Introduction to AWT

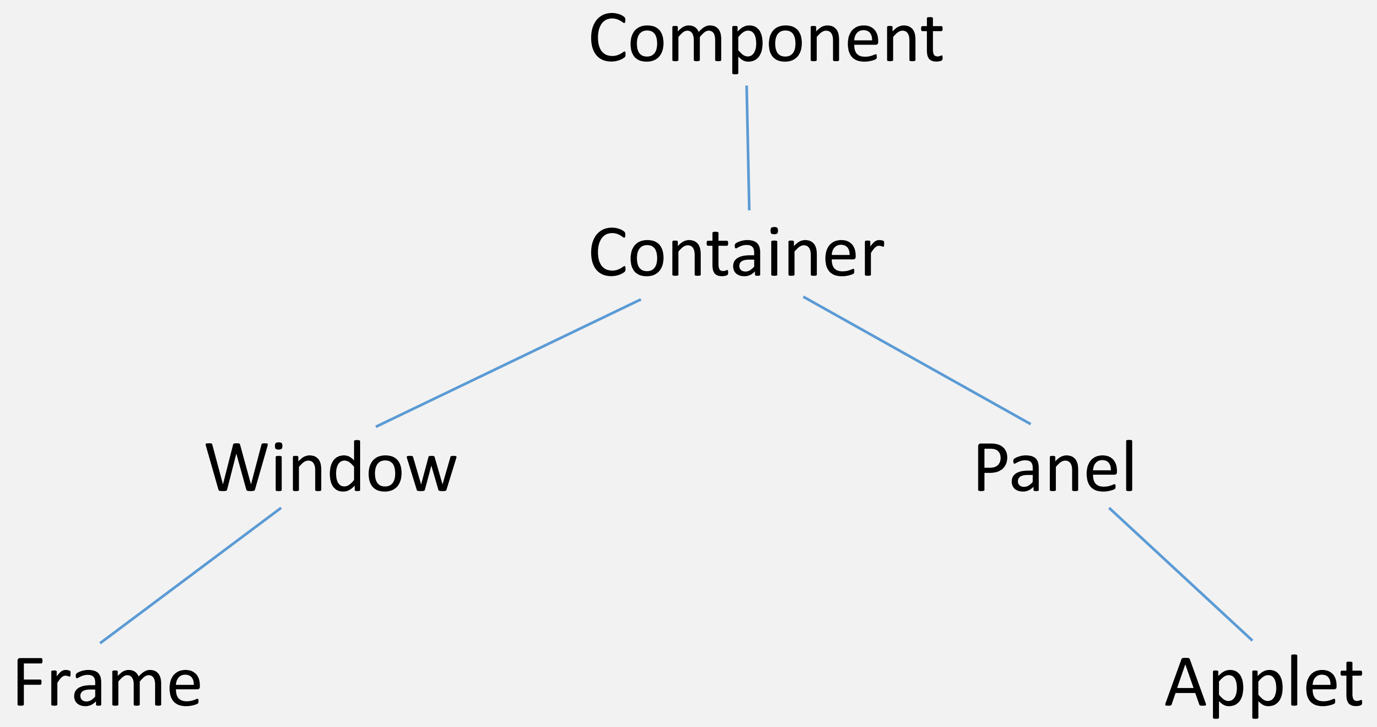
AWT Classes

- The AWT classes are contained in the java.awt package.

Window Fundamentals

- AWT defines windows according to a class hierarchy that adds functionality & specificity in each level.
- The two most common windows are
  - those derived from Panel
  - those derived from Frame

- The following fig. shows the class hierarchy for Panel and Frame



<u>Component</u>

- All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.
- Responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.

<u>Container</u>

- subclass of Component
- Responsible for laying out(i.e. positioning) any components that it contain.
- It does this through the use of various layout managers.

Panel

- Concrete subclass of Container.
- No new methods; it simply implements Container.
- Panel is the superclass for Applet.
- Panel is a window that does not contain a title bar, menu bar or border. (AppletViewer provides this; browser won’t)
- Components can be added to a Panel object by its add( ) method.
- Once added, they can be positioned and resized manually using the setLocation( ), setSize( ), or setBounds( ) methods defined by Component.

<u>Window</u>

- creates top-level window.
- Window objects generally are not created directly; instead we will use a subclass of Window called Frame.

<u>Frame</u>

- subclass of Window and has a title bar, menu bar, border and resizing corners.
- Frame object can be created from within an applet window
  - it will contain a message “Java Applet Window”
- When a Frame window is created by a program ; a normal window is created.

