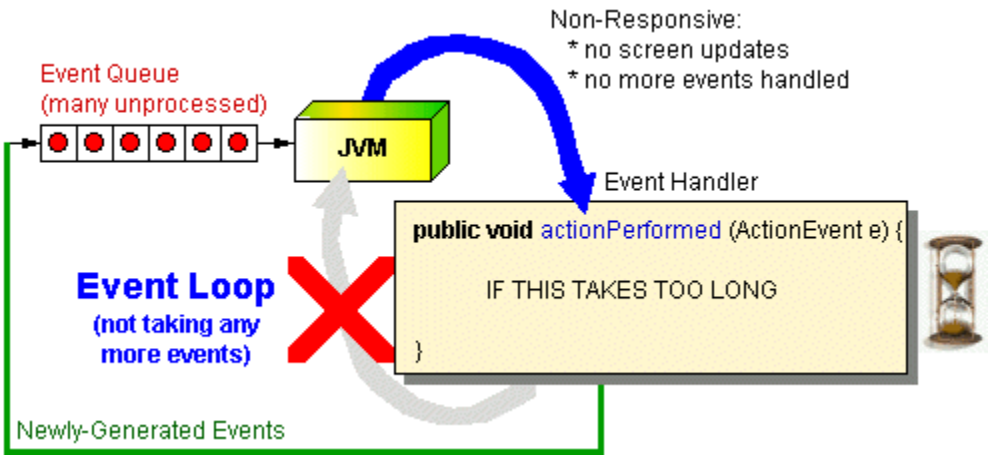
- events are queued (lined up on a first-come-first-served basis) in a buffer
- events are handled one at a time by an ***event handler*** (i.e., code that evaluates when event occurs)
- everything you want done in your application MUST go through this loop

Here is a picture of how the event loop works:



Notice that incoming events (i.e., customers/clients) are stored in the event queue in the order that they arrive. As we will see later, events MUST be handled by your program. The JVM spends all of its time taking an event out of the queue, processing it and then going back to the queue for another.

While each event is being handled, JAVA is unable to process any other events. You MUST be VERY careful to make sure that your event handling code does not take too long. Otherwise the JVM will not take any more events from the queue. This makes your application seem to "hang" so that the screen no longer updates, and all buttons, window components seem to freeze up !!!



In a way, the JVM event loop acts as a *server*. It serves (or handles) the incoming events one at a time on a first-come-first-served basis. So when an event is generated, JAVA needs to go to the appropriate method in your code to handle the event. How does JAVA know which method to call ? We will ***register*** each event-handler so that JAVA can call them when the events are generated. These event-handlers are called ***listeners*** (or ***callbacks***).

A ***listener:***

- acts on (i.e., handle) the event notification.
- must be ***registered*** so that it can be notified about events from a particular source.
- can be an instance of any class (as long as the class implements the appropriate listener interface)

So ... when creating a GUI, we must:

- decide what types of events we want to handle
- inform JAVA which ones we want to handle by *registering* the event handlers (i.e., the listeners)
- write the event handling code for each event

3.2 Listeners and Adapter Classes

You should understand now that when the user interacts with your user interface, some events will be generated automatically by JAVA. There are many types of events that can occur, and

we will choose to respond to some of them, while ignoring others. The JAVA VM is what actually generates the events, so we will have to "speak JAVA's language" in order to understand what the event means. In fact, to handle a particular event, we will have to write a particular method with a predefined name (chosen by JAVA).

Here is a list of the commonly used types of events in JAVA:

- **Action Events**: clicking buttons, selecting items from lists etc....
- **Component Events**: changes in the component's size, position, or visibility.
- **Focus Events**: gain or lose the ability to receive keyboard input.
- **Key Events**: key presses; generated only by the component that has the current keyboard focus.
- **Mouse Events**: mouse clicks and the user moving the cursor into or out of the component's drawing area.
- **Mouse Motion Events**: changes in the cursor's position over the component.
- **Container Events**: component has been added to or removed from the container.

Here are a couple of the "less used" types of events in JAVA:

- **Ancestor Events**: containment ancestors is added to or removed from a container, hidden, made visible, or moved.
- **Property Change Events**: part of the component has changed (e.g., color, size,...).

For each event type in JAVA, there are defined interfaces called **Listeners** which we must implement. Each listener interface defines one or more methods that MUST be implemented in order for the event to be handled properly.

There are many types of events that are generated and commonly handled. Here is a table of some of the common events. The table gives a short description of when the events may be generated, gives the interface that must be implemented by you in order for you to handle the events and finally lists the necessary methods that need to be implemented. Note, for a more complete description of these events, listeners and their methods, see the JAVA API specifications.

Event Type	**Generated By**	**Listener Interface**	**Methods that "YOU" must Write**
ActionEvent	a button was pressed, a menu item selected, pressing enter key in a text field or a timer event was generated	ActionListener	actionPerformed(ActionEvent e)
CaretEvent	moving cursor (caret) in a text-related component such as a	CaretListener	caretUpdate(CaretEvent e)

	JTextField		
ChangeEvent	value of a component such as a JSlider has changed	ChangeListener	stateChanged(ChangeEvent e)
DocumentEvent	changes have been made to a text document such as insertion, removal in an editor	DocumentListener	changedUpdate(DocumentEvent e) insertUpdate(DocumentEvent e) removeUpdate(DocumentEvent e)
ItemEvent	caused via a selection or deselection of something from a list, a checkbox or a toggle button	ItemListener	itemStateChanged(ItemEvent e)
ListSelectionEvent	selecting (click or double click) a list item	ListSelectionListener	valueChanged(ListSelectionEvent e)
WindowEvent	open/close, activate/deactivate, iconify/deiconify a window	WindowListener	windowOpened(WindowEvent e) windowClosed(WindowEvent e) windowClosing(WindowEvent e) windowActivated(WindowEvent e) windowDeActivated(WindowEvent e) windowIconified(WindowEvent e) windowDeiconified(WindowEvent e)
FocusEvent	a component has gained or lost focus. Pressing tab key changes focus of components in a window	FocusListener	focusGained(FocusEvent e) focusLost(FocusEvent e)
KeyEvent	pressing and/or releasing a key while within a component	KeyListener	keyPressed(KeyEvent e) keyReleased(KeyEvent e) keyTyped(KeyEvent e)
MouseEvent	pressing/releasing/clicking a mouse button, moving a mouse onto or away from a	MouseListener	mouseClicked(MouseEvent e) mouseEntered(MouseEvent e) mouseExited(MouseEvent e) mousePressed(MouseEvent e)

	component		mouseReleased(MouseEvent e)
MouseEvent	moving a mouse within a component while the button is up or down	MouseMotionListener	mouseDragged(MouseEvent e) mouseMoved(MouseEvent e)
ContainerEvent	Adding or removing a component to a container such as a panel	ContainerListener	componentAdded(ContainerEvent e) componentRemoved(ContainerEvent e)

So, if you want to handle a button press in your program, you need to write an **`actionPerformed()`**method:

```
public void actionPerformed(ActionEvent e) {
    //Do what needs to be done when the button is clicked
}
```

If you want to have something happen when the user presses a particular key on the keyboard, you need to write a **`keyPressed()`** method:

```
public void keyPressed(KeyEvent e) {
    //Do what needs to be done when a key is pressed
}
```

Once we decide which events we want to handle and then write our event handlers, we then need to ***register*** the event handler. This is like "plugging-in" the event handler to our window. In general, many applications can *listen for* events on the same component. So when the component event is generated, JAVA must inform everyone who is listening. We must therefore tell the component that we are listening for (or waiting for) an event. If we do not tell the component, it will not notify us when the event occurs (i.e., it will not call our event handler). So, when a component wants to signal/fire an event, it sends a specific message to all listener objects that have been registered (i.e., anybody who is "listening"). For every event, therefore, that we want to handle, we must write the listener (i.e., event handler) and also register that listener.

To help you understand why we need to do this, think of the Olympic games. There are various events in the Olympics and we may want to participate (i.e., handle) a particular event. Our training and preparation for the event is like writing the event handler code which defines what we do when the event happens. But, we don't get to participate in the Olympic games unless we "sign-up" (or register) for the events ... right ? So registering our event handlers is like joining JAVA's sign-up list so that JAVA informs us when the event happens and then allows our event handler to participate when the event occurs.

