



# 10.7.2: Swing

**IT1406 – Introduction to Programming**

**Level I - Semester 1**

## 10.7.2.1 Introducing Swings

- **Swing** is a framework that provides more powerful and flexible GUI components than does the AWT. As a result, it is the GUI that has been widely used by Java programmers for more than a decade.
- Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit.
- The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers.

## 10.7.2.1 Introducing Swings

- This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as *heavyweight*.

### Swing Is Built on the AWT

- Although Swing eliminates the limitations inherent in the AWT, Swing does not replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java.
- Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing.

# 10.7.2.1 Introducing Swings

## Two Key Swing Features

- As just explained, Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel.
- Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing. Each is examined here.

## Swing Components Are Lightweight

- With very few exceptions, Swing components are *lightweight*. This means that they are written entirely in Java and do not map directly to platform-specific peers.

## 10.7.2.1 Introducing Swings

- Thus, lightweight components are more efficient and more flexible.
- Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. As a result, each component will work in a consistent manner across all platforms.

### Swing Supports a Pluggable Look and Feel

- Swing supports a *pluggable look and feel* (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing.
- This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.

## 10.7.2.1 Introducing Swings

- Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects.
- In other words, it is possible to “plug in” a new look and feel for any given component without creating any side effects in the code that uses that component.
- Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply “plugged in.” Once this is done, all components are automatically rendered using that style.

## 10.7.2.1 Introducing Swings

- Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look and feel that acts like a specific platform.
- For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.
- Java 8 provides look-and-feels, such as metal and Nimbus, that are available to all Swing users.
- The metal look and feel is also called the *Java look and feel*. It is platform-independent and available in all Java execution environments.

## 10.7.2.1 Introducing Swings

- It is also the default look and feel. Windows environments also have access to the Windows look and feel. This chapter uses the default Java look and feel (metal) because it is platform independent.

### The MVC Connection

- In general, a visual component is a composite of three distinct aspects:
  - The way that the component looks when rendered on the screen
  - The way that the component reacts to the user
  - The state information associated with the component
- No matter what architecture is used to implement a component, it must implicitly contain these three parts.



## 10.7.2.1 Introducing Swings

- Over the years, one component architecture has proven itself to be exceptionally effective: *Model-View-Controller*, or MVC for short.
- The MVC architecture is successful because each piece of the design corresponds to an aspect of a component.
- In MVC terminology, the *model* corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.
- The *view* determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.

## 10.7.2.1 Introducing Swings

- The *controller* determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated.
- By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.
- Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components.

## 10.7.2.1 Introducing Swings

- Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the *UI delegate*. For this reason, Swing's approach is called either the *Model-Delegate* architecture or the *Separable Model* architecture.
- Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.
- Swing's pluggable look and feel is made possible by its Model-Delegate architecture.
- Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program.

## 10.7.2.1 Introducing Swings

- Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.
- To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate.
- Models are defined by interfaces. For example, the model for a button is defined by the **ButtonModel** interface.
- UI delegates are classes that inherit **ComponentUI**. For example, the UI delegate for a button is ButtonUI. Normally, your programs will not interact directly with the UI delegate.

## 10.7.2.2. Components and containers

- A Swing GUI consists of two key items: *components* and *containers*. However, this distinction is mostly conceptual because all containers are also components.
- The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider.
- A *container* holds a group of components. Thus, a container is a special type of component that is designed to hold other components.
- Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container.

## 10.7.2.2. Components and containers

- Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.
- Let's look a bit more closely at components and containers.

### Components

- In general, Swing components are derived from the **JComponent** class. JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel.
- JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component.

## 10.7.2.2. Components and containers

- All of Swing's components are represented by classes defined within the package **javax.swing**. The following table shows several class names for Swing components (including those used as containers).

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem

## 10.7.2.2. Components and containers

- All component classes begin with the letter **J**. For example, the class for a label is **JLabel**; the class for a push button is  **JButton**; and the class for a scroll bar is **JScrollBar**.

### Containers

- Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**.
- Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.



## 10.7.2.2. Components and containers

- As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container.
- Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is JFrame. The one used for applets is JApplet.
- The second type of containers supported by Swing are lightweight containers. Lightweight containers do inherit JComponent. An example of a lightweight container is **JPanel**, which is a general-purpose container.
- Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container.

## 10.7.2.3 The Swing packages

- Swing is a very large subsystem and makes use of many packages. At the time of this writing, these are the packages defined by Swing.

javax.swing	javax.swing.plaf.basic	javax.swing.text
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.html.parser
javax.swing.event	javax.swing.plaf.nimbus	javax.swing.text.rtf
javax.swing.filechooser	javax.swing.plaf.synth	javax.swing.tree
javax.swing.plaf	javax.swing.table	javax.swing.undo

- The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

## 10.7.2.4 Event handling in Swings

- Events can also be generated in ways not directly related to user input. For example, an event is generated when a timer goes off. Whatever the case, event handling is a large part of any Swing-based application.
- The event handling mechanism used by Swing is the same as that used by the AWT. This approach is called as **delegation event model**.
- In many cases, Swing uses the same events as does the AWT, and these events are packaged in `java.awt.event`. Events specific to Swing are stored in **`javax.swing.event`**.
- Although events are handled in Swing in the same way as they are with the AWT, it is still useful to work through a simple example. The following program handles the event generated by a Swing push button.

# 10.7.2.4 Event handling in Swings

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

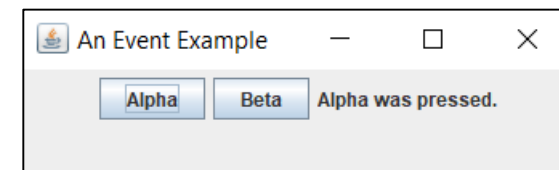
class EventDemo {
    JLabel jlab;
    EventDemo() {
        // Create a new JFrame container.
        JFrame jfrm = new JFrame("An Event Example");
        // Specify FlowLayout for the layout manager.
        jfrm.setLayout(new FlowLayout());
        // Give the frame an initial size.
        jfrm.setSize(220, 90);
        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Make two buttons.
        JButton jbtnAlpha = new JButton("Alpha");
        JButton jbtnBeta = new JButton("Beta");
        // Add action listener for Alpha.
        jbtnAlpha.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                jlab.setText("Alpha was pressed.");
            }
        });
        // Add action listener for Beta.
        jbtnBeta.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                jlab.setText("Beta was pressed.");
            }
        });
    }
}
```

```
// Add the buttons to the content pane.
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
// Create a text-based label.
jlab = new JLabel("Press a button.");
// Add the label to the content pane.
jfrm.add(jlab);

// Display the frame.
jfrm.setVisible(true);
}

public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new EventDemo();
        }
    });
}
```

Sample Output



## 10.7.2.4 Event handling in Swings

- First, the program now imports both the `java.awt` and `java.awt.event` packages. The `java.awt` package is needed because it contains the `FlowLayout` class, which supports the standard flow layout manager used to lay out components in a frame.
- The `java.awt.event` package is needed because it defines the `ActionListener` interface and the `ActionEvent` class.
- The `EventDemo` constructor begins by creating a `JFrame` called `jfrm`. It then sets the layout manager for the content pane of `jfrm` to `FlowLayout`.
- However, for this example, `FlowLayout` is more convenient. The `FlowLayout` is assigned using `jfrm.setLayout(new FlowLayout())`.

## 10.7.2.4 Event handling in Swings

- After setting the size and default close operation, EventDemo() creates two push buttons, as shown here:

```
JButton jbtnAlpha = new JButton("Alpha");
```

```
JButton jbtnBeta = new JButton("Beta");
```

- The first button will contain the text "Alpha" and the second will contain the text "Beta". Swing push buttons are instances of **JButton**. JButton supplies several constructors. The one used is **JButton(String msg)**. The *msg* parameter specifies the string that will be displayed inside the button.
- When a push button is pressed, it generates an ActionEvent. Thus, JButton provides the **addActionListener()** method, which is used to add an action listener.

## 10.7.2.4 Event handling in Swings

- The ActionListener interface defines only one method **void actionPerformed(ActionEvent ae)**. This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred.
- Next, event listeners for the button's action events are added by the code shown here:

```
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
    }
});
// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
    }
});
```

## 10.7.2.4 Event handling in Swings

- Here, anonymous inner classes are used to provide the event handlers for the two buttons. Each time a button is pressed, the string displayed in jlab is changed to reflect which button was pressed.
- Next, the buttons are added to the content pane of jfrm:

```
jfrm.add(jbtnAlpha);
```

```
jfrm.add(jbtnBeta);
```

- Finally, jlab is added to the content pane and window is made visible. When you run the program, each time you press a button, a message is displayed in the label that indicates which button was pressed.



## 10.7.2.5 Swings components at work

- The Swing components such as buttons, check boxes, trees, and tables provide rich functionality and allow a high level of customization. The Swing component classes described in this topic are shown here:

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JScrollPane
JTable	JTextField	JToggleButton	JTree

- These components are all lightweight, which means that they are all derived from JComponent.
- The Swing components are demonstrated in applets because the code for an applet is more compact than it is for a desktop application. However, the same techniques apply to both applets and applications.

# 10.7.2.5 Swings components at work

## JLabel and ImageIcon

- **JLabel** is Swing's easiest-to-use component and it creates a label. JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input.
- JLabel defines several constructors. Here are three of them:
  - JLabel(Icon *icon*)**
  - JLabel(String *str*)**
  - JLabel(String *str*, Icon *icon*, int *align*)**
- Here, *str* and *icon* are the text and icon used for the label. The *align* argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label.

## 10.7.2.5 Swings components at work

- It must be one of the following values: **LEFT**, **RIGHT**, **CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.
- Icons are specified by objects of type **Icon**, which is an interface defined by Swing. The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** implements **Icon** and encapsulates an image.
- Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor.
- There are several ways to provide the image, including reading it from a file or downloading it from a URL.

## 10.7.2.5 Swings components at work

- ImageIcon constructor used by the example in this section is **`ImageIcon(String filename)`**. It obtains the image in the file named *filename*.
- The icon and text associated with the label can be obtained by the following methods:

**`Icon getIcon()`**

**`String getText()`**

- The icon and text associated with a label can be set by these methods:

**`void setIcon(Icon icon)`**

**`void setText(String str)`**

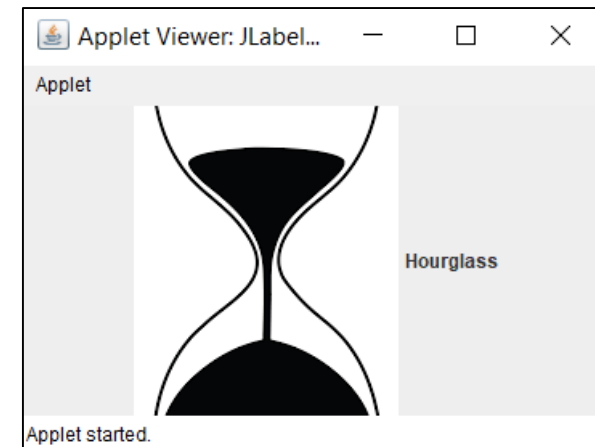
## 10.7.2.5 Swings components at work

- Here, *icon* and *str* are the icon and text, respectively. Therefore, using `setText()` it is possible to change the text inside a label during program execution.
- The following applet illustrates how to create and display a label containing both an icon and a string.
- It begins by creating an `ImageIcon` object for the file `hourglass.png` which depicts an hourglass. This is used as the second argument to the `JLabel` constructor.
- The first and last arguments for the `JLabel` constructor are the label text and the alignment. Finally, the label is added to the content pane.

# 10.7.2.5 Swings components at work