<u>Canvas</u>

- not part of hierarchy.
- Canvas encapsulates a blank window upon which you can draw.

Working with Frame Windows

Frame’s Constructors

- Frame( )
    - creates a standard window that does not contain a title.
- Frame(String *title*)
    - creates a window with the title specified by *title*.

Setting the Window’s Dimensions

- void setSize(int newWidth, int newHeight)
- void setSize(Dimension newSize)
    - set the dimensions of the window
- Dimension getSize( )
    - obtain the current size of a window

Hiding and Showing a Window

- After a frame window has been created, it will not be visible until you call setVisible( )

    void setVisible(boolean visibleFlag)

        -> visible if true otherwise hidden

Setting a Window's Title

- void setTitle(String newTitle)

Closing a Frame Window

- Remove window from the screen by calling setVisible(false), when it is closed.
- If Frame window is a child window, to intercept window-close event,
    - implement windowClosing( ) method of the WindowListener interface.
    - Inside windowClosing( ), remove the window from the screen.

```
import java.awt.Frame;
public class SampleFrame extends Frame {
        SampleFrame(String title) {
                super( );
                this.setTitle(title);
                this.setSize(300,200);
                this.setVisible(true);
        }
        public static void main(String args[ ]) {
                SampleFrame sf = new SampleFrame(“My Window”);
        }
}
```

Displaying Information within a Window

Working with Graphics

- All graphics are drawn relative to a window.
- The origin of each window is at the top-left corner and is 0,0
- Coordinates are specified in pixels.
- All output to a window takes place through a graphics context.
- A graphics context is encapsulated by the Graphics class and is obtained in 2 ways
  - It is passed to an applet when one of its various methods, such as paint( ) or update( ) is called.
  - It is returned by the getGraphics( ) method of Component.
- The Graphics class defines a number of drawing functions.

Drawing Lines

- Lines are drawn by means of the drawLine( ) method.

```
        void drawLine(int startX, int startY, int endX, int endY)
```

```
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines.class"width=300 height=200>
</applet>
*/
public class Lines extends Applet {
        public void paint(Graphics g) {
                g.drawLine(0,0,100,100);
                g.drawLine(40,25,250,180);
        }
}
```

Drawing Rectangles

- void drawRect(int top, int left, int width, int height)
- void fillRect(int top, int left, int width, int height)
  - top, left -> upper-left corner of the rectangle
  - width, height -> dimensions of the rectangle

Rounded Rectangles

- void drawRoundRect(int top, int left, int width, int height, int xDiam, int yDiam)
- void fillRoundRect(int top, int left, int width, int height, int xDiam, int yDiam)

```
eg) g.drawRect(10,10,60,50);
    g.fillRect(10,10,60,50);
    g.drawRoundRect(10,10,60,50,15,15);
```

Drawing Ellipses and Circles

- void drawOval(int top, int left, int width, int height)
- void fillOval(int top, int left, int width, int height)

Drawing Arcs

- void drawArc(int top, int left, int width, int height, int startAngle, int sweepAngle)
- void fillArc(int top, int left, int width, int height, int startAngle, int sweepAngle)
  - The arc is drawn from startAngle through the angular distance specified by sweepAngle.
  - Angles are specified in degrees; zero degrees is on the horizontal, at 3’0 clock position.
  - The arc is drawn counter clockwise if sweepAngle is positive and clockwise if sweepAngle is negative.

```
eg. g.drawArc(10,40,70,70,0,75)
    g.fillArc(10,40,70,70,0,75)
```

Working with Color

- Color is encapsulated by the Color class.
- Color defines several constants to specify a number of common colors.
- We can create our own colors, using one of the Color constructors.

```
        Color( int red, int green, int blue)
        Color( int rgbValue)
        Color( float red, float green, float blue)
                -> these values must be between 0 and 255.
eg. new Color(255, 100, 100); //light red
```

Color Methods

<u>Using Hue, Saturation and Brightness</u>

- The Hue-Saturation-Brightness(HSB) color model is an alternative to Red-Green-Blue(RGB) for specifying particular colors.
- Hue
  - is a wheel of Color
  - specified with a number between 0.0 and 1.0
  - colors approximately red, orange, yellow, green, blue, indigo and violet.
- Saturation
  - scale ranging from 0.0 to 1.0 representing light pastels to intense hues.
- Brightness
  - range from 0.0 to 1.0 where 1 is bright white and 0 is black.