To **register** for an event (i.e., enable it), we need to merely add the listener (i.e., your event handler) to the component by using an **`addXXXListener()`** method (where XXX depends on the type of event to be handled). Here are some examples:

```
aButton.addActionListener(ActionListener anActionListener);
aJPanel.addMouseListener(MouseListener aMouseListener);
aJFrame.addWindowListener(WindowListener aWindowListener);
```

Here **`anActionListener`**, **`aMouseListener`** and **`aWindowListener`** can be instances of **any class that implements the specific Listener interface**.
So, for example, if you wanted to have your application handle a button press, you can make your application itself be the **ActionListener** as follows:

```
public class SimpleEventTest extends JFrame implements ActionListener {

    public SimpleEventTest(String name) {
        super(name);

        JButton aButton = new JButton("Hello");
        add(aButton);

        // Plug-in the button's event handler
        aButton.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(200, 200);
    }
```

```
    // Must write this method now since SimpleEventTest implements
the ActionListener interface
    public void actionPerformed(ActionEvent e) {
        System.out.println("I have been pressed");
    }

     public static void main(String[] args) {
         JFrame frame = new SimpleEventTest("Making a
 Listener");
         frame.setVisible(true);
     }
 }
```

You can also "unregister" from an event (i.e., disable the listener), by merely removing it using a **removeXXXListener()** method. Here are some examples:

```
aButton.removeActionListener(ActionListener anActionListener);
aJPanel.removeMouseListener(MouseListener aMouseListener);
aJFrame.removeWindowListener(WindowListener aWindowListener);
```

Why would you want to disable a listener ? If we don't want to use it, why even make one ? We will see later that we sometimes need to temporarily disable events while other events are being handled so as to avoid overlapping events, which can cause problems.

Adapter Classes:

Assume that we would like to handle a single event ... a **mouseClicked** event whenever someone clicks the mouse inside of our application's window. Recall from COMP1405/1005, that if a class implements an interface it MUST implement ALL of the methods listed in the interface. For example, the **MouseListener** interface is defined as follows:

```
public interface MouseListener {
    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

So, if we simple make our main application implement the **MouseListener** interface, then we will be forced to implement all 5 methods: mouseClicked, mouseEntered, mouseExited, mousePressed and mouseReleased !!!! We can however, merely write empty methods for these other 4 event types but this is a lot of extra code writing that just wastes time and makes the code more confusing:

```
public class MyApplication extends JFrame implements
MouseListener {
    ...
    public void mouseClicked(MouseEvent e) { /* Put your code
```

```
here */  };
    public void mouseEntered(MouseEvent e) {};
    public void mouseExited(MouseEvent e) {};
    public void mousePressed(MouseEvent e) {};
    public void mouseReleased(MouseEvent e {};
    ...
}
```

It does seem a little silly to have to write 4 blank methods when we do not even want to handle these other kinds of events. The JAVA guys recognized this inconvenience and solved it using the notion of ***Adapter*** classes. For each listener interface that has more than one method specified, there exists an adapter class with a corresponding name:

- MouseListener has **MouseAdapter**
- MouseMotionListener has **MouseMotionAdapter**
- DocumentListener has **DocumentAdapter**
- WindowListener has **WindowAdapter**
- ...and so on.
- ActionListener does NOT have an adapter class since it is only one method long.

These adapter classes are **abstract** JAVA classes that implement the interfaces they correspond to. However, even though they implement these interfaces ... their methods remain empty. For example, the **MouseAdapter** class looks like this:

```
public abstract class MouseAdapter implements MouseListener {
    public void mouseClicked(MouseEvent e) {};
    public void mouseEntered(MouseEvent e) {};
    public void mouseExited(MouseEvent e) {};
    public void mousePressed(MouseEvent e) {};
    public void mouseReleased(MouseEvent e {};
}
```

They are merely classes that are provided for convenience sake to help us avoid writing empty methods. So, we can simply write subclasses of these adapter classes, then we can take advantage of these blank methods through inheritance.

Well, we don't want to have to make our user interfaces subclass one of these adapter classes ... this would be bad since we would lose the freedom of creating our own arbitrary class hierarchies. Consider handling an event for dealing with a simple mouse click. We could make our own ***internal class*** to handle this event.

```
class MyClickHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent event) {
        System.out.println("Do Something fun");
    }
}
```

But this strategy creates an additional .java file. It may make our code more complicated, since we increase the number of source code files. Well, we can actually write this code within our GUI class itself, provided that we don't put the **public** modifier in front of this class definition. This makes it an internal class. When you compile a .java file that has internal classes, you will notice that you will have additional **.class** files for these additional internal classes (which will have a $ in their name).

To reduce clutter, JAVA allows another way for us to create inner classes which uses VERY strange syntax:

```
new MouseAdapter() {
    public void mouseClicked(WindowEvent event) {
       System.out.println("Do Something fun");
    }
}
```

This syntax actually creates an internal class as a subclass of **MouseAdapter**. The class has no name, it is considered to be an ***anonymous*** class. This code actually creates an instance of the anonymous class and returns it to us. It is weird syntax. We will see below how we can "embed" this code inside other code just like we use any other objects.

Summary of Making Your Own Event Handlers:

Let us now summarize the various ways (i.e., styles) that you can write your event handler code. Here are 4 ways ... you should understand them all:

1. **Make your class implement the specific interfaces needed:**
   - Advantages:
     - Simple
   - Disadvantages:
     - must write methods for ALL events in the interface.
     - can get messy/confusing if your class has many components that trigger the same events or if your class handles many different types of events.

```
public class YourClass extends JFrame implements MouseListener {

    // This line must appear in some method, perhaps the constructor
    ... {
        aComponent.addMouseListener(this);
    }

    // Some more of your code

    public void mouseClicked(MouseEvent e) { /* Put your code here */
};
    public void mouseEntered(MouseEvent e) { /* Put your code here */
};
    public void mouseExited(MouseEvent e) { /* Put your code here */
};
    public void mousePressed(MouseEvent e) { /* Put your code here */
};
    public void mouseReleased(MouseEvent e { /* Put your code here */
};

    // Put your other methods here
}
```

2. **Create a separate class that implements the interface:**
    - Advantages:
        - nice separation between your code and the event handlers.
        - class can be reused by other classes
    - Disadvantages:
        - can end up with a lot of classes and class files
        - can be confusing as to which classes are just event handler classes

```
public class YourClass extends JFrame {

    // This line must appear in some method, perhaps the constructor
    ... {
        aComponent.addActionListener(new MyButtonListener(this));
    }

    // Some more of your code
}

public class MyButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent theEvent) {
        // Do what needs to be done when the button is clicked
    }
}
```

3. **Create an "inner" class that implements the interface:**
    - Advantages:
        - nice separation between your code and the event handlers.
        - class can be reused in different situations within your class
        - Inner class has access to the "guts" of your class
    - Disadvantages:
        - can still end up with a lot of class names to remember

```
public class YourClass extends JFrame {

    // This line must appear in some method, perhaps the constructor
    ... {
        aComponent.addActionListener(new MyButonListener());
    }

    // Some more of your code

    class MyButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent theEvent) {
            // Do what needs to be done when the button is clicked
        }
    }
}
```

4. **Create an anonymous subclass of an Adapter class or a Listener interface.**

- Advantages:
  - nice and compact
  - do not need to come up with class names, reduces complexity
  - only need to handle one event instead of worrying about all events in the interface.
- Disadvantages:
  - the syntax takes a little "getting use to"
  - requires event handler code to be specified where listener is registered (unless helper methods are used)

```
public class YourClass extends JFrame {

    // This line must appear in some method, perhaps the constructor
    ... {
        aComponent.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent theEvent) {
                   // Do what needs to be done when the button is
clicked
                }
            }
        );
    }

    // Some more of your code
}
```

3.3 Handling ActionEvents with ActionListeners

In this section, we give various examples showing how to handle one or more **ActionEvents** from different kinds of objects:

Handling two button clicks

We have already seen how to handle a simple button press by writing an **ActionPerformed** method. Here is an application that shows how to handle events for two different buttons. We will make use of the **getActionCommand()** method for the **ActionEvent** class that allows us to determine the label on the button that generated the event. Take notice of the packages that need to be imported.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Handle2Buttons extends JFrame implements ActionListener {
    public Handle2Buttons(String title)
        super(title);

        JButton aButton1 = new JButton("Press Me");
        JButton aButton2 = new JButton("Don't Press Me");

        setLayout(new FlowLayout());
        add(aButton1);
        add(aButton2);

        // Indicate that this class will handle 2 button clicks
        // and that both buttons will go to the SAME event handler
        aButton1.addActionListener(this);
        aButton2.addActionListener(this);

          setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250,100);
    }

    // This is the event handler for the buttons
    public void actionPerformed(ActionEvent e) {
        // Ask the event which button was the source that generated it
        if (e.getActionCommand().equals("Press Me"))
            System.out.println("That felt good!");
        else
            System.out.println("Ouch! Stop that!");
    }

    public static void main(String args[]) {
        Handle2Buttons frame =  new Handle2Buttons("Handling 2 Button
Presses");
        frame.setVisible(true);
    }
}
```

Notice that the **getActionCommand()** method is sent to the **ActionEvent**. It returns a **String** containing the text that is on the button that generated the event. We then compare this string with the labels that we put on the buttons to determine which button was pressed. One disadvantage of this approach is that our event handler depends on the label associated with the button. Although this is safe in this particular example, there are many occasions when the label associated with a button could change (e.g., international applications). Therefore, we could use the **getSource()** method which returns the component (i.e., an Object) that raised the event instead of **getActionCommand()** to compare the actual button objects instead of the labels. To do this, we need to make two modifications. First, we need to store the buttons we create into instance variables and second, we need to compare the object that generated the event with these buttons using the identity (==) comparison.

```
    // We need to make the buttons instance variables and assign
    // them in the constructor so that we can access these objects
    // from within our event handler code.
    JButton   aButton1, aButton2;

    // Change the event handler to use getSource() to compare the actual
objects
    public void actionPerformed(ActionEvent e) {
        // Ask the event which button was the source that generated the event
        if (e.getSource() == aButton1)
            System.out.println("That felt good!");
        else
            System.out.println("Ouch! Stop that!");
    }
```