```
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=200>
</applet>
*/
public class JLabelDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeLaterAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
    private void makeGUI() {
        // Create an icon.
        ImageIcon ii = new ImageIcon("Hourglass.png");
        // Create a label.
        JLabel jl = new JLabel("Hourglass", ii, JLabel.CENTER);
        // Add the label to the content pane.
        add(jl);
    }
}
```

Sample Output



# 10.7.2.5 Swings components at work

## TextField

- JTextField is the simplest Swing text component. It is also probably its most widely used text component.
- JTextField allows you to edit one line of text. It is derived from JTextComponent which provides the basic functionality common to Swing text components. JTextField uses the Document interface for its model.
- Three of JTextField's constructors are shown here:

**TextField(int *cols*)**

**TextField(String *str*, int *cols*)**

**TextField(String *str*)**

## 10.7.2.5 Swings components at work

- Here, *str* is the string to be initially presented, and *cols* is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.
- JTextField generates events in response to user interaction. For example, an **ActionEvent** is fired when the user presses enter. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (CaretEvent is packaged in javax.swing.event.) Other events are also possible.
- In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call **getText()**.



## 10.7.2.5 Swings components at work

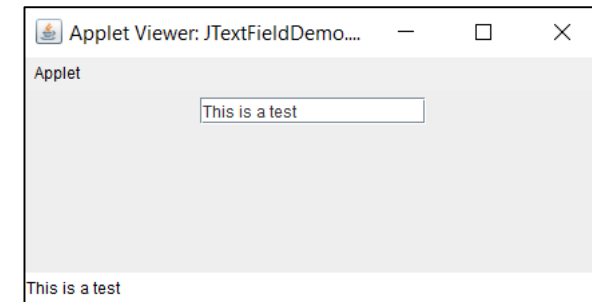
- The following example illustrates JTextField. It creates a JTextField and adds it to the content pane. When the user presses enter, an action event is generated. This is handled by displaying the text in the status window.

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

public class JTextFieldDemo extends JApplet {
    JTextField jtf;
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```

```
private void makeGUI() {
    // Change to flow layout.
    setLayout(new FlowLayout());
    // Add text field to content pane.
    jtf = new JTextField(15);
    add(jtf);
    jtf.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            // Show text when user presses ENTER.
            showStatus(jtf.getText());
        }
    });
}
```

Sample Output



# 10.7.2.5 Swings components at work

## The Swing Buttons

- Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**. All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.
- AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected.
- Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button.

## 10.7.2.5 Swings components at work

- The following methods set these icons:

**void setDisabledIcon(Icon *di*)**

**void setPressedIcon(Icon *pi*)**

**void setSelectedIcon(Icon *si*)**

**void setRolloverIcon(Icon *ri*)**

- Here *di*, *pi*, *si*, and *ri* are the icons to be used for the indicated purpose. The text associated with a button can be read and written via the following methods. Here, *str* is the text to be associated with the button.

**String getText()**

**void setText(String *str*)**

## 10.7.2.5 Swings components at work

- The model used by all buttons is defined by the **ButtonModel** interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

### **JButton**

- The JButton class provides the functionality of a push button. JButton allows an icon, a string, or both to be associated with the push button.
- Three of its constructors are shown here. *str* and *icon* are the string and icon used for the button.

**JButton(Icon *icon*)**

**JButton(String *str*)**

**JButton(String *str*, Icon *icon*)**

## 10.7.2.5 Swings components at work

- When the button is pressed, an `ActionEvent` is generated. Using the `ActionEvent` object passed to the `actionPerformed()` method of the registered `ActionListener`, you can obtain the *action command* string associated with the button.
- By default, this is the string displayed inside the button. However, you can set the action command by calling **`setActionCommand()`** on the button. You can obtain the action command by calling **`String getActionCommand()`** on the event object.
- The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

# 10.7.2.5 Swings components at work

- The following demonstrates an icon-based button.

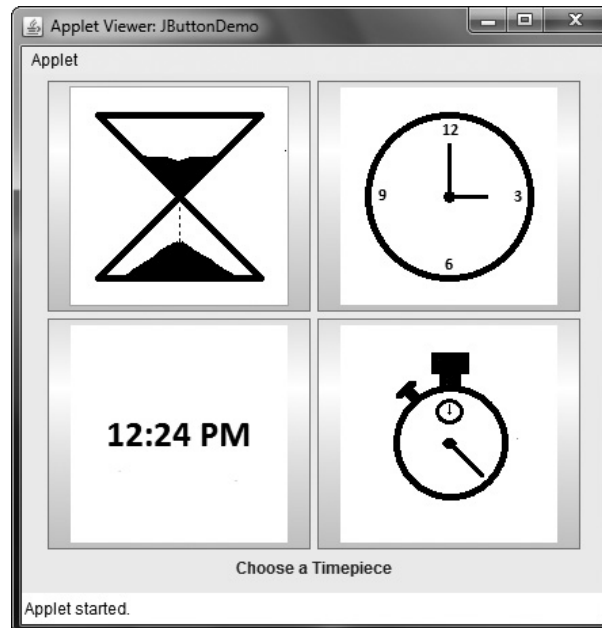
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JButtonDemo extends JApplet implements ActionListener {
    JLabel jlab;
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
    private void makeGUI() {
        // Change to flow layout.
        setLayout(new FlowLayout());
        // Add buttons to content pane.
        ImageIcon hourglass = new ImageIcon("hourglass.png");
        JButton jb = new JButton(hourglass);
        jb.setActionCommand("Hourglass");
        jb.addActionListener(this);
        add(jb);
    }
}
```

```
ImageIcon analog = new ImageIcon("analog.png");
jb = new JButton(analog);
jb.setActionCommand("Analog Clock");
jb.addActionListener(this);
add(jb);
ImageIcon digital = new ImageIcon("digital.png");
jb = new JButton(digital);
jb.setActionCommand("Digital Clock");
jb.addActionListener(this);
add(jb);
ImageIcon stopwatch = new ImageIcon("stopwatch.png");
jb = new JButton(stopwatch);
jb.setActionCommand("Stopwatch");
jb.addActionListener(this);
add(jb);
// Create and add the label to content pane.
jlab = new JLabel("Choose a Timepiece");
add(jlab);
}
// Handle button events.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}
```

## 10.7.2.5 Swings components at work

- It displays four push buttons and a label. Each button displays an icon that represents a timepiece. When a button is pressed, the name of that timepiece is displayed in the label.



Output from the button example is shown here.

## 10.7.2.5 Swings components at work

### JToggleButton

- A useful variation on the push button is called a *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released.
- That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up).
- Therefore, each time a toggle button is pushed, it toggles between its two states.
- Toggle buttons are objects of the **JToggleButton** class. JToggleButton implements AbstractButton.



## 10.7.2.5 Swings components at work

- In addition to creating standard toggle buttons, `JToggleButton` is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**.
- Thus, `JToggleButton` defines the basic functionality of all two-state components. `JToggleButton` defines several constructors. The one used by the example in this topic is **`JToggleButton(String str)`**.
- This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position. Other constructors enable to create toggle buttons that contain images, or images and text.
- `JToggleButton` uses a model defined by a nested class called **`JToggleButton.Toggle-ButtonModel`**.

## 10.7.2.5 Swings components at work

- Like JButton, JToggleButton generates an action event each time it is pressed. Unlike JButton, however, JToggleButton also generates an item event.
- This event is used by those components that support the concept of selection. When a JToggleButton is pressed in, it is selected. When it is popped out, it is deselected.
- To handle item events, you must implement the **ItemListener** interface. Each time an item event is generated, it is passed to the **itemStateChanged()** method defined by ItemListener.
- Inside itemStateChanged( ), the **Object getItem()** method can be called on the ItemEvent object to obtain a reference to the JToggleButton instance that generated the event.

## 10.7.2.5 Swings components at work

- A reference to the button is returned. You will need to cast this reference to `JToggleButton`.
- The easiest way to determine a toggle button's state is by calling the **boolean isSelected()** method (inherited from `AbstractButton`) on the button that generated the event. It returns `true` if the button is selected and `false` otherwise.
- Here is an example that uses a toggle button. It simply calls `isSelected()` to determine the button's state.

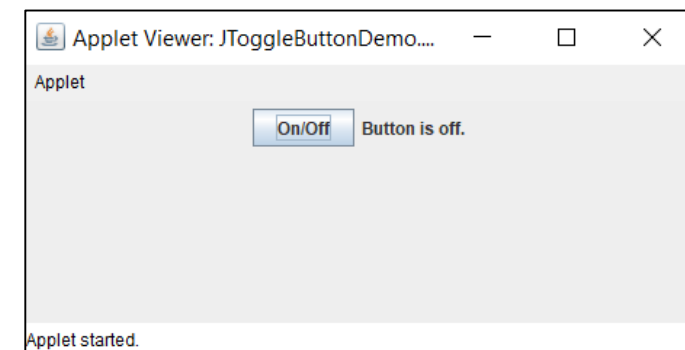
# 10.7.2.5 Swings components at work

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

public class JToggleButtonDemo extends JApplet {
    JLabel jlab;
    JToggleButton jtbn;
    public void init() {
        try {
            SwingUtilities.invokeLaterAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```

```
private void makeGUI() {
    setLayout(new FlowLayout()); // Change to flow layout.
    jlab = new JLabel("Button is off."); // Create a label.
    jtbn = new JToggleButton("On/Off"); // Make a toggle button.
    // Add an item listener for the toggle button.
    jtbn.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent ie) {
            if(jtbn.isSelected())
                jlab.setText("Button is on.");
            else
                jlab.setText("Button is off.");
        }
    });
    // Add the toggle button and label to the content pane.
    add(jtbn);
    add(jlab);
}
```

Sample Output



# 10.7.2.5 Swings components at work

## Check Boxes

- The `JCheckBox` class provides the functionality of a check box. Its immediate superclass is `JToggleButton`.
- `JCheckBox` defines several constructors. The one used here is **`JCheckBox(String str)`**. It creates a check box that has the text specified by *str* as a label. Other constructors let you specify the initial selection state of the button and specify an icon.
- When the user selects or deselects a check box, an `ItemEvent` is generated. You can obtain a reference to the **`JCheckBox`** that generated the event by calling `getItem()` on the `ItemEvent` passed to the `itemStateChanged()` method defined by `ItemListener`.

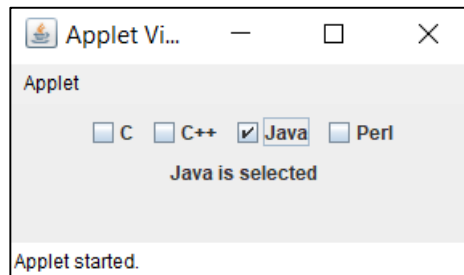
## 10.7.2.5 Swings components at work

- The easiest way to determine the selected state of a check box is to call **isSelected()** on the JCheckBox instance.
- The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an ItemEvent is generated.
- Inside the itemStateChanged() method, getItem() is called to obtain a reference to the JCheckBox object that generated the event.
- Next, a call to isSelected() determines if the box was selected or cleared. The getText() method gets the text for that check box and uses it to set the text inside the label.

# 10.7.2.5 Swings components at work

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

public class JCheckBoxDemo extends JApplet implements ItemListener {
    JLabel jlab;
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```



Sample Output

```
private void makeGUI() {
    // Change to flow layout.
    setLayout(new FlowLayout());
    // Add check boxes to the content pane.
    JCheckBox cb = new JCheckBox("C");
    cb.addItemListener(this);
    add(cb);
    cb = new JCheckBox("C++");
    cb.addItemListener(this);
    add(cb);
    cb = new JCheckBox("Java");
    cb.addItemListener(this);
    add(cb);
    cb = new JCheckBox("Perl");
    cb.addItemListener(this);
    add(cb);
    // Create the label and add it to the content pane.
    jlab = new JLabel("Select languages");
    add(jlab);
}

// Handle item events for the check boxes.
public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();
    if(cb.isSelected())
        jlab.setText(cb.getText() + " is selected");
    else
        jlab.setText(cb.getText() + " is cleared");
}
```

# 10.7.2.5 Swings components at work

## Radio Buttons

- Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **JRadioButton** class, which extends JToggleButton.
- JRadioButton provides several constructors. The one used in the example is **JRadioButton(String *str*)**. Here, *str* is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.
- In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time.