- Color supplies two methods to convert between RGB and HSB

  static int HSBtoRGB(float hue, float saturation, float brightness)

  static float[ ] RGBtoHSB(int red, int green, int blue, float values[ ])

- RGBtoHSB( ) returns a float array of HSB values corresponding to RGB integers
  - the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

getRed( ), getGreen( ), getBlue( )

- To obtain the red, green and blue components of a color independently

Eg. int getRed( )

int getGreen( )

int getBlue( )

getRGB( )

- To obtain a packed RGB representation of a color.

  eg. int getRGB( )

Setting the current Graphics Color

- void setColor(Color newColor)
  - newColor -> specifies the new drawing color
- Color getColor( )
  - to obtain the current color

Setting the Paint Mode

- The paint mode determines how objects are drawn in a window.
- By default, new output to a window overwrites any pre-existing contents
- However it is possible to have new objects XORed into the window by using setXORMode( )

```
void setXORMode(Color xorColor)
```

xorColor -> color that will be XORed to the window when an object is drawn

- To return to overwrite mode, call setPaintMode( )

```
void setPaintMode( )
```

Working with Fonts

- The AWT supports multiple type fonts.
- Fonts have a family name, a logical font name and a face name.
- family name -> general name of font. Eg. Courier
- logical name -> category of font. Eg. Monospaced
- face name -> specific font. Eg. Courier Italic
- Fonts are encapsulated by the Font class.
- Methods (refer book)
- The Font class defines these variables
  - String name – name of the font
  - float pointSize – size of the font in points
  - int size – size of the font in points
  - int style – font style

Determining the Available Fonts

- String[ ] getAvailableFontFamilyNames( )
- Font[ ] getAllFonts( )

  -> members of GraphicsEnvironment class

  -> need a reference of GraphicsEnvironment class to call them.

  -> reference can be obtained by using the
     getLocalGraphicsEnvironment( ) static method.

  static GraphicsEnvironment getLocalGraphicsEnvironment( )

  Eg.

  String Fontlist[ ];

  GraphicsEnvironment ge =
  GraphicsEnvironment.getLocalGraphicsEnvironment( );

  Fontlist = ge.getAvailableFontFamilyNames( );

Creating and Selecting a Font

- Font constructor

  Font(String fontName, int fontStyle, int pointSize)

- fontName
  - name of the desired font
  - Dialog, DialogInput, SansSerif, Serif and Monospaced supported by all Java environments
  - default -> Dialog
- fontStyle
  - style of the font
  - 3 constants (Font.PLAIN, Font.BOLD, Font.ITALIC)
    - can be ORed together
    - eg. Font.BOLD | Font.ITALIC
- pointSize -> size of the font in points

Obtaining Font Information

- Font getFont( )
  - To obtain information about the currently selected font.
  - defined by the Graphics class

AWT Controls

- The AWT supports the following types of controls
  - Labels
  - Push Buttons
  - Check boxes
  - Choice lists
  - Lists
  - Scroll bars
  - Text editing

    -> subclasses of Component

Adding and Removing Controls

- **To add a control to a window**
  - first create an instance of the desired control.
  - then add it to a window by calling add( )

    Component add(Component *compObj*)

    compObj -> instance of the control to add

- **To remove a control**

  void remove(Component *obj*)

  obj -> control to be removed

  removeAll( ) -> to remove all controls

- Except for labels, which are passive controls, all controls generate events when they are accessed by user.
- Example

Labels

- A label is an object of type **Label**, and it contains a string which it displays.
- Passive controls – do not support any interaction with the user.
- Label defines the following constructors

  Label( )

  Label(String *str*) – string is left-justified

  Label(String *str,* int *how*)

  - alignment specified by *how*

  - value of *how* must be one of the 3 constants

    - Label.LEFT, Label.RIGHT, Label.CENTER

- To set or change the text in a label, use the setText( ) method.

  void setText(String *str*)

- To obtain the current label, call getText( )

  String getText( )

- To set the alignment of the string within the label, call setAlignment( )

  void setAlignment(int *how*)

- To obtain the current alignment, call getAlignment( )

  int getAlignment( )

- Example

Buttons

- A push button is a component that contains a label and that generates an event when it is pressed.
- Push buttons are objects of type **Button.**
- Button defines these two constructors

  Button( )

  Button(String *str*) – contains str as a label
- To set the label of a button, call setLabel( )

  void setLabel(String *str*)
- To retrieve the label of a button, call getLabel( )