Another disadvantage of the previous example is that if more buttons (or other components that generate action events) are added, the number of "if-statements" in our handler will increase and become more complex, which may not be desirable. One way to handle this disadvantage (and the previous one as well) is by using anonymous classes. The following code would replace the code in the constructor that registers our frame subclass as a listener. The **actionPerformed** method of our class would no longer be required. Here, each button has its own event handler:

```
    aButton1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("That felt good!");
        }
    });
    aButton2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Ouch! Stop that!");
        }
    });
```

A Simple Slide Show

Now let us make a more interesting example that uses anonymous classes for two buttons. We will create a simple Slide Show application. We will create a window that has a **JPanel** which uses a **CardLayout** to represent the slides (one at a time) and then we will also add two arrow buttons to the window to rewind and forward the slides. Notice the following:

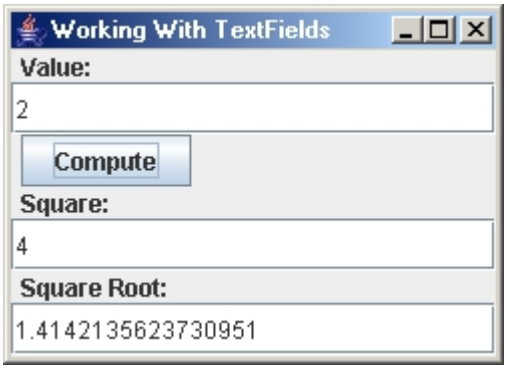
- the buttons we will use are the standard **BasicArrowButton** objects that are available in JAVA in the **`javax.swing.plaf.basic`** package.
- the **JPanel** and **CardLayout** are made into instance variables so that we can access them from our event handlers
- the main window is set to use a **FlowLayout**, the default was **BorderLayout**.
- we used **BorderFactory.createLineBorder()** to make a nice black border around our panel.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.basic.BasicArrowButton;
public class SlideShow extends JFrame {

    JPanel      slides;
    CardLayout  layoutManager;

    public SlideShow(String title) {
        super(title);

        setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

        // Create a JPanel with a CardLayout manager for the
slides
        slides = new JPanel();
        slides.setBackground(Color.WHITE);

slides.setBorder(BorderFactory.createLineBorder(Color.BLACK));
        slides.setLayout(layoutManager = new CardLayout(0,0));
        slides.add("trilobot.jpg", new JLabel(new
ImageIcon("trilobot.jpg")));
        slides.add("laptop.jpg", new JLabel(new
ImageIcon("laptop.jpg")));
```

```
        slides.add("satelite.jpg", new JLabel(new
ImageIcon("satelite.jpg")));
        slides.add("torch7.gif", new JLabel(new
ImageIcon("torch7.gif")));
        slides.add("SIGNIN.jpg", new JLabel(new
ImageIcon("SIGNIN.jpg")));
        add(slides);

        // Now add some slide show buttons for forward and
reverse
        JButton rev = new BasicArrowButton(JButton.WEST);
        add(rev);
        JButton fwd = new BasicArrowButton(JButton.EAST);
        add(fwd);

        // Set up the listeners using anonymous classes
        rev.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                layoutManager.previous(slides);
            }
        });

        fwd.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                layoutManager.next(slides);
            }
        });

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(220,300);
      }

    public static void main(String args[]) {
        SlideShow frame =  new SlideShow("Simple Slide Show");
        frame.setVisible(true);
    }
}
```

Working With JTextFields

Here is a new application that has a button and some text fields. One text field will hold an integer. When the button is pressed, it will compute and display (in two other text fields) the "square" as well as the "square root" of the value within the first text field. Note a few things about the code:



- When creating JTextFields, we can specify the initial content to be displayed (a string) as well as the maximum number of characters allowed to be entered in them (8, 16 and 20 in this example).
- We need to convert to and from Strings when accessing/modifying text field data
- We access/modify a text field's contents using **getText()** and **setText()**
- The code below will generate exceptions if a valid integer is not entered within the value text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HandleTextFieldAndButton extends JFrame {
    JTextField valueField, squareField, rootField;

    public HandleTextFieldAndButton(String title) {
        super(title);

        setLayout(new BoxLayout(this.getContentPane(),BoxLayout.Y_AXIS));

        // Add the value text field, along with a label
        add(new JLabel("Value:"));
        valueField = new JTextField("10", 8);
        add(valueField);

        // Add the compute button
        JButton aButton = new JButton("Compute");
        add(aButton);

        // Add the square text field, along with a label
        add(new JLabel("Square:"));
        squareField = new JTextField("0", 16);
        add(squareField);

        // Add the square root text field, along with a label
        add(new JLabel("Square Root:"));
        rootField = new JTextField("0", 20);
        add(rootField);
```

```
        // Handle the button click
        aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int value = Integer.parseInt(valueField.getText());
                squareField.setText("" + value * value);
                rootField.setText("" + Math.sqrt(value));
            }
        });

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250,180);
    }

    public static void main(String args[]) {
        HandleTextFieldAndButton frame = new
HandleTextFieldAndButton("Working With TextFields");
        frame.setVisible(true);
    }
}
```

Notes:

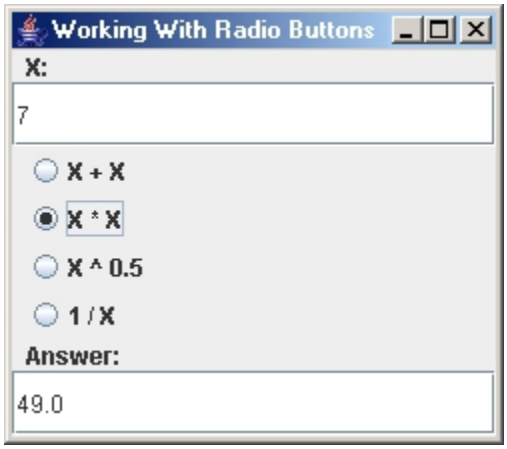
Although not done in our example here, we can actually handle an **ActionEvent** for a text field. An action event is generated when the user presses the ENTER key while typing in a text field. As with button clicks, you can handle this ENTER key press in the text field by writing an **actionPerformed()** method for the text field. In such a method, the **getActionCommand()** method will return the text inside the text field. We can also send the g**etSource()** method to the action event to get the text field itself and then get its text as follows:

```
((JTextField)e.getSource()).getText()
```

As we will discuss later, the style of coding that we are using in our example here is not "clean" since the button accesses the text field directly.

Handling RadioButtons

Let us modify the previous example by using radio buttons that allow us to decide which kind of operation we will do on the value entered in the text field. We will replace the **Compute** button with 4 radio buttons where each radio button, when clicked, will perform a different operation on the value from the text field and then display the result in the answer text field. Here are some interesting points about the code:



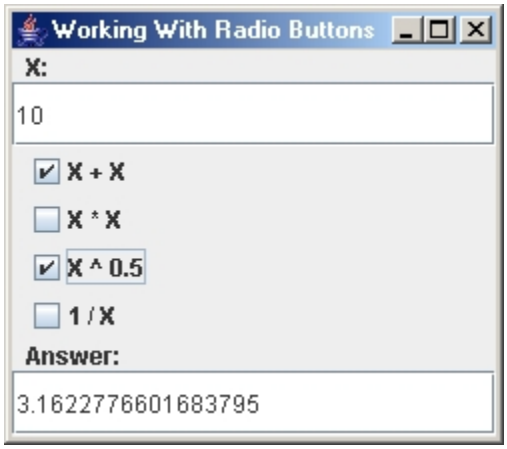
- All **JRadioButtons** go to the same event handler.
- The **JRadioButtons** are stored in an array, which is searched using a FOR loop to determine which one generated the event so that we could perform the desired operation.
- The **JRadioButtons** are added to a **ButtonGroup** as well, which ensures that JAVA allows only one to be "on" at a time. When created, we can specify with a boolean whether a particular button is to be "on" upon window startup.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HandleTextFieldWithRadioButtons extends JFrame
implements ActionListener {

    JTextField            valueField, answerField;
    JRadioButton[]        buttons;

    public HandleTextFieldWithRadioButtons(String title) {
        super(title);

        setLayout(new
BoxLayout(this.getContentPane(),BoxLayout.Y_AXIS));

        // Add the value text field, along with a label
        add(new JLabel("X:"));
        add(valueField = new JTextField("10", 8));

        // Add the operation type radio buttons to the window
and
        // also to a ButtonGroup so that one is on at a time
        ButtonGroup operations = new ButtonGroup();
        buttons = new JRadioButton[4];
```