## 10.7.2.5 Swings components at work

- For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected.
- A button group is created by the **ButtonGroup** class. Its default constructor is invoked for this purpose. Elements are then added to the button group via **void add(AbstractButton *ab*)** method. Here, *ab* is a reference to the button to be added to the group.
- A JRadioButton generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the ActionListener interface.

## 10.7.2.5 Swings components at work

- Inside the actionPerformed() method, you can use a number of different ways to determine which button was selected. First, you can check its action command by calling **getActionCommand()**.
- By default, the action command is the same as the button label, but you can set the action command to something else by calling **setActionCommand()** on the radio button.
- Second, you can call **getSource()** on the(ActionEvent) object and check that reference against the buttons. Third, you can check each radio button to find out which one is currently selected by calling isSelected() on each button. Finally, each button could use its own action event handler implemented as an anonymous inner class.

## 10.7.2.5 Swings components at work

- Each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.
- The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior.
- Pressing a radio button generates an action event, which is handled by `actionPerformed()`. Within that handler, the `getActionCommand()` method gets the text that is associated with the radio button and uses it to set the text within a label.

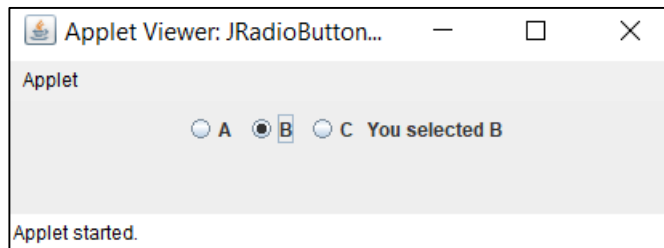
# 10.7.2.5 Swings components at work

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

public class JRadioButtonDemo extends JApplet implements ActionListener {
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeLaterAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```

## Sample Output



```
private void makeGUI() {
    // Change to flow layout.
    setLayout(new FlowLayout());
    // Create radio buttons and add them to content pane.
    JRadioButton b1 = new JRadioButton("A");
    b1.addActionListener(this);
    add(b1);
    JRadioButton b2 = new JRadioButton("B");
    b2.addActionListener(this);
    add(b2);
    JRadioButton b3 = new JRadioButton("C");
    b3.addActionListener(this);
    add(b3);
    // Define a button group.
    ButtonGroup bg = new ButtonGroup();
    bg.add(b1);
    bg.add(b2);
    bg.add(b3);
    // Create a label and add it to the content pane.
    jlab = new JLabel("Select One");
    add(jlab);
}

// Handle button selection.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}
}
```

# 10.7.2.5 Swings components at work

## JTabbedPane

- JTabbedPane encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront.
- Tabbed panes are very common in the modern GUI, and you have no doubt used them many times. Given the complex nature of a tabbed pane, they are surprisingly easy to create and use.
- JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane. The other two constructors let you specify the location of the tabs, which can be along any of the four sides.

## 10.7.2.5 Swings components at work

- JTabbedPane uses the **SingleSelectionModel** model.
- Tabs are added by calling **void addTab(String *name*, Component *comp*)**. Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab.
- Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components.
- The general procedure to use a tabbed pane is outlined here:
  1. Create an instance of JTabbedPane.
  2. Add each tab by calling addTab().
  3. Add the tabbed pane to the content pane.

## 10.7.2.5 Swings components at work

- The following example illustrates a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city.
- The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color.
- The third tab is titled "Flavors" and contains one combo box. This enables the user to select one of three flavors.

# 10.7.2.5 Swings components at work

```
import javax.swing.*;

public class JTabbedPaneDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Cities", new CitiesPanel());
        jtp.addTab("Colors", new ColorsPanel());
        jtp.addTab("Flavors", new FlavorsPanel());
        add(jtp);
    }
}
```

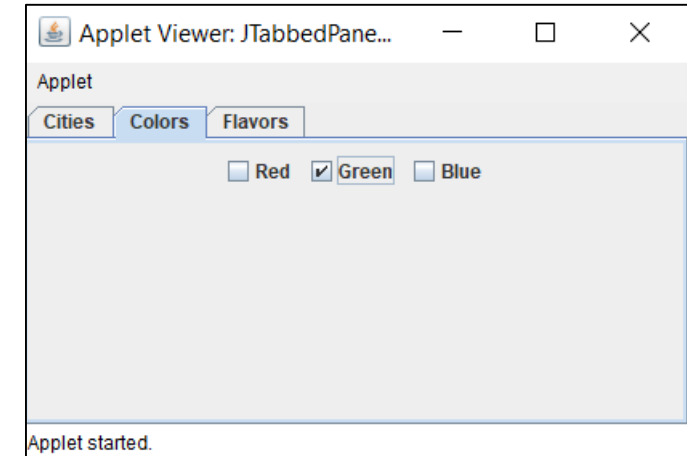
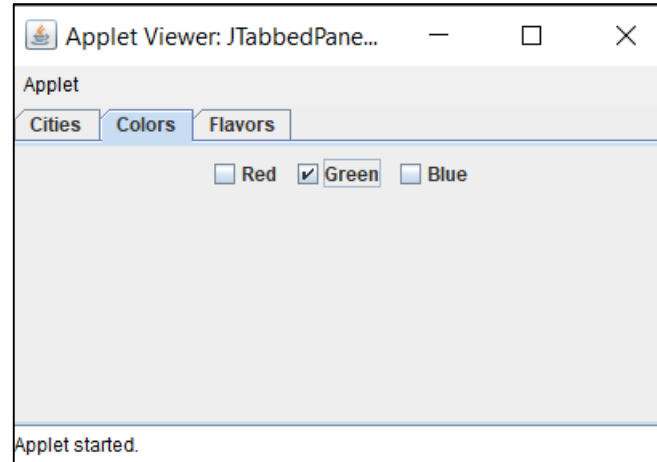
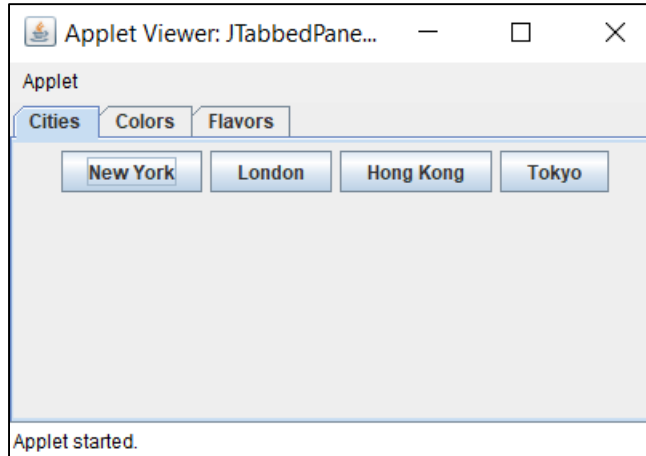
```
// Make the panels that will be added to the tabbed pane.
class CitiesPanel extends JPanel {
    public CitiesPanel() {
        JButton b1 = new JButton("New York");
        add(b1);
        JButton b2 = new JButton("London");
        add(b2);
        JButton b3 = new JButton("Hong Kong");
        add(b3);
        JButton b4 = new JButton("Tokyo");
        add(b4);
    }
}

class ColorsPanel extends JPanel {
    public ColorsPanel() {
        JCheckBox cb1 = new JCheckBox("Red");
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green");
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue");
        add(cb3);
    }
}

class FlavorsPanel extends JPanel {
    public FlavorsPanel() {
        JComboBox<String> jcb = new JComboBox<String>();
        jcb.addItem("Vanilla");
        jcb.addItem("Chocolate");
        jcb.addItem("Strawberry");
        add(jcb);
    }
}
```



## 10.7.2.5 Swings components at work



Output from the tabbed pane example is shown in the above three illustrations.

# 10.7.2.5 Swings components at work

## JScrollPane

- JScrollPane is a lightweight container that automatically handles the scrolling of another component.
- The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a JPanel.
- In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane. Because JScrollPane automates scrolling, it usually eliminates the need to manage individual scroll bars.

## 10.7.2.5 Swings components at work

- The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled.
- The scroll bars scroll the component through the viewport. In its default behavior, a JScrollPane will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added.
- If the component will completely fit within the viewport, the scroll bars are removed. JScrollPane defines several constructors. The one used is **JScrollPane(Component *comp*)**. The component to be scrolled is specified by *comp*.

## 10.7.2.5 Swings components at work

- Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport. Here are the steps to follow to use a scroll pane:
  1. Create the component to be scrolled.
  2. Create an instance of JScrollPane, passing to it the object to scroll.
  3. Add the scroll pane to the content pane.
- The following example illustrates a scroll pane.
- First, a JPanel object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically.

# 10.7.2.5 Swings components at work

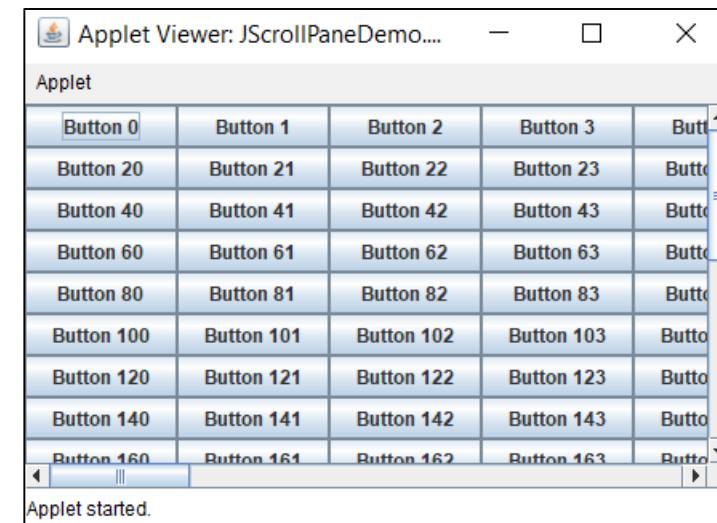
```
import java.awt.*;
import javax.swing.*;

public class JScrollPaneDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {
        // Add 400 buttons to a panel.
        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(20, 20));
        int b = 0;
        for(int i = 0; i < 20; i++) {
            for(int j = 0; j < 20; j++) {
                jp.add(new JButton("Button " + b));
                ++b;
            }
        }

        // Create the scroll pane.
        JScrollPane jsp = new JScrollPane(jp);
        // Add the scroll pane to the content pane.
        // Because the default border layout is used,
        // the scroll pane will be added to the center.
        add(jsp, BorderLayout.CENTER);
    }
}
```

## Sample Output



## 10.7.2.5 Swings components at work

### JList

- In Swing, the basic list class is called JList. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed.
- JList provides several constructors. The one used is **JList(E[ ] *items*)**. This creates a JList that contains the items in the array specified by *items*.
- JList is based on two models. The first is **ListModel**. This interface defines how access to the list data is achieved. The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.

## 10.7.2.5 Swings components at work

- Although a JList will work properly by itself, most of the time you will wrap a JList inside a **JScrollPane**. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the JList component.
- A JList generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item.
- It is handled by implementing **ListSelectionListener**. This listener specifies only one method called void **valueChanged(ListSelectionEvent *e*)**. Here, *e* is a reference to the event.

## 10.7.2.5 Swings components at work

- Although `ListSelectionEvent` does provide some methods of its own, normally you will interrogate the `JList` object itself to determine what has occurred.
- Both `ListSelectionEvent` and `ListSelectionListener` are packaged in **`javax.swing.event`**. By default, a `JList` allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **`void setSelectionMode(int mode)`**. Here, *mode* specifies the selection mode. It must be one of these values defined by `ListSelectionModel`:
  - **`SINGLE_SELECTION`**
  - **`SINGLE_INTERVAL_SELECTION`**
  - **`MULTIPLE_INTERVAL_SELECTION`**



## 10.7.2.5 Swings components at work

- The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. It's just that they also allow a range to be selected.
- You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling **int getSelectedIndex()**. Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, -1 is returned.
- Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling **getSelectedValue()**. It returns a reference to the first selected value. If no value has been selected, it returns null.