  String getLabel( )

Handling Buttons

- Each time a button is pressed, an **action event is generated**.
- This is sent to any listeners that previously registered an interest in receiving event notifications from that component.
- Each listener implements the **ActionListener interface.**
- This interface defines the **actionPerformed( )** method, which is called when an event occurs.
- An **ActionEvent object** is supplied as the argument to this method.
- It contains both a reference to the button that generated the event and a reference to the string that is the label of the button.
- Example

Check Boxes

- A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not.
- There is a label associated with each check box that describes what option the box represents.
- Check boxes can be used individually or as part of a group.
- Check boxes are objects of the **Checkbox** class.
- Checkbox supports these constructors

  Checkbox( ) – label is initially blank; state of the checkbox is unchecked.

  Checkbox(String *str*) – label is specified by str; state of the checkbox is unchecked.

Checkbox(String *str*, boolean *on*)

- if *on* is true, the checkbox is initially checked; otherwise cleared.

Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*)

- group is specified by *cbGroup*; if this checkbox is not part of a group, then *cbGroup* must be null.

Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*)

- To retrieve the current state of a checkbox, call getState( )

  boolean getState( )

- To set the state of a checkbox, call setState( )

  void setState(boolean *on*)

- To obtain the current label associated with a checkbox, call getLabel( )

  String getLabel( )

- To set the label, call setLabel( )

  void setLabel(String *str*)

Handling Check Boxes

- Each time a check box is selected or deselected, an item event is generated.
- This is sent to the registered listeners.
- Each listener implements the ItemListener interface which defines the itemStateChanged( ) method.
- Example

CheckboxGroup

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- These check boxes are often called radio buttons.
- To create a set of mutually excusive check boxes, first define the group to which they will belong and then specify that group when constructing the checkboxes.
- Check box groups are objects of type **CheckboxGroup**
- Only default constructor – CheckboxGroup( ) -> creates an empty group.

- To determine which checkbox in a group is currently selected, call getSelectedCheckbox( )

  Checkbox getSelectedCheckbox( )

- To set a check box, call setSelectedCheckbox( )

  void setSelectedCheckbox(Checkbox *which*)

  which -> checkbox that has to be selected; previously selected check box will be turned off

- Example

Choice Controls

- The **Choice** class is used to create a pop-up list of items from which the user may choose.
- Thus, a choice control is a form of menu.
- When inactive, a Choice component takes up only enough space to show the currently selected item.
- When the user clicks on it, the whole list of choices pops up, and a new selection can be made.
- Choice only defines the default constructor, which creates an empty list.
- To add a selection to the list, call add( ). Items are added to the list in the order in which calls to add( ) occur.

    void add(String *name*)

        name -> name of the item being added;

- To determine which item is currently selected, call **getSelectedItem( )** or **getSelectedIndex( )**

  String getSelectedItem( )

  -> returns a string containing the name of the item

  int getSelectedIndex( )

  -> returns the index of the item. The first item is at index 0

- To obtain the number of items in the list, call **getItemCount( )**

  int getItemCount( )

- To set the currently selected item with either a zero-based index or a string that will match a name in the list, call select( )

  void select(int *index*)