```
        buttons[0] = new JRadioButton("X + X", false);
        buttons[1] = new JRadioButton("X * X", false);
        buttons[2] = new JRadioButton("X ^ 0.5", false);
        buttons[3] = new JRadioButton("1 / X", false);
        for (int i=0; i<4; i++) {
            add(buttons[i]);
            operations.add(buttons[i]);
            buttons[i].addActionListener(this);
        }

        // Add the answer text field, along with a label
        add(new JLabel("Answer:"));
        add(answerField = new JTextField("0", 16));

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250,220);
    }

    // Handle a radio button click
    public void actionPerformed(ActionEvent e) {
        int value = Integer.parseInt(valueField.getText());
        int buttonNumber = 0;
        for (buttonNumber=0; buttonNumber<4; buttonNumber++) {
            if (buttons[buttonNumber] == e.getSource())
                break;
        }
        double result=0;
        switch (buttonNumber) {
            case 0: result = value + value; break;
            case 1: result = value * value; break;
            case 2: result = Math.sqrt(value); break;
            case 3: result = 1 / (double)value; break;
        }
        answerField.setText("" + result);
    }

    public static void main(String args[]) {
        JFrame frame = new
HandleTextFieldWithRadioButtons("Working With Radio Buttons");
        frame.setVisible(true);
    }
}
```

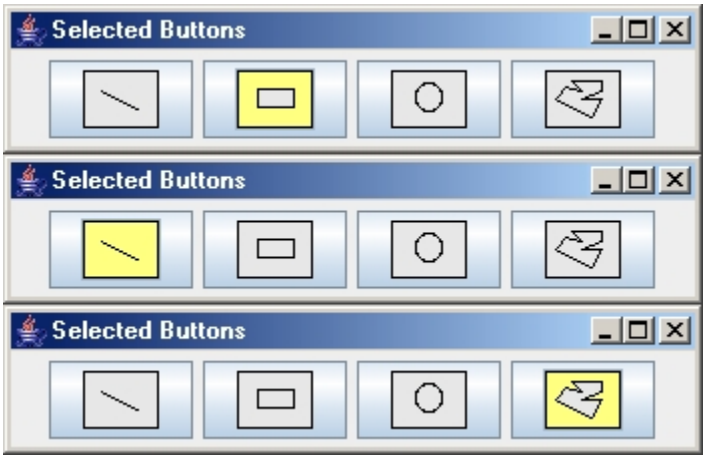
Note that for this example it seems appropriate to have our **JFrame** class act as a listener since the event handling code is the same for all components. Nothing is gained by using another class to handle the events.

Note as well that the **JCheckBox** works similar to the **JRadioButton**, except that normally **JRadioButtons** should have only one on at a time, while **JCheckBoxes** may normally have many on at a time. Here is how the window would look if **JCheckBoxes** were used instead (although keep in mind that in this application, it doesn't make sense to have more than one button on at a time.



Handling JButton Selections

In addition to radio buttons and checkbox buttons, **JButtons** themselves can maintain selected states. For example, we can create an application that allows one of several **JButtons** to be selected. As it turns out, **JButtons** have a **setSelectedIcon()** method that allows us to change the picture on a button when it is selected. Here is an example that makes 4 **JButtons** with icons on them which may be used to select a shape for drawing. When the user selects one of these buttons, the image changes on the button and so the button appears selected. All we have to do is use the **setSelected()** and **getSelected()** methods to change or query the button's state.




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class SelectedButtonsExample extends JFrame implements
ActionListener {
```

```
    private JButton[]  buttons;

    public SelectedButtonsExample(String title) {
        super(title);

        setLayout(new FlowLayout());

        ButtonGroup group = new ButtonGroup();
        buttons = new JButton[4];
        for (int i=0; i<4; i++)  {
            buttons[i] = new JButton(new ImageIcon("button" +
(i+1) + ".gif"));
            buttons[i].setSelectedIcon(new ImageIcon("button" +
(i+1) + "b.gif"));
            buttons[i].setRolloverEnabled(false);
            buttons[i].addActionListener(this);
            add(buttons[i]);
            group.add(buttons[i]);
        }

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(350,75);
    }

    public void actionPerformed(ActionEvent e)  {
        for (int i=0; i<4; i++)
            buttons[i].setSelected(e.getSource() ==
buttons[i]);
    }

    public static void main(String args[]) {
        SelectedButtonsExample frame =  new
SelectedButtonsExample("Selected Buttons");
        frame.setVisible(true);
    }
}
```

Notice that we did not use a **ButtonGroup** to ensure that only one button is selected by itself. That is because **JButtons** do not work with **ButtonGroups**. Instead, we had to do everything manually. If we wanted to allow multiple buttons on at a time, we would merely change the action performed method to be:

```
public void actionPerformed(ActionEvent e) {
    JButton src = (JButton)e.getSource();
    src.setSelected(!src.isSelected());
}
```

This would allow the buttons to be toggled on and off individually. Notice one good thing, we don't have to worry about changing the icon. It is done automatically for us when we set the button to be selected or not.

Notice that we did a **setRolloverEnabled(false)** for our buttons. This is because JAVA, by default, has a default rollover enabled value of **true** for the buttons, which redraws our buttons whenever the mouse passes over them. In this case, a completely separate icon may be used. So, we can modify our code to have our image show when the mouse rolls over the button instead of when we click it by setting this icon and enabling the rollover effect.

We would need to use these methods to accomplish this: **setRolloverIcon()** and **setRolloverSelectedIcon()**.

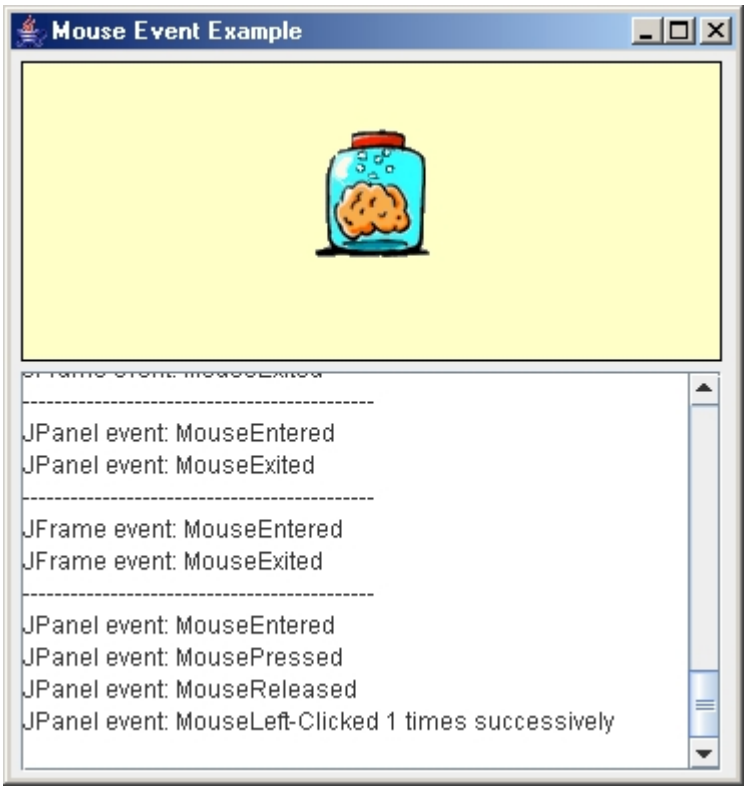
3.4 Handling MouseEvents with MouseListeners

In this section, we talk about mouse events. Mouse events are typically used in graphics applications (as we'll see later in a graph editing application). There are many kinds of mouse events as we have shown in an earlier table:

- **`mouseClicked`** - one of the mouse buttons has been both pressed and released within the same component.
- **`mouseEntered`** - the mouse cursor has entered the component's area.
- **`mouseExited`** - the mouse cursor has left the component's area.
- **`mousePressed`** - a mouse button has been pressed.
- **`mouseReleased`** - a mouse button has been released.

Example using Mouse Listeners

In this example we create two components: a yellow **JPanel** and a white **JTextArea** within a **JScrollPane**. Both the **JFrame** itself as well as the **JPanel** will respond to all **MouseEvents** and display an appropriate message within the text area. We will also add an **ImageIcon** to the JPanel as a **JLabel** and we will move it around depending on where the user presses the mouse. We make use of the following **MouseEvent** methods to get information about mouse presses:



- **`getClickCount()`** - returns the number of successive mouse clicks
- **`getButton()`** - returns the button that was pressed (1 = left, 2 = middle, 3 = right)
- **`getX()`** - returns the x coordinate of the mouse location w.r.t. top left corner of component.
- **`getY()`** - returns the x coordinate of the mouse location w.r.t. top left corner of component.
- **`getPoint()`** - returns the (x,y) point of the mouse location w.r.t. top left corner of component.