## 10.7.2.5 Swings components at work

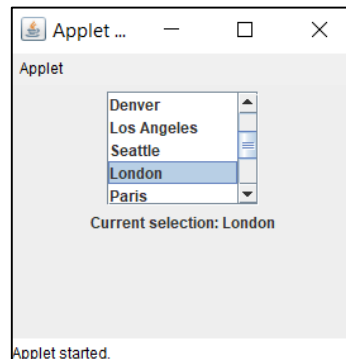
- The following applet demonstrates a simple JList, which holds a list of cities.
- Each time a city is selected in the list, a ListSelectionEvent is generated, which is handled by the **valueChanged()** method defined by ListSelectionListener.
- It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

# 10.7.2.5 Swings components at work

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

public class JListDemo extends JApplet {
    JList<String> jlst;
    JLabel jlab;
    JScrollPane jscrlp;
    // Create an array of cities.
    String Cities[] = { "New York", "Chicago", "Houston", "Denver",
        "Los Angeles", "Seattle", "London", "Paris", "New Delhi",
        "Hong Kong", "Tokyo", "Sydney" };
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```

Sample Output



```
private void makeGUI() {
    // Change to flow layout.
    setLayout(new FlowLayout());
    // Create a JList.
    jlst = new JList<String>(Cities);
    // Set the list selection mode to single selection.
    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    // Add the list to a scroll pane.
    jscrlp = new JScrollPane(jlst);
    // Set the preferred size of the scroll pane.
    jscrlp.setPreferredSize(new Dimension(120, 90));
    // Make a label that displays the selection.
    jlab = new JLabel("Choose a City");
    // Add selection listener for the list.
    jlst.addListSelectionListener(new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent le) {
            // Get the index of the changed item.
            int idx = jlst.getSelectedIndex();
            // Display selection, if item was selected.
            if (idx != -1)
                jlab.setText("Current selection: " + Cities[idx]);
            else // Otherwise, reprompt.
                jlab.setText("Choose a City");
        }
    });
    // Add the list and label to the content pane.
    add(jscrlp);
    add(jlab);
}
```

## 10.7.2.5 Swings components at work

### JComboBox

- Swing provides a combo box (a combination of a text field and a drop-down list) through the **JComboBox** class.
- A combo box normally displays one entry, but it will also display a dropdown list that allows a user to select a different entry. You can also create a combo box that lets the user enter a selection into the text field.
- The JComboBox constructor used by the example is **JComboBox(E[ ] items)**. Here, *items* is an array that initializes the combo box. Other constructors are available.
- JComboBox uses the **ComboBoxModel**. Mutable combo boxes (those whose entries can be changed) use the **MutableComboBoxModel**.

## 10.7.2.5 Swings components at work

- In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **void addItem(E *obj*)** method. Here, *obj* is the object to be added to the combo box. This method must be used only with mutable combo boxes.
- JComboBox generates an action event when the user selects an item from the list. JComboBox also generates an item event when the state of selection changes, which occurs when an item is selected or deselected.
- Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item. Often, it is sufficient to simply listen for action events, but both event types are available for your use.

## 10.7.2.5 Swings components at work

- One way to obtain the item selected in the list is to call **Object.getSelectedItemAt()** on the combo box. It is shown here:
- You will need to cast the returned value into the type of object stored in the list.
- The following example demonstrates the combo box. The combo box contains entries for "Hourglass", "Analog", "Digital", and "Stopwatch". When a timepiece is selected, an icon-based label is updated to display it. You can see how little code is required to use this powerful component.

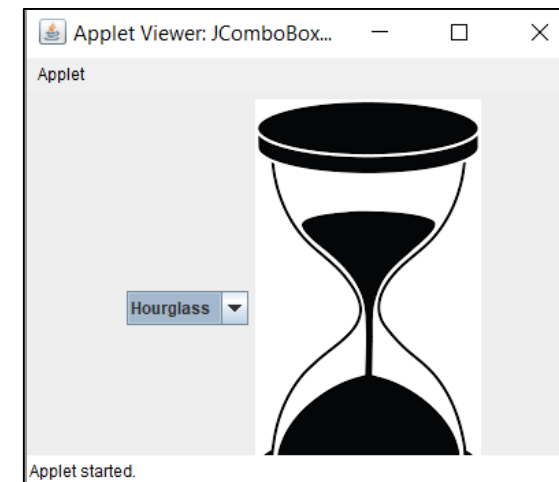
# 10.7.2.5 Swings components at work

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

public class JComboBoxDemo extends JApplet {
    JLabel jlab;
    ImageIcon hourglass, analog, digital, stopwatch;
    JComboBox<String> jcb;
    String timepieces[] = { "Hourglass", "Analog", "Digital", "Stopwatch" };
    public void init() {
        try {
            SwingUtilities.invokeLaterAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```

```
private void makeGUI() {
    // Change to flow layout.
    setLayout(new FlowLayout());
    // Instantiate a combo box and add it to the content pane.
    jcb = new JComboBox<String>(timepieces);
    add(jcb);
    // Handle selections.
    jcb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            String s = (String) jcb.getSelectedItem();
            jlab.setIcon(new ImageIcon(s + ".png"));
        }
    });
    // Create a label and add it to the content pane.
    jlab = new JLabel(new ImageIcon("hourglass.png"));
    add(jlab);
}
```

Sample Output



# 10.7.2.5 Swings components at work

## Trees

- A tree is a component that presents a hierarchical view of data. The user has the ability to expand or collapse individual subtrees in this display.
- Trees are implemented in Swing by the **JTree** class. A sampling of its constructors is shown here:

**JTree(Object obj [ ])**

**JTree(Vector<?> v)**

**JTree(TreeNode tn)**

- In the first form, the tree is constructed from the elements in the array *obj*. The second form constructs the tree from the elements of vector *v*. In the third form, the tree whose root node is specified by *tn* specifies the tree.



## 10.7.2.5 Swings components at work

- Although JTree is packaged in javax.swing, its support classes and interfaces are packaged in **javax.swing.tree**. This is because the number of classes and interfaces needed to support JTree is quite large.
- JTree relies on two models: **TreeModel** and **TreeSelectionModel**. A JTree generates a variety of events, but three relate specifically to trees: **TreeExpansionEvent**, **TreeSelectionEvent**, and **TreeModelEvent**.
- TreeExpansionEvent events occur when a node is expanded or collapsed. A TreeSelectionEvent is generated when the user selects or deselects a node within the tree. A TreeModelEvent is fired when the data or structure of the tree changes.
- The listeners for these events are **TreeExpansionListener**, **TreeSelectionListener**, and **TreeModelListener**, respectively.

## 10.7.2.5 Swings components at work

- The tree event classes and listener interfaces are packaged in **javax.swing.event**.
- The event handled by the sample program shown in this section is `TreeSelectionEvent`. To listen for this event, implement `TreeSelectionListener`. It defines only one method, called **valueChanged()**, which receives the `TreeSelectionEvent` object.
- You can obtain the path to the selected object by calling **TreePath getPath()** on the event object. It returns a `TreePath` object that describes the path to the changed node.
- The `TreePath` class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods.

## 10.7.2.5 Swings components at work

- The **TreeNode** interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes.
- The **MutableTreeNode** interface extends **TreeNode**. It declares methods that can insert and remove child nodes or change the parent node.
- The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of its constructors is **DefaultMutableTreeNode(Object *obj*)**. Here, *obj* is the object to be enclosed in this tree node.
- The new tree node doesn't have a parent or children. To create a hierarchy of tree nodes, the **void add(MutableTreeNode *child*)** method of **DefaultMutableTreeNode** can be used.

## 10.7.2.5 Swings components at work

- Here, *child* is a mutable tree node that is to be added as a child to the current node.
- JTree does not provide any scrolling capabilities of its own. Instead, a JTree is typically placed within a **JScrollPane**. This way, a large tree can be scrolled through a smaller viewport.
- Here are the steps to follow to use a tree:
  1. Create an instance of JTree.
  2. Create a JScrollPane and specify the tree as the object to be scrolled.
  3. Add the tree to the scroll pane.
  4. Add the scroll pane to the content pane.

## 10.7.2.5 Swings components at work

- The following example illustrates how to create a tree and handle selections. The program creates a `DefaultMutableTreeNode` instance labeled "Options". This is the top node of the tree hierarchy.
- Additional tree nodes are then created, and the `add()` method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the `JTree` constructor.
- The tree is then provided as the argument to the `JScrollPane` constructor. This scroll pane is then added to the content pane.
- Next, a label is created and added to the content pane. The tree selection is displayed in this label. To receive selection events from the tree, a `TreeSelectionListener` is registered for the tree. Inside the `valueChanged()` method, the path to the current selection is obtained and displayed.

# 10.7.2.5 Swings components at work

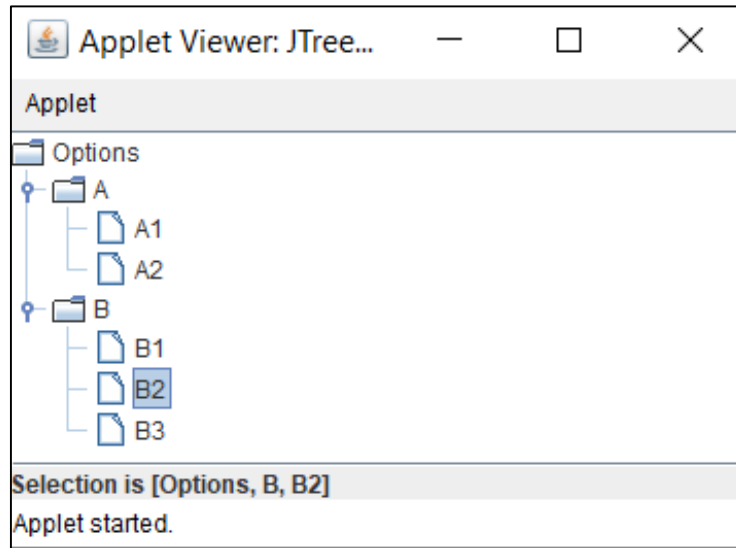
```
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class JTreeDemo extends JApplet {
    JTree tree;
    JLabel jlab;
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
    private void makeGUI() {
        // Create top node of tree.
        DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");
        // Create subtree of "A".
        DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
        top.add(a);
        DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
        a.add(a1);
        DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
        a.add(a2);

        // Create subtree of "B"
        DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
        top.add(b);
        DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
        b.add(b1);
        DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
        b.add(b2);
        DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
        b.add(b3);
        // Create the tree.
        tree = new JTree(top);
        // Add the tree to a scroll pane.
        JScrollPane jsp = new JScrollPane(tree);
        // Add the scroll pane to the content pane.
        add(jsp);
        // Add the label to the content pane.
        jlab = new JLabel();
        add(jlab, BorderLayout.SOUTH);
        // Handle tree selection events.
        tree.addTreeSelectionListener(new TreeSelectionListener() {
            public void valueChanged(TreeSelectionEvent tse) {
                jlab.setText("Selection is " + tse.getPath());
            }
        });
    }
}
```

## 10.7.2.5 Swings components at work

- Output from the tree example is shown here:



The string presented in the text field describes the path from the top tree node to the selected node.

## 10.7.2.5 Swings components at work

### JTable

- JTable is a component that displays rows and columns of data. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table.
- Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell.
- JTable is a sophisticated component that offers many more options and features. Like JTree, JTable has many classes and interfaces associated with it. These are packaged in **javax.swing.table**.



## 10.7.2.5 Swings components at work

- JTable does not provide any scrolling capabilities of its own. Instead, you will normally wrap a JTable inside a JScrollPane.
- JTable supplies several constructors. The one used here is **JTable(Object *data*[], Object *colHeads*[])**. Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.
- JTable relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format.
- The second is the table column model, which is represented by **TableColumnModel**. JTable is defined in terms of columns, and it is TableColumnModel that specifies the characteristics of a column.