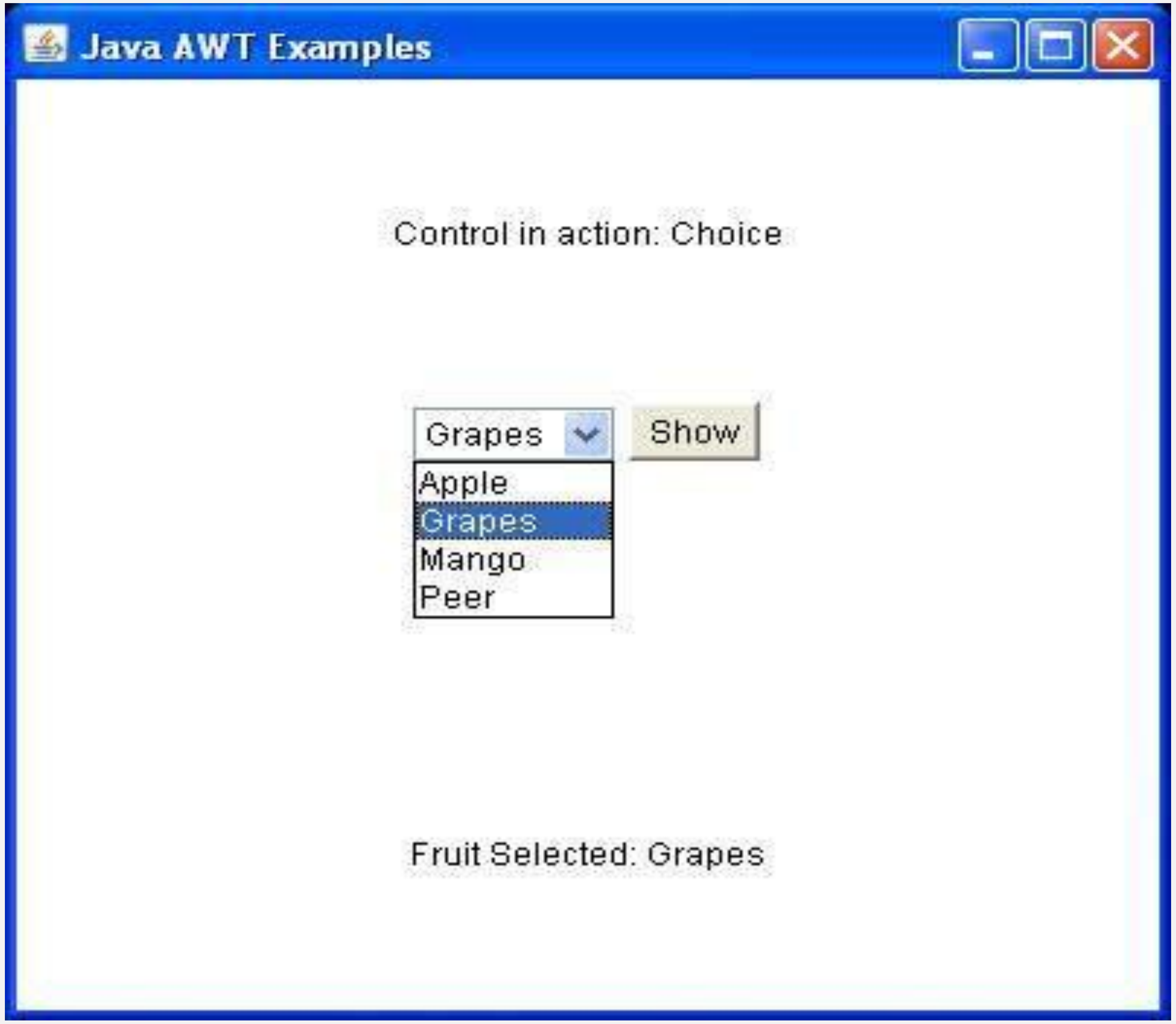
  void select(String *name*)

- To obtain the name associated with the item at an index, call getItem( )

  String getItem(int *index*)

Handling Choice Lists

- Each time a choice is selected, an item event is generated.
- The registered listeners implements the **ItemListener** interface that defines the **itemStateChanged( )** method.
- Example



Control in action: Choice

Fruit Selected: Grapes

Lists

- The **List** class provides a compact, multiple-choice, scrolling selection list.
- A List object can be constructed to show any number of choices in the visible window.
- It can also be created to allow multiple selections.
- List provides 3 constructors

  List( ) – creates a List control that allows only one item to be selected at any one time.

  List(int *numRows*) – *numRows* specifies the number of entries in the list that will always be visible.

  List(int *numRows*, boolean *multipleSelect*) – if *multipleSelect* is true, then the user may select two or more items at a time. If it is false, only one item may be selected.

- To add a selection to the list, call add( )

  void add(String *name*) – *name* is the name of the item added to the list.

  void add(String *name*, int *index*) – adds the item at the index specified by *index.*