Here is the code for the application:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class WorkingWithMouseEvents extends JFrame implements
MouseListener {
     JPanel     blankArea;
     JTextArea  textArea;
     JLabel     movableImage;
     Class      latestComponent;
```

```
     public WorkingWithMouseEvents(String title) {
        super(title);

        setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));

        // Create a yellow JPanel
        blankArea = new JPanel();
        blankArea.setLayout(null);
        blankArea.setBackground(new Color(255,255,200));
        blankArea.setOpaque(true);

blankArea.setBorder(BorderFactory.createLineBorder(Color.black));
        blankArea.setPreferredSize(new Dimension(350, 150));
        add(blankArea);

        // Add an image to the JPanel
        blankArea.add(movableImage = new JLabel(new
ImageIcon("brain.gif")));
        movableImage.setSize(80,80);
        movableImage.setLocation(100,100);

        // Create a text area to display event information
        textArea = new JTextArea();
        textArea.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(textArea);
        scrollPane.setPreferredSize(new Dimension(350, 200));
        add(scrollPane);

        //Register for mouse events on the JPanel AND the JFrame
        blankArea.addMouseListener(this);
        addMouseListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(370,390);
    }

    // Handle the mouse events (pressed, released, entered,
exited & clicked)
    public void mousePressed(MouseEvent event) {
        addToTextArea("MousePressed", event);
        if (event.getComponent().getClass() != this.getClass())
            movableImage.setLocation(event.getX()-40,
event.getY()-40);
    }

    public void mouseReleased(MouseEvent event) {
        addToTextArea("MouseReleased", event);
    }

    public void mouseEntered(MouseEvent event) {
        addToTextArea("MouseEntered", event);
    }
```

```
        public void mouseExited(MouseEvent event) {
            addToTextArea("MouseExited", event);
        }

        public void mouseClicked(MouseEvent event) {
            String s;
            if (event.getButton() == 3)
                s = "Right";
            else s = "Left"; // Ignores the middle button case
            addToTextArea("Mouse" + s + "-Clicked " +
    event.getClickCount() +
                                " times successively ", event);
        }

        // Append the specified event-specific text to the text area
        private void addToTextArea(String eventDescription,
    MouseEvent event) {
            if (latestComponent != event.getComponent().getClass())
                textArea.append("-----------------------------------
    --------\n");
            latestComponent = event.getComponent().getClass();
            if (latestComponent == this.getClass())
                textArea.append("JFrame event: ");
            else
                textArea.append("JPanel event: ");
            textArea.append(eventDescription + "\n");
        }

        public static void main(String args[]) {
            JFrame frame = new WorkingWithMouseEvents("Mouse Event
    Example");
            frame.setVisible(true);
        }
    }
```

How can we modify the code to only handle clicked events if it was a double-click ?

```
public void mouseClicked(MouseEvent event) {
    if (event.getClickCount() == 2)
       // do something
}
```

3.5 Key Press Events

Every Component in JAVA can listen for ***KeyEvents***. **KeyEvents** are generated when the user presses, releases or types a key while in a component. In order for the event to be generated, the component MUST have the focus. The ***focus*** represents the current component that is selected (e.g., we all understand how the TAB key moves the focus from one component to another in many windows applications). Thus, if a particular component is listening for a key press, but that component does not have the focus, then no events are generated.

We can have a component listen for **KeyEvents** by adding a **KeyListener** with the **addKeyListener()** method. Inside these listeners, we can determine which key was pressed by examining the **KeyEvent** object itself. The **KeyEvent** class has a bunch of static constants that represent all the keys on the keyboard. These constants all begin with **VK_** and you can look in the JAVA API to get the exact names. Here are a few:

- **VK_A**, **VK_B**, **VK_C**, ..., **VK_Z**
- **VK_SHIFT**, **VK_ALT**, **VK_CONTROL**, **VK_ENTER**
- **VK_DOWN**, **VK_UP**, **VK_LEFT**, **VK_RIGHT**
- etc...

We send the **getKeyCode()** message to the **KeyEvent** to get back the code representing the key that was pressed. We then compare the code to one of these constants. Since every key press generates an event, if we want to detect multiple keys pressed at the same time, me must make use of both **keyPressed()** and **keyReleased()** listeners and keep track by ourselves as to which key has been pressed. There is also a **keyTyped()** event which can detect the entering of a Unicode character. ~~Often a **keyTyped()** event is synonymous with a key press/release sequence (kinda like a **mouseClicked** event).~~

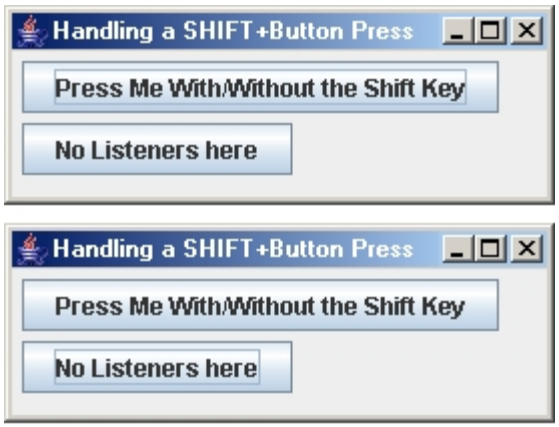
The **keyTyped()** event is generated along with a **keyPressed()** event, whenever letter/number/symbol keys are pressed. However, the **keyTyped()** event is not generated when the control-related keys are pressed (e.g., shift, alt, ctrl, caps-lock, insert, home, end, pageup/down, break, arrow keys, function keys etc...) In this case, just the **keyPressed()** and **keyReleased()** events are generated. Odly enough, some control-related keys do generate **keyTyped()** events (e.g., esc, del), some keys generate only a **keyReleased()** event (e.g., print screen) and some keys do not generate events at all (e.g., tab, function key)!!!!! Check to make sure that the key you want to handle behaves the way you want it to.

Also, we can combine other listeners with key presses ... for example, if we want to detect a SHIFT-CLICK operation.

Here is an example with code that detects three things:

- Pressing the 'A' key by itself
- Pressing the 'SHIFT' and 'A' keys together
- Pressing the 'SHIFT' key and pressing a button together

Note that there are two buttons. The bottom one is not hooked up to the listeners so when it has the focus, no key events are generated. The image on the top (below) shows the first button having the focus (notice the thin gray line around the text of the button). The bottom image shows the non-listener button with the focus.




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ShiftButtonTest extends JFrame implements ActionListener,
KeyListener {
    private boolean    shiftPressed;

    public ShiftButtonTest(String title) {
        super(title);
        setLayout(new FlowLayout(5));

        JButton aButton = new JButton("Press Me With/Without the Shift Key");
        JButton bButton = new JButton("No Listeners here");
        add(aButton);
        add(bButton);

        shiftPressed = false;

        //Indicate that this class will handle the button click
        aButton.addActionListener(this);
        aButton.addKeyListener(this); // Change aButton to this if you want
to ignore focus

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,100);
    }

    //This is the event handler for the button
    public void actionPerformed(ActionEvent e) {
        if (shiftPressed)
            System.out.println("You SHIFTED Me!!");
        else
            System.out.println("You did not SHIFT Me.");
    }
```

```
    public void keyPressed(KeyEvent e)  {
        if (e.getKeyCode() == KeyEvent.VK_SHIFT)
            shiftPressed = true;
        else if (e.getKeyCode() == KeyEvent.VK_A)   {
                if (shiftPressed)
                       System.out.println("You pressed the [SHIFT]+[A]
keys");
                else
                       System.out.println("You pressed the [A] key");
        }
    }
    public void keyReleased(KeyEvent e)  {
        if (e.getKeyCode() == KeyEvent.VK_SHIFT)
            shiftPressed = false;
    }
    public void keyTyped(KeyEvent e)  {
        // Get and display the character for each key typed
       System.out.println("Key Typed: " + e.getKeyChar());
    }

    public static void main(String args[]) {
        ShiftButtonTest frame = new ShiftButtonTest("Example: Handling a
SHIFT+Button Press");
        frame.setVisible(true);
    }
}
```

If you do not want this "focus-oriented" behavior (e.g., perhaps you want to listen for a particular key press regardless of which component has been selected) you can have the **JFrame** listen for the key press. In this case, you must "disable" the focus ability for all the components on the window (but not the JFrame itself). In our example, we would replace the line:

```
aButton.addKeyListener(this);
```

with the following lines that disallow the buttons to have the focus:

```
this.addKeyListener(this);
aButton.setFocusable(false);
bButton.setFocusable(false);
```

Be aware however, that this disables the "normal" behavior for the window and will prevent standard use of the TAB key to traverse between components in the window.

3.6 Proper Coding Style for Component Interaction

Recall that before designing an application, we must distinguish between the **model** and the **interface**.

Recall as well that the ***model*** is:

- the underlying "meat" of the application (represents "business/problem domain" logic)
- corresponds to all classes and objects that do not deal with the user interface appearance or operation.