## 10.7.2.5 Swings components at work

- These two models are packaged in **javax.swing.table**. The third model determines how items are selected, and it is specified by the **ListSelectionModel**.
- A JTable can generate several different events. The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**.
- A ListSelectionEvent is generated when the user selects something in the table. By default, JTable allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected.
- TableModelEvent is fired when that table's data changes in some way.

## 10.7.2.5 Swings components at work

- Here are the steps required to set up a simple JTable that can be used to display data:
  1. Create an instance of JTable.
  2. Create a JScrollPane object, specifying the table as the object to scroll.
  3. Add the table to the scroll pane.
  4. Add the scroll pane to the content pane.
- The following example illustrates how to create and use a simple table. A one-dimensional array of strings called **colHeads** is created for the column headings.
- A two-dimensional array of strings called **data** is created for the table cells. Each element in the array is an array of three strings.

## 10.7.2.5 Swings components at work

- These arrays are passed to the JTable constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane.
- The table displays the data in the data array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is data in this case.

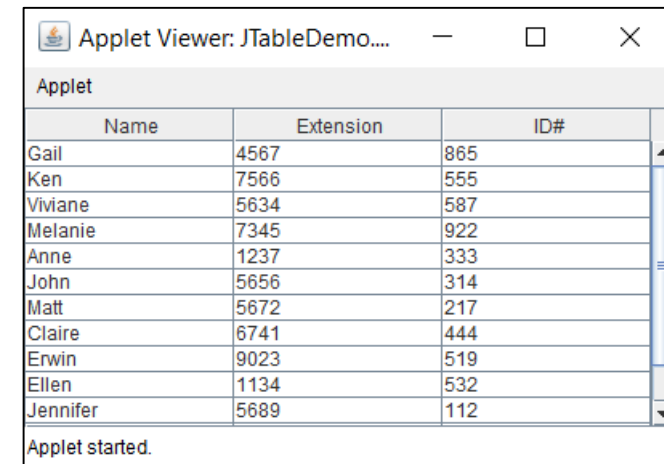
```
import javax.swing.*;

public class JTableDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeLaterAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```

# 10.7.2.5 Swings components at work

```
private void makeGUI() {  
    // Initialize column headings.  
    String[] colHeads = { "Name", "Extension", "ID#" };  
    // Initialize data.  
    Object[][] data = {  
        { "Gail", "4567", "865" }, { "Ken", "7566", "555" },  
        { "Viviane", "5634", "587" }, { "Melanie", "7345", "922" },  
        { "Anne", "1237", "333" }, { "John", "5656", "314" },  
        { "Matt", "5672", "217" }, { "Claire", "6741", "444" },  
        { "Erwin", "9023", "519" }, { "Ellen", "1134", "532" },  
        { "Jennifer", "5689", "112" }, { "Ed", "9030", "133" },  
        { "Helen", "6751", "145" }  
    };  
    // Create the table.  
    JTable table = new JTable(data, colHeads);  
    // Add the table to a scroll pane.  
    JScrollPane jsp = new JScrollPane(table);  
    // Add the scroll pane to the content pane.  
    add(jsp);  
}
```

## Sample Output



Name	Extension	ID#
Gail	4567	865
Ken	7566	555
Viviane	5634	587
Melanie	7345	922
Anne	1237	333
John	5656	314
Matt	5672	217
Claire	6741	444
Erwin	9023	519
Ellen	1134	532
Jennifer	5689	112

Applet started.

## 10.7.2.6 Creating Swing Menus

- The Swing menu system is supported by a group of related classes. The ones used in this topic are shown in the following table and they represent the core of the menu system. Although they may seem a bit confusing at first, Swing menus are quite easy to use.
- Swing allows a high degree of customization, if desired; however, you will normally use the menu classes as-is because they support all of the most needed options. For example, you can easily add images and keyboard shortcuts to a menu.

## 10.7.2.6 Creating Swing Menus

Class	Description
JMenuBar	An object that holds the top-level menu for the application.
JMenu	A standard menu. A menu consists of one or more JMenuItem.
JMenuItem	An object that populates menus.
JCheckBoxMenuItem	A check box menu item.
JRadioButtonMenuItem	A radio button menu item
JSeparator	The visual separator between menu items.
JPopupMenu	A menu that is typically activated by right-clicking the mouse.

- To create the top-level menu for an application, you first create a **JMenuBar** object. This class is, loosely speaking, a container for menus. To the JMenuBar instance, you will add instances of **JMenu**.

## 10.7.2.6 Creating Swing Menus

- Each JMenu object defines a menu. That is, each JMenu object contains one or more selectable items. The items displayed by a JMenu are objects of **JMenuItem**. Thus, a JMenuItem defines a selection that can be chosen by the user.
- As an alternative or adjunct to menus that descend from the menu bar, you can also create stand-alone, popup menus. To create a popup menu, first create an object of type **JPopupMenu**. Then, add JMenuItem's to it. A popup menu is normally activated by clicking the right mouse button when the mouse is over a component for which a popup menu has been defined.



## 10.7.2.6 Creating Swing Menus

- In addition to “standard” menu items, you can also include check boxes and radio buttons in a menu. A check box menu item is created by **JCheckBoxMenuItem**. A radio button menu item is created by **JRadioButtonMenuItem**. Both of these classes extend JMenuItem. They can be used in standard menus and popup menus.
- **JToolBar** creates a stand-alone component that is related to the menu. It is often used to provide fast access to functionality contained within the menus of the application. For example, a toolbar might provide fast access to the formatting commands supported by a word processor.
- **JSeparator** is a convenience class that creates a separator line in a menu. Each menu item extends **AbstractButton**. Thus, all menu items are, essentially, buttons.

## 10.7.2.6 Creating Swing Menus

- Obviously, they won't actually look like buttons when used in a menu, but they will, in many ways, act like buttons. For example, selecting a menu item generates an action event in the same way that pressing a button does.
- JMenuItem is a superclass of JMenu. This allows the creation of submenus, which are, essentially, menus within menus. To create a submenu, create and populate a JMenu object and then add it to another JMenu object.
- Menus can also generate other types of events. For example, each time that a menu is activated, selected, or canceled, a **MenuEvent** is generated that can be listened for via a **MenuListener**. Other menu-related events include **MenuKeyEvent**, **MenuDragMouseEvent**, and **PopupMenuEvent**.

## 10.7.2.6 Creating Swing Menus

- Before create a menu, you need to know something about the three core menu classes: **JMenuBar**, **JMenu**, and **JMenuItem**. These form the minimum set of classes needed to construct a main menu for an application. JMenu and JMenuItem are also used by popup menus. Thus, these classes form the foundation of the menu system.

### JMenuBar

- JMenuBar is essentially a container for menus. Like all components, it inherits **JComponent** (which inherits Container and Component).
- It has only one constructor, which is the default constructor. Therefore, initially the menu bar will be empty, and you will need to populate it with menus prior to use. Each application has one and only one menu bar.

## 10.7.2.6 Creating Swing Menus

- JMenuBar defines several methods, but often you will only need to use one, **JMenu add(JMenu *menu*)** method. It adds a JMenu to the menu bar. Here, *menu* is a JMenu instance that is added to the menu bar. A reference to the menu is returned.
- Menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use **Component add(Component *menu*, int *idx*)**, which is inherited from Container. Here, *menu* is added at the index specified by *idx*. Indexing begins at 0, with 0 being the left-most menu.
- In some cases, you might want to remove a menu that is no longer needed. You can do this by calling `remove()`, which is inherited from Container.

## 10.7.2.6 Creating Swing Menus

- It has these two forms:

**`void remove(Component menu)`**

**`void remove(int idx)`**

- Here, *menu* is a reference to the menu to remove, and *idx* is the index of the menu to remove. Indexing begins at zero.

### JMenu

- JMenu encapsulates a menu, which is populated with JMenuItem. It is derived from JMenuItem. This means that one JMenu can be a selection in another JMenu. This enables one menu to be a submenu of another.
- JMenu defines a number of constructors. The one used in this section is **`JMenu(String name)`**.

## 10.7.2.6 Creating Swing Menus

- This constructor creates a menu that has the title specified by *name*. To create an unnamed menu, you can use the default constructor as **JMenu()**.
- Other constructors are also supported. In each case, the menu is empty until menu items are added to it.
- JMenu defines many methods. To add an item to the menu, use the add() method, which has a number of forms, including the two shown here:

**JMenuItem add(JMenuItem *item*)**

**JMenuItem add(Component *item*, int *idx*)**

- Here, *item* is the menu item to add. The first form adds the item to the end of the menu. The second form adds the item at the index specified by *idx*. Indexing starts at zero. Both forms return a reference to the item added.

## 10.7.2.6 Creating Swing Menus

- An item can remove from a menu by calling `remove( )`. Two of its forms are **`void remove(JMenuItem menu)`** and **`void remove(int idx)`**. Here *menu* is a reference to the item to remove and *idx* is the index of the item to remove.

### JMenuItem

- JMenuItem encapsulates an element in a menu. This element can be a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed.
- JMenuItem is derived from AbstractButton, and every item in a menu can be thought of as a special kind of button. Therefore, when a menu item is selected, an action event is generated.

## 10.7.2.6 Creating Swing Menus

- JMenuItem defines many constructors. The ones used in this section are shown here:

**JMenuItem(String *name*)**

**JMenuItem(Icon *image*)**

**JMenuItem(String *name*, Icon *image*)**

**JMenuItem(String *name*, int *mnem*)**

**JMenuItem(Action *action*)**

- The first constructor creates a menu item with the name specified by *name*. The second creates a menu item that displays the image specified by *image*. The third creates a menu item with the name specified by *name* and the image specified by *image*.



## 10.7.2.6 Creating Swing Menus

- The fourth creates a menu item with the name specified by *name* and uses the keyboard mnemonic specified by *mnem*. This mnemonic enables you to select an item from the menu by pressing the specified key. The last constructor creates a menu item using the information specified in *action*. A default constructor is also supported.
- Because menu items inherit `AbstractButton`, you have access to the functionality provided by `AbstractButton`. One such method that is often useful with menus is **`void setEnabled(boolean enable)`**, which you can use to enable or disable a menu item.
- If *enable* is true, the menu item is enabled. If *enable* is false, the item is disabled and cannot be selected.

## 10.7.2.6 Creating Swing Menus

### Create a Main Menu

- Traditionally, the most commonly used menu is the *main menu*. This is the menu defined by the menu bar, and it is the menu that defines all (or nearly all) of the functionality of an application.
- Fortunately, Swing makes creating and managing the main menu easy. Constructing the main menu requires several steps.
- First, create the JMenuBar object that will hold the menus. Next, construct each menu that will be in the menu bar. In general, a menu is constructed by first creating a JMenu object and then adding JMenuItem's to it.
- After the menus have been created, add them to the menu bar. The menu bar, itself, must then be added to the frame by calling setJMenuBar().

## 10.7.2.6 Creating Swing Menus

- Finally, for each menu item, add an action listener that handles the action event fired when the menu item is selected.
- Here is a program that creates a simple menu bar that contains three menus.
- The first is a standard File menu that contains Open, Close, Save, and Exit selections. The second menu is called Options, and it contains two submenus called Colors and Priority. The third menu is called Help, and it has one item, About.
- When a menu item is selected, the name of the selection is displayed in a label in the content pane.

# 10.7.2.6 Creating Swing Menus

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

class MenuDemo implements ActionListener {
    JLabel jlab;
    MenuDemo() {
        // Create a new JFrame container.
        JFrame jfrm = new JFrame("Menu Demo");
        // Specify FlowLayout for the layout manager.
        jfrm.setLayout(new FlowLayout());
        // Give the frame an initial size.
        jfrm.setSize(220, 200);
        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Create a label that will display the menu selection.
        jlab = new JLabel();
        // Create the menu bar.
        JMenuBar jmb = new JMenuBar();
        // Create the File menu.
        JMenu jmFile = new JMenu("File");
        JMenuItem jmiOpen = new JMenuItem("Open");
        JMenuItem jmiClose = new JMenuItem("Close");
        JMenuItem jmiSave = new JMenuItem("Save");
        JMenuItem jmiExit = new JMenuItem("Exit");
        jmFile.add(jmiOpen);
        jmFile.add(jmiClose);
        jmFile.add(jmiSave);
        jmFile.addSeparator();
        jmFile.add(jmiExit);
        jmb.add(jmFile);

        // Create the Options menu.
        JMenu jmOptions = new JMenu("Options");
        // Create the Colors submenu.
        JMenu jmColors = new JMenu("Colors");
        JMenuItem jmiRed = new JMenuItem("Red");
        JMenuItem jmiGreen = new JMenuItem("Green");
        JMenuItem jmiBlue = new JMenuItem("Blue");
        jmColors.add(jmiRed);
        jmColors.add(jmiGreen);
        jmColors.add(jmiBlue);
        jmOptions.add(jmColors);
        // Create the Priority submenu.
        JMenu jmPriority = new JMenu("Priority");
        JMenuItem jmiHigh = new JMenuItem("High");
        JMenuItem jmiLow = new JMenuItem("Low");
        jmPriority.add(jmiHigh);
        jmPriority.add(jmiLow);
        jmOptions.add(jmPriority);
        // Create the Reset menu item.
        JMenuItem jmiReset = new JMenuItem("Reset");
        jmOptions.addSeparator();
        jmOptions.add(jmiReset);
        // Finally, add the entire options menu to
        // the menu bar
        jmb.add(jmOptions);
        // Create the Help menu.
        JMenu jmHelp = new JMenu("Help");
        JMenuItem jmiAbout = new JMenuItem("About");
        jmHelp.add(jmiAbout);
        jmb.add(jmHelp);
    }
}
```