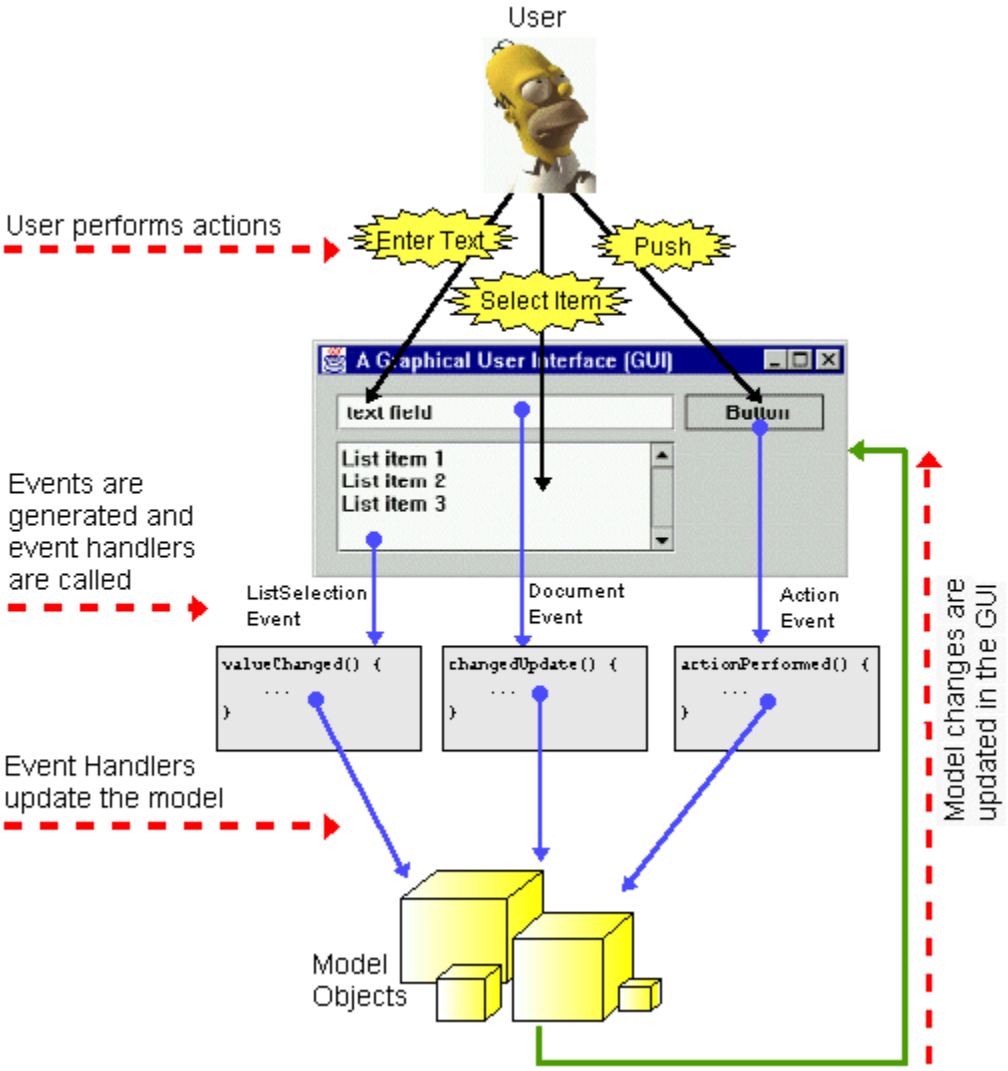
The ***GUI (Graphical User Interface)*** is:

- the portion of your code that deals with the appearance of the application and the interaction between the interface components.

It is IMPORTANT to keep the model separate from the interface. Also, with respect to the GUI, we need to have a "good" understanding of how the components of the interface will work and interact with each other. Let us see if we can explain how the components interact. We need to determine ALL of the following:

1. What events do we need to handle for each component ?
2. What should happen when each event is triggered ?
3. How do the events affect the model ?
4. How do we make changes to the model ?
5. How do these model changes affect the appearance of the interface ?

Recall that a user interface works as follows (based on what is called the *Observer Pattern*).



There are two questions regarding the updating stage of the components:

- When do we do this updating ? (i.e., where in our code)
- Which components need updating ?

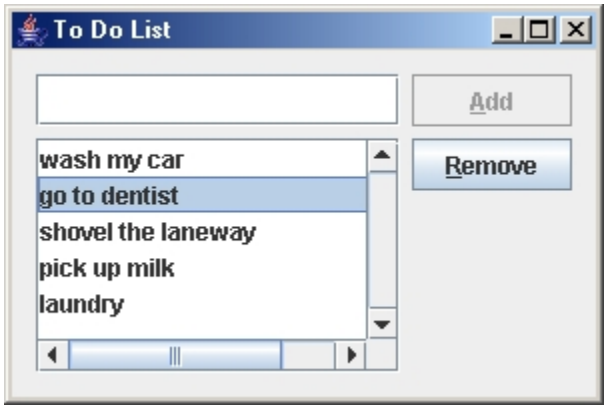
Well, as a rule of thumb, we should do an update whenever we make any change to the model. When we go to do the update, if we know our code very well, we can determine which components will need updating based on the particular change to the model. However, in case we have a complex application, we may not know which ones need updating, so we can take the simple brainless approach and just update everything. We often simply write an **update()** method, where we update all the components (i.e., a kind of ***global*** update is performed). We call this method whenever the model changes.

So, REMEMBER the two VERY important things that you should normally do in every event handler:

1. **Change the model**
2. **Call** update()

Example:

Let us now build the following application which represents a list of "things to do":



The application will work as follows:

- The model consists of a collection (stored in a list) of items. In our example, the items will be Strings.
- The user can add new items to the list by typing the new item in the text field and clicking the **Add** Button.
- Items are removed from the list by selecting an item from the list and clicking the **Remove** Button.

In this "to do list" application, the **model** is simply the collection (e.g., a Vector) of things to do (i.e. Strings). So we don't need to make a model class in this simple example. What about the GUI ? First, we determine how the interface should ***react*** to user input. We must determine which events are necessary to be handled. The events and their consequences are as follows:

AddButton

`actionPerformed()` - Should take text from text field and add it to the list.

RemoveButton

`actionPerformed()` - Should determine selected item from the list and then remove it.

TextField

Nothing for now. We will add some behavior here later.

List

Nothing for now. We will add some behavior here later.

Now what about the model updating ? How do these events change the model ? How does the model then change the window again ?

- **Adding** an item should cause a new entry to be added to the list (i.e., model). Then we must show these changes in the list.
- **Removing** an item also changes the model and we should show the changes right away in the list as well.

Refreshing the user interface to show the changes in the model is called ***updating.***

Let us now look at a basic "working" application. We will handle the Adding and Removing of items from the list. The highlighted code below indicates the code required for handling the events from the buttons and updating the interface:

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class  TodoListFrame extends JFrame {
    private JTextField newItemField;
    private JList      itemsList;
    private JButton    addButton;
    private JButton    removeButton;

    private Vector<String>  items;  // The model

    public TodoListFrame() {
        this(new Vector<String>());
    }
    public TodoListFrame(Vector<String> todoEntries) {
        super("To Do List");
        items = todoEntries;

        // ...
        // The code for building the window has been omitted
        // ...


        // Add listeners for the buttons and then enable them
        addButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                addButtonEventHandler();
            }
        });
        removeButton.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
                removeButtonEventHandler();
            }
        });


        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,200);


        update();


    }

    // Event Handler for the Add button
    private void addButtonEventHandler() {
        items.add(newItemField.getText());
        update();
    }


    // Event Handler for the Remove button
    private void removeButtonEventHandler() {
        items.remove((String)itemsList.getSelectedValue());
        update();
    }
    // Update all the components
    private void update() {
        itemsList.setListData(items);
    }

    public static void main(String[] args) {
        // Set up the items to be put into the list
        Vector<String> todoItems = new Vector<String>();
        todoItems.add("wash my car");
        todoItems.add("go to dentist");
        todoItems.add("shovel the laneway");
        todoItems.add("pick up milk");
        todoItems.add("laundry");

        TodoListFrame frame = new TodoListFrame(todoItems);
        frame.setVisible(true);
    }
}
```

Notice:

- we made a single **`update()`** method that is called from the Add and Remove button event handlers as well as when the window is first opened.
- the update really just updated the data in the **JList** to reflect the changes in the model.

So, REMEMBER the two VERY important things that you should normally do in your **update()** method:

1. **Read the model's information**
2. **Change the "look" of the interface components**

There are two problems with the application:

1. When no text is in the text field and **Add** is pressed, a blank item is added.
2. When no item is selected from the list and a **Remove** is done, our code tries to remove a **null** item from the model. Since the model is a **Vector**, and the remove method for vectors handles this attempt with grace (i.e., no exception), then it is not really a problem. However, what if someone changes the underlying model to be something other than a vector ? We should fix this.

How can we fix these ? First check if there is any text before doing the **Add**. If there is none, don't add. The change occurs in the event handler for the **Add** button. Here is the changed code:

```
private void addButtonEventHandler() {
   if (newItemField.getText().length()
> 0) {
        items.add(newItemField.getText());
        update();
   }
}
```

For the remove problem, we would like to have a way of determining whether or not anything was selected from the list. To do this, we merely ask if the selected list value is **null**:

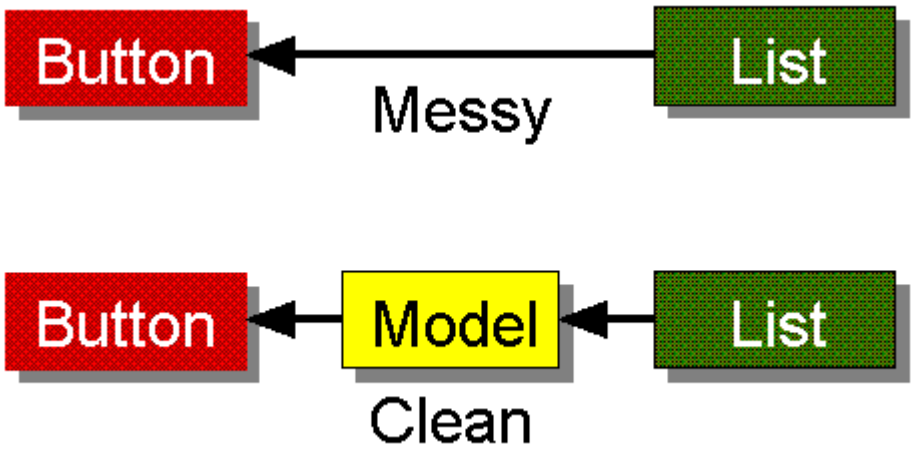
```
private void removeButtonEventHandler() {
   if (itemsList.getSelectedValue() !=
null) {

items.remove((String)itemsList.getSelectedValue());
        update();
   }
}
```

We have fixed the problems ... but now we have a messy situation. It seems that the **JButtons** MUST know about the **JList** component. The **JButtons** are somehow "tied" with the **JList** so

that if the **JList** is removed and perhaps replaced by something else, we must go into the **JButton** event handler and make changes. This is "messy".

- code is not easily maintained when many components rely on other components
- components need to know exactly how they affect other components

A better way to solve these problems is to make the list selection a part of the model. We would like to use the model as a kind of "middle man" between all component interaction so that this "dependence" between components is severed.



So ... we will keep track of the item that was selected and this will be part of our model.
Of course this means that we will have to handle the *selection* event for the **JList** component.

Here is what we need to add/change:

1. Add an instance variable to store the selected item:

```
private String  selectedItem; // Part of the model
```

2. Modify the constructor to initially select a list item if there is one available, and also add a list selection listener for whenever someone makes a selection from the list:

```
public TodoListFrame(Vector<String> todoEntries) {
    super("To Do List");
    items = todoEntries;


   if (items.size() > 0)
       selectedItem = (String)items.firstElement();
   else
       selectedItem = null;


    // ... Some code has been omitted ...
```

```
        // Add listeners for the buttons and then enable them
        addButton.addActionListener(...);
        removeButton.addActionListener(...);


        itemsList.addListSelectionListener(new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent e) {
                listSelectionEventHandler();
            }
        });


        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,200);
        update();
    }
```

3. Write the **listSelectionEventHandler()** so that the new instance variable is updated to reflect the latest selection made:

```
    private void listSelectionEventHandler() {
        selectedItem = (String)itemsList.getSelectedValue();
        update();
    }
```

4. Modify the **Remove** button event handler to use the new **selectedItem** variable now. We must make sure to set the **selectedItem** to **null** after an item is removed, since the list will not have anything selected in it anymore:

```
    private void removeButtonEventHandler() {

        if (selectedItem != null) {
            items.remove(selectedItem);
            selectedItem == null;
             update();
        }
    }
```

5. Modify the **update()** method to make sure that the **selectedItem** variable always matches the item selected from the list:

```
    private void update() {
        itemsList.setListData(items);

        itemsList.setSelectedValue(selectedItem, true);


    }
```

At this point, we have a strange bug in our application. It seems that we are unable to actually select anything from our **JList** now ! The problem is that our **update()** method calls **setSelectedValue()** which changes the contents of the list. This generates an internal **valueChanged()** event ... which is the event that we are handling. Hence, when we do a list

selection, our event handler is called, which itself calls **update()**. Then update generates another **valueChanged()** event which again calls our handler and **update()** again. Really, this is an endless loop. JAVA is able to deal with this problem without generating exceptions, but it does not give us desirable results in that we cannot really select anything from the list ;).

The simplest and most logical solution is to disable the list selection listener while updating and then re-enable it afterwards. To do this, we will need the actual listener object and de-register it at the beginning of the **update()** method, then re-register it afterwards. Here are the steps:

1. Declare the following instance variable:

```
private ListSelectionListener   itemsListListener;
```

2. Store the ListSelectionListener that was created in our constructor:

```
itemsList.addListSelectionListener(itemsListListener =
    new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e) {
            listSelectionEventHandler();
        }
    }
);
```

3. Disable and then re-enable the listener in our **update()** method:

```
private void update() {
   itemsList.removeListSelectionListener(itemsListListener);
    itemsList.setListData(items);
    itemsList.setSelectedValue(selectedItem, true);
   itemsList.addListSelectionListener(itemsListListener);
}
```

The application now works and has nice clean code.

We have prevented the **Add** button from doing anything when there is no text in the text field and the **Remove** button from doing anything when there is nothing selected. It is best to let the user know that these buttons will not work under these circumstances. The proper way of doing this is to disable the buttons at these times. Let us make these changes now. We will make use of the **`setEnabled()`** method for buttons which enables or disables the button according to a given boolean.

Where do we write the code for disabling these buttons ? Well, does it have to do with functionality or with appearance ?
After some thought, you realize that this is a "cosmetic" issue and that it has to do with the "look" of the buttons. Hence, we should make these changes in the **update()** method.

Disabling the **Remove** button is easy. Just add the following line to the **update()** method:

```
removeButton.setEnabled(selectedItem != null);
```

For the **Add** button, we can add a similar line:

```
addButton.setEnabled(newItemField.getText().length() > 0);
```

There is a small problem. When the interface starts up, the text field is empty and so the **Add** button is disabled. That's good. But when the user starts typing in the text field, there is then text in the text field but the **Add** button remains enabled. The problem is that **update()** is not being called unless an event occurs. So we need to have some kind of event for when the user types text in the text field. So we will need to make some changes. In addition, this approach to enabling the **Add** button results in a dependency on there being a text field. We should create another instance variable to indicate whether or not there is any text in the text field. We can just use a boolean, but we may as well keep the whole item that is in there instead of just a flag. First we need to make the following additions:

```
// Add this as an instance variable
private String  newItem;

// Add this to the constructor
newItem = "";
```

Now, we could use handle **ActionEvents** for the **TextField**, but these events only occur when the user presses **Enter** within that field. Instead, we will make use of something called a ***DocumentListener***. Every **JTextField** has a ***document*** object associated with it that can be obtained with **`getDocument()`**. We can then add the listener to this object. This way, we can handle events that occur whenever the text changes (character by character) even if no **Enter** key is pressed. We need to make the following changes to our code:

```
// Declare this instance variable at the top
private DocumentListener   newItemFieldListener;

// Add this to the constructor
newItemField.getDocument().addDocumentListener(newItemFieldListener =
    new DocumentListener() {
        public void changedUpdate(DocumentEvent theEvent) {
            handleTextFieldEntry();
        }
        public void insertUpdate(DocumentEvent theEvent) {
            handleTextFieldEntry();
        }
        public void removeUpdate(DocumentEvent theEvent) {
            handleTextFieldEntry();
        }
    }
);

// Add this event handler for the text field
private void handleTextFieldEntry() {
    newItem = newItemField.getText();
    update();
}
```

Notice that there are three events that may be generated by the **DocumentListener**. These correspond to typing in text, inserting and removing (i.e., paste/cut). Notice that despite the particular edit change in the text field, all three events call our helper method which simply sets the **newItem** variable to match the contents of the text field.

Of course, we will want to now modify the event handler for the **Add** button to make use of the **newItem** field. We will also select the item that was just added. This is not necessary, but it is a nice form of "feedback" for the user:

```
private void addButtonEventHandler() {
    if (newItem.length() > 0) {
        items.add(newItem);
        selectedItem = newItem; // select the newly added item
        update();
    }
}
```

We should also modify our **update()** method so that the **Add** button uses the **newItem** variable now:

```
addButton.setEnabled(newItem.length() > 0);
```

Now our code should work fine.

One last feature that we will add is to clear out the text field AFTER we add a new item. Well, to do this, we will have to set the **newItem** to "" in the **Add** button handler as follows:

```
private void addButtonEventHandler() {
    if (newItem.length() > 0) {
        items.add(newItem);
        selectedItem = newItem;
        newItem = "";
        update();
    }
}
```

Notice however, that at this point, the **newItem** variable will be "", but there will still be some contents in the text field ... so they are not in agreement. To fix this, we will have to actually clear out the text field contents. This, of course, has to do with the appearance of the text field, so we could try placing the following code within the **update()** method:

```
newItemField.setText(newItem);
```

But, we have a small problem. If we were to run our code right now, we would notice a bug when we tried to type into the text field. Our code would generate the following exception:

```
java.lang.IllegalStateException: Attempt to mutate in
notification
```

JAVA version 1.4 and onward, however, will not allow us to set or modify the contents of the **JTextField** while we are handling one of its document events. So, we will need to make the following changes:

1. avoid setting the text field's text within the **update()** whenever we call **update()** from a **DocumentListener**
2. remove/add the document listener at the start/end of the **update()** method.

To do this, we will make a special **update()** method. In fact, we will split up our **update()** method as follows:

```
// Update called by Document Listeners directly
private void update(boolean calledFromTextField) {
    itemsList.removeListSelectionListener(itemsListListener);
    newItemField.getDocument().removeDocumentListener(newItemFieldListener);

    itemsList.setListData(items);
    itemsList.setSelectedValue(selectedItem, true);
    removeButton.setEnabled(selectedItem != null);
    addButton.setEnabled(newItem.length() > 0);
    if (!calledFromTextField)  newItemField.setText(newItem);

    itemsList.addListSelectionListener(itemsListListener);
    newItemField.getDocument().addDocumentListener(newItemFieldListener);
}

// Update used by all the event handlers, except Document event handlers
private void update() {
    update(false);
}
```

The make the following change in the **handleTextFieldEntry()** method:

```
private void handleTextFieldEntry() {
    newItem = newItemField.getText();
    update(true); // since we called from this document
listener
}
```

Note that all event handlers will call the usual **update()** method (which will set the text in the text field), but the DocumentListeners will call **update(true)** so as to avoid setting the text illegally. Thus, when we press the **Add** button and set the **newItem** variable to "", the call to **update()** at the end of the event handler will ensure that the text field's contents are set to "". Meanwhile, if we make changes to the text field directly, we will not be updating the text field appearance, as it does not need updating since it will always have the same contents as the **newItem** variable anyway.

Now the last improvement is within the **update()** method. When we make changes to our code by adding or removing components, we must go to the update method and make changes. It is difficult to determine what code pertains to which components. We can extract this update code so that we make separate methods such as **updateTextField()**, **updateList()**, and **updateButtons()**. These will do the corresponding updates for the individual components. This way, when we modify or remove a component, it is clear as to what code should be modified/removed. This alternative method also allows us to update only those components that have changed (not all). This is good if the interface becomes slow in drawing the components.

We can speed everything up by only updating necessary components. We can also extract the code for disabling/enabling the listeners into separate methods as well.

The final completed code is shown below (the class name has been changed to **ToDoListFrame2**):

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class ToDoListFrame2 extends JFrame {
    private JTextField   newItemField;
    private JList        itemsList;
    private JButton      addButton;
    private JButton      removeButton;

    // Listeners that need to be disabled/enabled during update()
    private ListSelectionListener    itemsListListener;
    private DocumentListener         newItemFieldListener;

    private Vector<String>  items;        // The model
    private String          selectedItem; // item selected in the list
    private String          newItem;      // String contained in text field

    public ToDoListFrame2() {
        this(new Vector<String>());
    }

    public ToDoListFrame2(Vector<String> todoEntries) {
        super("To Do List");
        items = todoEntries;

        if (items.size() > 0)
            selectedItem = (String)items.firstElement();
        else
            selectedItem = null;

        newItem = ""; // nothing in the text field yet

        initializeComponents();

        // Add listeners for the buttons, list and text field
        addButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                addButtonEventHandler();
            }
        });
        removeButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                removeButtonEventHandler();
            }
        });
```

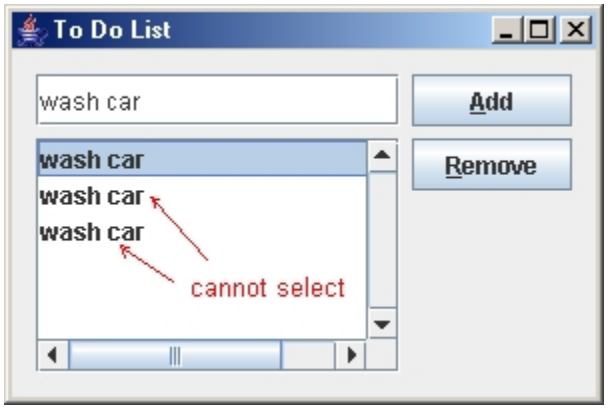
```
        itemsList.addListSelectionListener(itemsListListener =
            new ListSelectionListener() {
                  public void valueChanged(ListSelectionEvent e) {
                      listSelectionEventHandler();
                  }
            }
        );
        newItemField.getDocument().addDocumentListener(newItemFieldListener =
            new DocumentListener() {
                public void changedUpdate(DocumentEvent theEvent) {
                    handleTextFieldEntry();
                }
                public void insertUpdate(DocumentEvent theEvent) {
                    handleTextFieldEntry();
                }
                public void removeUpdate(DocumentEvent theEvent) {
                    handleTextFieldEntry();
                }
            }
        );

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,200);

        update();
    }

    // Build the frame by adding all the components
    private void initializeComponents() {
        GridBagLayout layout = new GridBagLayout();
        GridBagConstraints constraints = new GridBagConstraints();
        setLayout(layout);

        newItemField = new JTextField();
        constraints.gridx = 0;
        constraints.gridy = 0;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(12, 12, 3, 3);
        constraints.weightx = 1;
        constraints.weighty = 0;
        layout.setConstraints(newItemField, constraints);
        add(newItemField);

        addButton = new JButton("Add");
        addButton.setMnemonic('A');
        constraints.gridx = 1;
        constraints.gridy = 0;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.insets = new Insets(12, 3, 3, 12);
        constraints.anchor = GridBagConstraints.NORTHWEST;
        constraints.weightx = 0;
        constraints.weighty = 0;
        layout.setConstraints(addButton, constraints);
        add(addButton);
```

```
        itemsList = new JList();
        itemsList.setPrototypeCellValue("xxxxxxxxxxxxxxxxxxxxxxxxxxxx");
        JScrollPane scrollPane = new JScrollPane(itemsList,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        constraints.gridx = 0;
        constraints.gridy = 1;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(3, 12, 12, 3);
        constraints.weightx = 1;
        constraints.weighty = 1;
        layout.setConstraints(scrollPane, constraints);
        add(scrollPane);

        removeButton = new JButton("Remove");
        removeButton.setMnemonic('R');
        constraints.gridx = 1;
        constraints.gridy = 1;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.insets = new Insets(3, 3, 0, 12);
        constraints.anchor = GridBagConstraints.NORTH;
        constraints.weightx = 0;
        constraints.weighty = 0;
        layout.setConstraints(removeButton, constraints);
        add(removeButton);
    }

    // Event Handler for the Add button
    private void addButtonEventHandler() {
        if (newItem.length() > 0) {
            items.add(newItem);
            selectedItem = newItem; // select the newly added item
            newItem = "";           // clear the text
            update();
        }
    }

    // Event Handler for the Remove button
    private void removeButtonEventHandler() {
        if (selectedItem != null) {
            items.remove(selectedItem);
            selectedItem = null;
            update();
        }
    }

    // Event Handler for List Selection
    private void listSelectionEventHandler() {
        selectedItem = (String)itemsList.getSelectedValue();
        update();
    }

    // Handler for entering text in the text field
    private void handleTextFieldEntry() {
        newItem = newItemField.getText();
        update(true);
    }
```

```
    // Update all the components
    private void update(boolean calledFromTextField) {
        disableListeners();

        updateList();
        updateButtons();
        if (!calledFromTextField)
            updateTextField();

        enableListeners();
    }

    private void update() {
        update(false);
    }

    private void disableListeners() {
        itemsList.removeListSelectionListener(itemsListListener);

newItemField.getDocument().removeDocumentListener(newItemFieldListener);
    }
    private void enableListeners() {
        newItemField.getDocument().addDocumentListener(newItemFieldListener);
        itemsList.addListSelectionListener(itemsListListener);
    }
    private void updateList() {
        itemsList.setListData(items);
        itemsList.setSelectedValue(selectedItem, true);
    }
    private void updateButtons() {
        removeButton.setEnabled(selectedItem != null);
        addButton.setEnabled(newItem.length() > 0);
    }
    private void updateTextField() {
        newItemField.setText(newItem);
    }

    public static void main(String[] args) {
        // Set up the items to be put into the list
        Vector<String> todoItems = new Vector<String>();
        todoItems.add("wash my car");
        todoItems.add("go to dentist");
        todoItems.add("shovel the laneway");
        todoItems.add("pick up milk");
        todoItems.add("laundry");

        ToDoListFrame2 frame = new ToDoListFrame2(todoItems);
        frame.setVisible(true);
    }
}
```

Exercise:

But wait a minute! There is still a problem with our code. If we try adding two or more items with the same name, JAVA will not allow us to select any of these items except the topmost one!



Looking at the **updateList()** method and the **listSelectionEventHandler()** method, can you determine what the problem is ? As a practice exercise, try to solve this problem by making some appropriate changes to the code. You may want to look at the **JList** class in the JAVA documentation.