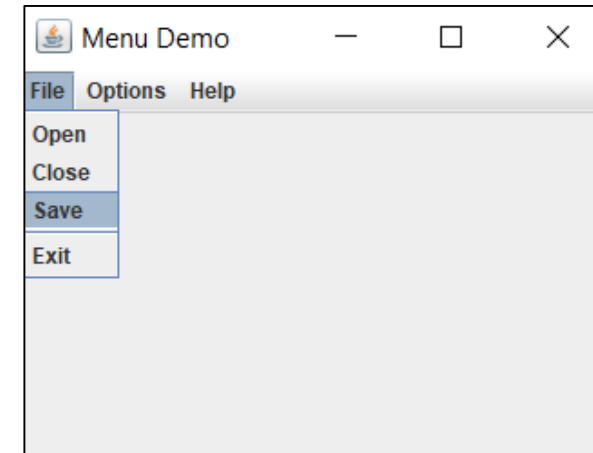
# 10.7.2.6 Creating Swing Menus

```
// Add action listeners for the menu items.
jmiOpen.addActionListener(this); jmiClose.addActionListener(this);
jmiSave.addActionListener(this); jmiExit.addActionListener(this);
jmiRed.addActionListener(this); jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this); jmiHigh.addActionListener(this);
jmiLow.addActionListener(this); jmiReset.addActionListener(this);
jmiAbout.addActionListener(this);
// Add the label to the content pane.
jfrm.add(jlab);
// Add the menu bar to the frame.
jfrm.setJMenuBar(jmb);
// Display the frame.
jfrm.setVisible(true);
}

// Handle menu item action events.
public void actionPerformed(ActionEvent ae) {
    // Get the action command from the menu selection.
    String comStr = ae.getActionCommand();
    // If user chooses Exit, then exit the program.
    if(comStr.equals("Exit")) System.exit(0);
    // Otherwise, display the selection.
    jlab.setText(comStr + " Selected");
}

public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MenuDemo();
        }
    });
}
```

Sample Output



## 10.7.2.6 Creating Swing Menus

### Add Mnemonics and Accelerators to Menu Items

- A menu usually includes support for keyboard shortcuts because they give an experienced user the ability to select menu items rapidly. Keyboard shortcuts come in two forms: **mnemonics** and **accelerators**.
- As it applies to menus, a **mnemonic** defines a key that lets you select an item from an active menu by typing the key. Thus, a mnemonic allows to use the keyboard to select an item from a menu that is already being displayed. An **accelerator** is a key that lets you select a menu item without having to first activate the menu.
- A mnemonic can be specified for both JMenuItem and JMenu objects. There are two ways to set the mnemonic for JMenuItem.

## 10.7.2.6 Creating Swing Menus

- First, it can be specified when an object is constructed using the constructor as **JMenuItem(String *name*, int *mnem*)**. The name is passed in *name* and the mnemonic is passed in *mnem*.
- Second, you can set the mnemonic by calling **setMnemonic()**. To specify a mnemonic for JMenu, you must call **void setMnemonic(int *mnem*)**.
- This method is inherited by both classes from AbstractButton. Here, *mnem* specifies the mnemonic. It should be one of the constants defined in **java.awt.event.KeyEvent**, such as **KeyEvent.VK\_F** or **KeyEvent.VK\_Z**.
- Mnemonics are not case sensitive, so in the case of VK\_A, typing either a or A will work.

## 10.7.2.6 Creating Swing Menus

- By default, the first matching letter in the menu item will be underscored. In cases in which you want to underscore a letter other than the first match, specify the index of the letter as an argument to **void setDisplayedMnemonicIndex(int *idx*)**, which is inherited by both JMenu and JMenuItem from AbstractButton. The index of the letter to underscore is specified by *idx*.
- An accelerator can be associated with a JMenuItem object. It is specified by calling **void setAccelerator(KeyStroke *ks*)**. Here, *ks* is the key combination that is pressed to select the menu item.
- **KeyStroke** is a class that contains several factory methods that construct various types of keystroke accelerators.



## 10.7.2.6 Creating Swing Menus

- The following are three examples:

**static KeyStroke getKeyStroke(char *ch*)**

**static KeyStroke getKeyStroke(Character *ch*, int *modifier*)**

**static KeyStroke getKeyStroke(int *ch*, int *modifier*)**

- Here, *ch* specifies the accelerator character. In the first version, the character is specified as a char value. In the second, it is specified as an object of type Character. In the third, it is a value of type KeyEvent.
- The value of *modifier* must be one or more of the following constants, defined in the **java.awt.event.InputEvent** class.

## 10.7.2.6 Creating Swing Menus

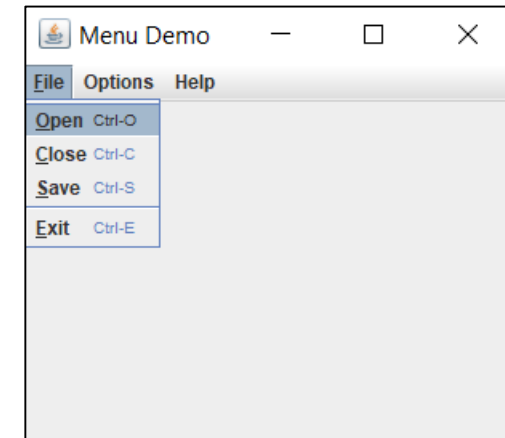
InputEvent.ALT_DOWN_MASK	InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.CTRL_DOWN_MASK	InputEvent.META_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK	

- Therefore, if you pass `VK_A` for the key character and `InputEvent.CTRL_DOWN_MASK` for the modifier, the accelerator key combination is Ctrl-A.
- The following example adds both mnemonics and accelerators to the File menu created by the **MenuDemo** program in the previous section. After making this change, you can select the File menu by typing Alt-F. Then, you can use the mnemonics o, c, s, or e to select an option.

## 10.7.2.6 Creating Swing Menus

```
// Create the File menu with mnemonics and accelerators.
JMenu jmFile = new JMenu("File");
jmFile.setMnemonic(KeyEvent.VK_F);
JMenuItem jmiOpen = new JMenuItem("Open", KeyEvent.VK_O);
jmiOpen.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
    InputEvent.CTRL_DOWN_MASK));
JMenuItem jmiClose = new JMenuItem("Close", KeyEvent.VK_C);
jmiClose.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C,
    InputEvent.CTRL_DOWN_MASK));
JMenuItem jmiSave = new JMenuItem("Save", KeyEvent.VK_S);
jmiSave.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
    InputEvent.CTRL_DOWN_MASK));
JMenuItem jmiExit = new JMenuItem("Exit", KeyEvent.VK_E);
jmiExit.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_E,
    InputEvent.CTRL_DOWN_MASK));
```

The File menu after adding mnemonics and accelerators



## 10.7.2.6 Creating Swing Menus

### Add Images and Tooltips to Menu Items

- You can add images to menu items or use images instead of text. The easiest way to add an image is to specify it when the menu item is being constructed using one of these constructors:

**JMenuItem(*Icon image*)**

**JMenuItem(*String name, Icon image*)**

- The first creates a menu item that displays the image specified by *image*. The second creates a menu item with the name specified by *name* and the image specified by *image*.

## 10.7.2.6 Creating Swing Menus

- For example, here the About menu item is associated with an image when it is created.

```
ImageIcon icon = new ImageIcon("AboutIcon.gif");
```

```
JMenuItem jmiAbout = new JMenuItem("About", icon);
```

- After this addition, the icon specified by icon will be displayed next to the text "About" when the Help menu is displayed.
- You can also add an icon to a menu item after the item has been created by calling **setIcon()**, which is inherited from AbstractButton. You can specify the horizontal alignment of the image relative to the text by calling **setHorizontalTextPosition()**.

## 10.7.2.6 Creating Swing Menus

- You can specify a disabled icon, which is shown when the menu item is disabled, by calling **setDisabledIcon()**. Normally, when a menu item is disabled, the default icon is shown in gray. If a disabled icon is specified, then that icon is displayed when the menu item is disabled.
- A *tooltip* is a small message that describes an item. It is automatically displayed if the mouse remains over the item for a moment. You can add a tooltip to a menu item by calling **void setToolTipText(String msg)** on the item, specifying the text you want displayed. *msg* is the string that will be displayed when the tooltip is activated. For example, this creates a tooltip for the About item:

```
jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

## 10.7.2.6 Creating Swing Menus

### Use `JRadioButtonMenuItem` and `JCheckBoxMenuItem`

- Check boxes and radio buttons can streamline a GUI by allowing a menu to provide functionality that would otherwise require additional, stand-alone components.
- To add a check box to a menu, create a `JCheckBoxMenuItem`. It defines several constructors. The one used in this section is `JCheckBoxMenuItem(String name)`. Here, *name* specifies the name of the item.
- The initial state of the check box is unchecked. If you want to specify the initial state, you can use this constructor `JCheckBoxMenuItem(String name, boolean state)`.

## 10.7.2.6 Creating Swing Menus

- In this case, if *state* is true, the box is initially checked. Otherwise, it is cleared.
- **JCheckBoxMenuItem** also provides constructors that let you specify an icon. Here is one example:

**JCheckBoxMenuItem(String *name*, Icon *icon*)**

- In this case, *name* specifies the name of the item and the image associated with the item is passed in *icon*. The item is initially unchecked. Other constructors are also supported.
- Check boxes in menus work like stand-alone check boxes. For example, they generate action events and item events when their state changes.



## 10.7.2.6 Creating Swing Menus

- Check boxes are especially useful in menus when you have options that can be selected and you want to display their selected/deselected status.
- A radio button can be added to a menu by creating an object of type **JRadioButtonMenuItem**. JRadioButtonMenuItem inherits JMenuItem. It provides a rich assortment of constructors. The ones used in this section are:

**JRadioButtonMenuItem(String *name*)**

**JRadioButtonMenuItem(String *name*, boolean *state*)**

- The first constructor creates an unselected radio button menu item that is associated with the name passed in *name*. The second lets you specify the initial state of the button.

## 10.7.2.6 Creating Swing Menus

- If *state* is true, the button is initially selected. Otherwise, it is deselected. Other constructors let you specify an icon. Here is one example:

**JRadioButtonMenuItem(String *name*, Icon *icon*, boolean *state*)**

- This creates a radio button menu item that is associated with the name passed in *name* and the image passed in *icon*. If *state* is true, the button is initially selected. Otherwise, it is deselected. Several other constructors are supported.
- A **JRadioButtonMenuItem** works like a stand-alone radio button, generating item and action events. Like stand-alone radio buttons, menu-based radio buttons must be put into a button group in order for them to exhibit mutually exclusive selection behavior.

## 10.7.2.6 Creating Swing Menus

- Because both `JCheckBoxMenuItem` and `JRadioButtonMenuItem` inherit `JMenuItem`, each has all of the functionality provided by `JMenuItem`. Aside from having the extra capabilities of check boxes and radio buttons, they act like and are used like other menu items.
- To try check box and radio button menu items, first remove the code that creates the Options menu in the MenuDemo example program.
- Then substitute the following code sequence, which uses check boxes for the Colors submenu and radio buttons for the Priority submenu.

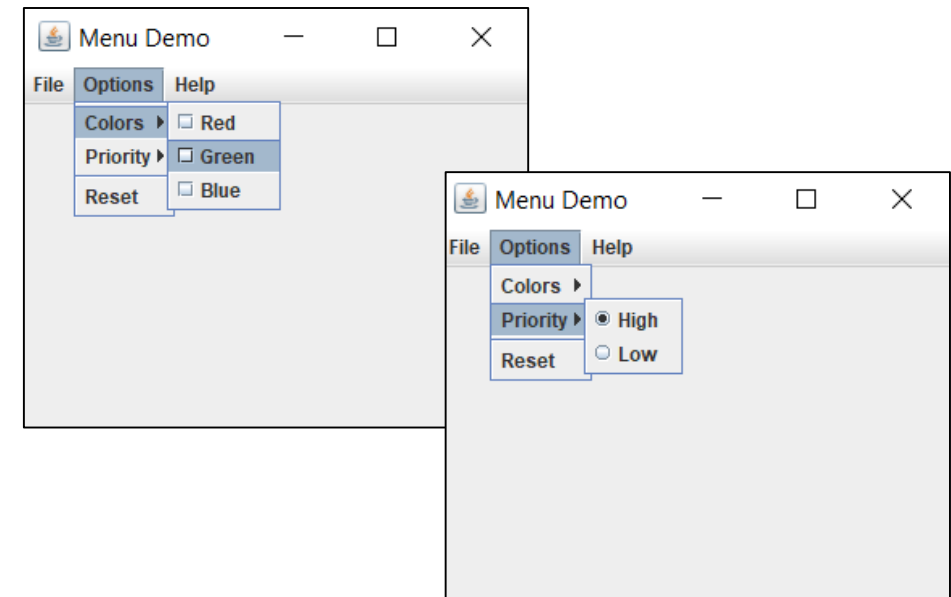
# 10.7.2.6 Creating Swing Menus

```
// Create the Options menu.
JMenu jmOptions = new JMenu("Options");
// Create the Colors submenu.
JMenu jmColors = new JMenu("Colors");
// Use check boxes for colors. This allows
// the user to select more than one color.
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");
jmColors.add(jmiRed); jmColors.add(jmiGreen);
jmColors.add(jmiBlue); jmOptions.add(jmColors);

// Create the Priority submenu.
JMenu jmPriority = new JMenu("Priority");
// Use radio buttons for the priority setting.
// This lets the menu show which priority is used.
// But ensures that one and only one priority can be selected at any one time.
// Notice that the High radio button is initially selected.
JRadioButtonMenuItem jmiHigh = new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow = new JRadioButtonMenuItem("Low");
jmPriority.add(jmiHigh); jmPriority.add(jmiLow); jmOptions.add(jmPriority);

// Create button group for the radio button menu items.
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh); bg.add(jmiLow);
// Create the Reset menu item.
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);
// Finally, add the entire options menu to
// the menu bar
jmb.add(jmOptions);
```

The effects of check box and radio button menu items



## 10.7.2.6 Creating Swing Menus

### Create a Popup Menu

- A popular alternative or addition to the menu bar is the popup menu. Typically, a popup menu is activated by clicking the right mouse button when over a component. Popup menus are supported in Swing by the **JPopupMenu** class.
- JPopupMenu has two constructors. In this section, only the default constructor as **JPopupMenu()** is used. It creates a default popup menu. The other constructor lets you specify a title for the menu.
- In general, popup menus are constructed like regular menus. First, create a JPopupMenu object, and then add menu items to it. Menu item selections are also handled in the same way by listening for action events.

## 10.7.2.6 Creating Swing Menus

- The main difference between a popup menu and regular menu is the activation process. Activating a popup menu requires three steps.
  1. You must register a listener for mouse events.
  2. Inside the mouse event handler, you must watch for the popup trigger.
  3. When a popup trigger is received, you must show the popup menu by calling `show()`.
- Let's examine each of these steps closely.
- A popup menu is normally activated by clicking the right mouse button when the mouse pointer is over a component for which a popup menu is defined. Thus, the *popup trigger* is usually caused by right-clicking the mouse on a popup menu enabled component.

## 10.7.2.6 Creating Swing Menus

- To listen for the popup trigger, implement the **MouseListener** interface and then register the listener by calling the **addMouseListener()** method.
- Two methods are very important relative to the popup menu: **void mousePressed(MouseEvent *me*)** and **void mouseReleased(MouseEvent *me*)**.
- Two events can trigger a popup menu. It is often easier to use a **MouseAdapter** to implement the MouseListener interface and simply override **mousePressed()** and **mouseReleased()**.
- The **MouseEvent** class defines several methods, but only four are commonly needed when activating a popup menu. They are **int getX()**, **int getY()**, **boolean isPopupTrigger()** and **Component getComponent()**.

## 10.7.2.6 Creating Swing Menus

- The current X,Y location of the mouse relative to the source of the event is found by calling `getX()` and `getY()`. These are used to specify the upper-left corner of the popup menu when it is displayed.
- The `isPopupTrigger()` method returns true if the mouse event represents a popup trigger and false otherwise. You will use this method to determine when to pop up the menu. To obtain a reference to the component that generated the mouse event, call `getComponent()`.
- To actually display the popup menu, call the **`void show(Component invoker, int upperX, int upperY)`** method defined by `JPopupMenu`. Here, *invoker* is the component relative to which the menu will be displayed.



## 10.7.2.6 Creating Swing Menus

- The values of *upperX* and *upperY* define the X,Y location of the upper-left corner of the menu, relative to *invoker*.
- A common way to obtain the invoker is to call `getComponent()` on the event object passed to the mouse event handler.
- Here is a program that creates a popup Edit menu to the **MenuDemo** program. This menu will have three items called Cut, Copy, and Paste.
- This program begins by constructing an instance of `JPopupMenu` and storing it in *jpu*. Then, it creates the three menu items, Cut, Copy, and Paste, in the usual way, and adds them to *jpu*. This finishes the construction of the popup Edit menu. Popup menus are not added to the menu bar or any other object.

## 10.7.2.6 Creating Swing Menus

- Next, a `MouseListener` is added by creating an anonymous inner class. This class is based on the `MouseAdapter` class, which means that the listener need only override those methods that are relevant to the popup menu: **`mousePressed()`** and **`mouseReleased()`**.
- The adapter provides default implementations of the other `MouseListener` methods. The mouse listener is added to `jfrm`. This means that a right-button click inside any part of the content pane will trigger the popup menu.
- The `mousePressed()` and `mouseReleased()` methods call **`isPopupTrigger()`** to determine if the mouse event is a popup trigger event. If it is, the popup menu is displayed by calling **`show()`**.

## 10.7.2.6 Creating Swing Menus

- The invoker is obtained by calling **getComponent()** on the mouse event. In this case, the invoker will be the content pane. The X,Y coordinates of the upper-left corner are obtained by calling **getX()** and **getY()**. This makes the menu pop up with its upper-left corner directly under the mouse pointer.
- Finally, you also need to add these action listeners to the program. They handle the action events fired when the user selects an item from the popup menu.

**jmiCut.addActionListener(this);**

**jmiCopy.addActionListener(this);**

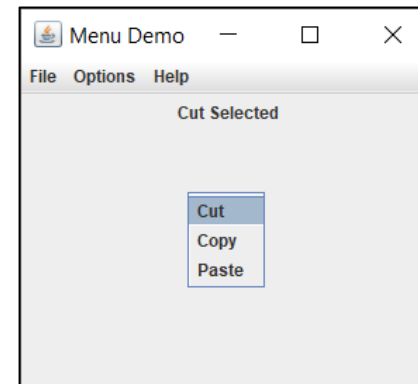
**jmiPaste.addActionListener(this);**

## 10.7.2.6 Creating Swing Menus

- After you have made these additions, the popup menu can be activated by clicking the right mouse button anywhere inside the content pane of the application.

```
JPopupMenu jpu;  
  
// Create an Edit popup menu.  
jpu = new JPopupMenu();  
// Create the popup menu items.  
JMenuItem jmiCut = new JMenuItem("Cut");  
JMenuItem jmiCopy = new JMenuItem("Copy");  
JMenuItem jmiPaste = new JMenuItem("Paste");  
// Add the menu items to the popup menu.  
jpu.add(jmiCut);  
jpu.add(jmiCopy);  
jpu.add(jmiPaste);  
  
jmiCut.addActionListener(this);  
jmiCopy.addActionListener(this);  
jmiPaste.addActionListener(this);
```

```
// Add a listener for the popup trigger.  
jfrm.addMouseListener(new MouseAdapter() {  
    public void mousePressed(MouseEvent me) {  
        if(me.isPopupTrigger())  
            jpu.show(me.getComponent(), me.getX(), me.getY());  
    }  
  
    public void mouseReleased(MouseEvent me) {  
        if(me.isPopupTrigger())  
            jpu.show(me.getComponent(), me.getX(), me.getY());  
    }  
});
```



A popup  
Edit menu

## 10.7.2.7 Toolbars in Swings

- A toolbar is a component that can serve as both an alternative and as an adjunct to a menu. A toolbar contains a list of buttons (or other components) that give the user immediate access to various program options.
- For example, a toolbar might contain buttons that select various font options, such as bold, italics, highlight, or underline. These options can be selected without needing to drop through a menu. Typically, toolbar buttons show icons rather than text, although either or both are allowed.
- Furthermore, tooltips are often associated with icon-based toolbar buttons. Toolbars can be positioned on any side of a window by dragging the toolbar, or they can be dragged out of the window entirely, in which case they become free floating.

## 10.7.2.7 Toolbars in Swings

- In Swing, toolbars are instances of the **JToolBar** class. Its constructors enable you to create a toolbar with or without a title. You can also specify the layout of the toolbar, which will be either horizontal or vertical.
- The JToolBar constructors are shown here:

**JToolBar( )**

**JToolBar(String *title*)**

**JToolBar(int *how*)**

**JToolBar(String *title*, int *how*)**

- The first constructor creates a horizontal toolbar with no title. The second creates a horizontal toolbar with the title specified by *title*. The title will show only when the toolbar is dragged out of its window.
- The third creates a toolbar that is oriented as specified by *how*. The value of *how* must be either **JToolBar.VERTICAL** or **JToolBar.HORIZONTAL**.

## 10.7.2.7 Toolbars in Swings

- The fourth constructor creates a toolbar that has the title specified by *title* and is oriented as specified by *how*.
- A toolbar is typically used with a window that uses a border layout. There are two reasons for this. First, it allows the toolbar to be initially positioned along one of the four border positions. Frequently, the top position is used. Second, it allows the toolbar to be dragged to any side of the window.
- In addition to dragging the toolbar to different locations within a window, you can also drag it out of the window. Doing so creates an *undocked* toolbar. If you specify a title when you create the toolbar, then that title will be shown when the toolbar is undocked. To add buttons to a toolbar simply call **add()**. The components are shown in the toolbar in the order in which they are added.

## 10.7.2.7 Toolbars in Swings

- Once you have created a toolbar, you *do not* add it to the menu bar (if one exists). Instead, you add it to the window container. Typically you will add a toolbar to the top position of a border layout, using a horizontal orientation.
- The component that will be affected is added to the center of the border layout. Using this approach causes the program to begin running with the toolbar in the expected location. However, you can drag the toolbar to any of the other positions.
- To illustrate the toolbar, we will add one to the **MenuDemo** program. The toolbar will present three debugging options: set a breakpoint, clear a breakpoint, and resume program execution. Three steps are needed to add the toolbar.



## 10.7.2.7 Toolbars in Swings

- First, remove **jfrm.setLayout(new FlowLayout());** line from the program. By removing this line, the **JFrame** automatically uses a border layout.
- Second, because BorderLayout is being used, change the line **jfrm.add(jlab, BorderLayout.CENTER);** that adds the label jlab to the frame.
- This line explicitly adds jlab to the center of the border layout. Explicitly specifying the center position is technically not necessary because, by default, components are added to the center when a border layout is used.
- However, explicitly specifying the center makes it clear to anyone reading the code that a border layout is being used and that jlab goes in the center. Next, add the following code, which creates the Debug toolbar.

## 10.7.2.7 Toolbars in Swings

```
// Create a Debug toolbar.
JToolBar jtb = new JToolBar("Debug");
// Load the images.
ImageIcon set = new ImageIcon("setBP.gif");
ImageIcon clear = new ImageIcon("clearBP.gif");
ImageIcon resume = new ImageIcon("resume.gif");
// Create the toolbar buttons.
JButton jbbtnSet = new JButton(set);
jbbtnSet.setActionCommand("Set Breakpoint");
jbbtnSet.setToolTipText("Set Breakpoint");
JButton jbbtnClear = new JButton(clear);
jbbtnClear.setActionCommand("Clear Breakpoint");
jbbtnClear.setToolTipText("Clear Breakpoint");
JButton jbbtnResume = new JButton(resume);
jbbtnResume.setActionCommand("Resume");
jbbtnResume.setToolTipText("Resume");
// Add the buttons to the toolbar.
jtb.add(jbbtnSet);
jtb.add(jbbtnClear);
jtb.add(jbbtnResume);
// Add the toolbar to the north position of
// the content pane.
jfrm.add(jtb, BorderLayout.NORTH);
```

- First, a JToolBar is created and given the title "Debug". Then, a set of ImageIcon objects are created that hold the images for the toolbar buttons. Next, three toolbar buttons are created.
- Each has an image, but no text. Also, each is explicitly given an action command and a tooltip. The action commands are set because the buttons are not given names when they are constructed.

## 10.7.2.7 Toolbars in Swings

- Tooltips are especially useful when applied to icon-based toolbar components because sometimes it's hard to design images that are intuitive to all users.
- The buttons are then added to the toolbar, and the toolbar is added to the north side of the border layout of the frame. Finally, add the action listeners for the toolbar, as shown here:

```
jbtnSet.addActionListener(this);
```

```
jbtnClear.addActionListener(this);
```

```
jbtnResume.addActionListener(this);
```

- Each time the user presses a toolbar button, an action event is fired, and it is handled in the same way as the other menu-related events.

## 10.7.2.7 Toolbars in Swings

### Use Actions

- Often, a toolbar and a menu item contain items in common. For example, the same functions provided by the Debug toolbar in the preceding example might also be offered through a menu selection. In such a case, selecting an option (such as setting a breakpoint) causes the same action to occur, independently of whether the menu or the toolbar was used.
- Also, both the toolbar button and the menu item would (most likely) use the same icon. Furthermore, when a toolbar button is disabled, the corresponding menu item would also need to be disabled. Such a situation would normally lead to a fair amount of duplicated, interdependent code, which is less than optimal. Fortunately, Swing provides a solution: the *action*.

## 10.7.2.7 Toolbars in Swings

- An action is an instance of the **Action** interface. Action extends the **ActionListener** interface and provides a means of combining state information with the **actionPerformed()** event handler.
- This combination allows one action to manage two or more components. For example, an action lets you centralize the control and handling of a toolbar button and a menu item. Instead of having to duplicate code, your program need only create an action that automatically handles both components.
- Because Action extends ActionListener, an action must provide an implementation of the actionPerformed() method. This handler will process the action events generated by the objects linked to the action.

## 10.7.2.7 Toolbars in Swings

- In addition to the inherited `actionPerformed()` method, **Action** defines several methods of its own.
- One of particular interest is **`void putValue(String key, Object val)`**. It sets the value of the various properties associated with an action. It assigns *val* to the property specified by *key* that represents the desired property.
- **Action** also supplies the **`Object getValue(String key)`** method that obtains a specified property. It returns a reference to the property specified by *key*.
- The key values used by `putValue()` and `getValue()` include those shown here:

## 10.7.2.7 Toolbars in Swings

Key Value	Description
static final String ACCELERATOR_KEY	Represents the accelerator property. Accelerators are specified as KeyStroke objects.
static final String ACTION_COMMAND_KEY	Represents the action command property. An action command is specified as a string.
static final String DISPLAYED_MNEMONIC_INDEX_KEY	Represents the index of the character displayed as the mnemonic. This is an Integer value.
static final String LARGE_ICON_KEY	Represents the large icon associated with the action. The icon is specified as an object of type Icon.
static final String LONG_DESCRIPTION	Represents a long description of the action. This description is specified as a string.
static final String MNEMONIC_KEY	Represents the mnemonic property. A mnemonic is specified as a KeyEvent constant.
static final String NAME	Represents the name of the action (which also becomes the name of the button or menu item to which the action is linked). The name is specified as a string.

## 10.7.2.7 Toolbars in Swings

- For example, to set the mnemonic to the letter X, call to `actionOb.putValue(MNEMONIC_KEY, new Integer(KeyEvent.VK_X));`
- One Action property that is not accessible through `putValue()` and `getValue()` is the enabled/disabled status. For this, you use the **void `setEnabled(boolean enabled)`** and **boolean `isEnabled()`** methods.
- For **`setEnabled()`**, if *enabled* is true, the action is enabled. Otherwise, it is disabled. If the action is enabled, **`isEnabled()`** returns true. Otherwise, it returns false.
- Although you can implement all of the Action interface yourself, you won't usually need to. Instead, Swing provides a partial implementation called **AbstractAction** that you can extend.



## 10.7.2.7 Toolbars in Swings

- By extending `AbstractAction`, you need implement only one method: **`actionPerformed()`**. The other Action methods are provided for you.
- `AbstractAction` provides three constructors. The one used in this section is **`AbstractAction(String name, Icon image)`**. It constructs an `AbstractAction` that has the name specified by *name* and the icon specified by *image*.
- Once you have created an action, it can be added to a `JToolBar` and used to construct a `JMenuItem`. To add an action to a `JToolBar`, use **`void add(Action actObj)`**. Here, *actObj* is the action that is being added to the toolbar. The properties defined by *actObj* are used to create a toolbar button.

## 10.7.2.7 Toolbars in Swings

- To create a menu item from an action, use **JMenuItem(Action actObj)** constructor. Here, *actObj* is the action used to construct a menu item according to its properties.
- To illustrate the benefit of actions, we will use them to manage the **Debug toolbar** created in the previous section. We will also add a Debug submenu under the Options main menu.
- The Debug submenu will contain the same selections as the Debug toolbar: Set Breakpoint, Clear Breakpoint, and Resume. The same actions that support these items in the toolbar will also support these items in the menu. Therefore, instead of having to create duplicate code to handle both the toolbar and menu, both are handled by the actions.

## 10.7.2.7 Toolbars in Swings

- Begin by creating an inner class called **DebugAction** that extends AbstractAction, as shown here:

```
// A class to create an action for the Debug menu
// and toolbar.
class DebugAction extends AbstractAction {
    public DebugAction(String name, Icon image, int mnem, int accel, String tTip)
    {
        super(name, image);
        putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(accel,
            InputEvent.CTRL_DOWN_MASK));
        putValue(MNEMONIC_KEY, new Integer(mnem));
        putValue(SHORT_DESCRIPTION, tTip);
    }
    // Handle events for both the toolbar and the
    // Debug menu.
    public void actionPerformed(ActionEvent ae) {
        String comStr = ae.getActionCommand();
        jlab.setText(comStr + " Selected");
        // Toggle the enabled status of the
        // Set and Clear Breakpoint options.
        if(comStr.equals("Set Breakpoint")) {
            clearAct.setEnabled(true);
            setAct.setEnabled(false);
        }
        else if(comStr.equals("Clear Breakpoint")) {
            clearAct.setEnabled(false);
            setAct.setEnabled(true);
        }
    }
}
```

## 10.7.2.7 Toolbars in Swings

- DebugAction extends AbstractAction. It creates an action class that will be used to define the properties associated with the Debug menu and toolbar. Its constructor has five parameters that let you specify Name, Icon, Mnemonic, Accelerator and Tooltip items.
- The first two are passed to **AbstractAction**'s constructor via **super**. The other three properties are set through calls to **putValue()**.
- The **actionPerformed()** method of **DebugAction** handles events for the action. This means that when an instance of **DebugAction** is used to create a toolbar button and a menu item, events generated by either of those components are handled by the actionPerformed() method in DebugAction.

## 10.7.2.7 Toolbars in Swings

- This handler displays the selection in jlab. In addition, if the Set Breakpoint option is selected, then the Clear Breakpoint option is enabled and the Set Breakpoint option is disabled.
- If the Clear Breakpoint option is selected, then the Set Breakpoint option is enabled and the Clear Breakpoint option is disabled. This illustrates how an action can be used to enable or disable a component.
- When an action is disabled, it is disabled for all uses of that action. In this case, if Set Breakpoint is disabled, then it is disabled both in the toolbar and in the menu. Next, add **DebugAction setAct**, **DebugAction clearAct** and **DebugAction resumeAct** DebugAction instance variables to **MenuDemo**.

## 10.7.2.7 Toolbars in Swings

- Next, create three **ImageIcons** that represent the Debug options, as shown here:

```
// Load the images for the actions.
ImageIcon setIcon = new ImageIcon("setBP.gif");
ImageIcon clearIcon = new ImageIcon("clearBP.gif");
ImageIcon resumeIcon = new ImageIcon("resume.gif");
```

- Now, create the actions that manage the Debug options, as shown here:

```
// Create actions.
setAct = new DebugAction("Set Breakpoint", setIcon, KeyEvent.VK_S,
    KeyEvent.VK_B, "Set a break point.");
clearAct = new DebugAction("Clear Breakpoint", clearIcon, KeyEvent.VK_C,
    KeyEvent.VK_L, "Clear a break point.");
resumeAct = new DebugAction("Resume", resumeIcon, KeyEvent.VK_R,
    KeyEvent.VK_R, "Resume execution after breakpoint.");

// Initially disable the Clear Breakpoint option.
clearAct.setEnabled(false);
```

## 10.7.2.7 Toolbars in Swings

- The accelerator for Set Breakpoint is B and the accelerator for Clear Breakpoint is L. The reason these keys are used rather than S and C is that these keys are already allocated by the File menu for Save and Close.
- However, they can still be used as mnemonics because each mnemonic is localized to its own menu. Also the action that represents Clear Breakpoint is initially disabled. It will be enabled only after a breakpoint has been set.
- Next, use the actions to create buttons for the toolbar and then add those buttons to the toolbar, as shown here:

## 10.7.2.7 Toolbars in Swings

```
// Create the toolbar buttons by using the actions.
JButton jbtnSet = new JButton(setAct);
JButton jbtnClear = new JButton(clearAct);
JButton jbtnResume = new JButton(resumeAct);

// Create a Debug toolbar.
JToolBar jtb = new JToolBar("Breakpoints");

// Add the buttons to the toolbar.
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// Add the toolbar to the north position of
// the content pane.
jfrm.add(jtb, BorderLayout.NORTH);
```

- Finally, create the Debug menu, as shown next:

```
// Now, create a Debug menu that goes under the Options
// menu bar item. Use the actions to create the items.
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);
```

- After making these changes and additions, the actions that you created will be used to manage both the Debug menu and the toolbar. Thus, changing a property in the action (such as disabling it) will affect all uses of that action.