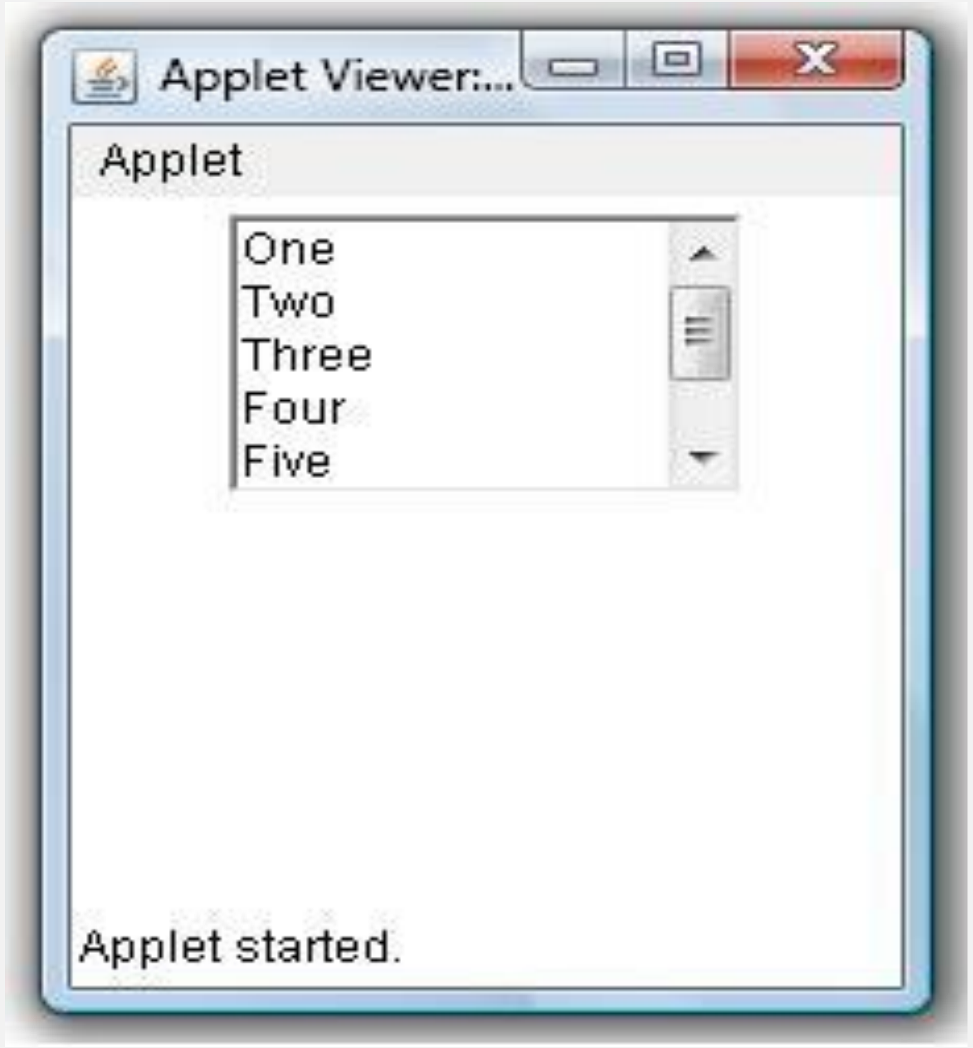
- For lists that allow only single selection, to determine which item is currently selected, call getSelectedItem( ) or getSelectedIndex( )
- For lists that allow multiple selection, call getSelectedItems( ) or getSelectedIndexes( )

  String[ ] getSelectedItems( )

  int[ ] getSelectedIndexes( )

Handling Lists

- Each time a List item is double-clicked, an **ActionEvent** object is generated.
- Its **getActionCommand( )** method can be used to retrieve the name of the newly selected item.
- Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated.
- Its **getStateChange( )** method can be used to determine whether a selection or deselection triggered this event.
- **getItemSelectable( )** returns a reference to the object that triggered the event.
- Example



```
One
Two
Three
Four
Five
```

Applet started.

TextFields

- The **TextField** class implements a single-line text-entry area, usually called an edit control.
- **TextField** is a subclass of **TextComponent.**
- TextField defines the following constructors

TextField( ) – creates a default text field

TextField(int *numChars*) – creates a text field that is *numChars* wide.

TextField(String *str*) – initializes the text field with the string contained in *str*.

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

- To obtain the string currently contained in the text field, call **getText( )**

  String getText( )

- To set the text, call **setText( )**

  void setText(String *str*) – *str* is the new string

- To select a portion of the text in a text field, call **select( )**

  void select(int *startIndex*, int *endIndex*)

  - selects the characters beginning at *startIndex* and ending at *endIndex – 1*

- To obtain the currently selected text, call **getSelectedText( )**

  String getSelectedText( )

- To control whether the contents of a text field may be modified by the user or not, call **setEditable( )**

    void setEditable(boolean *canEdit*)

        - if *canEdit* is true, the text may be changed. If it is false, the text cannot be altered.

- To determine the editability, call **isEditable( )**

    boolean isEditable( )

        - returns true if the text may be changed and false if not.

- To disable the echoing of characters as they are typed(eg.while typing passwords), call **setEchoChar( )** method.

  void setEchoChar(char *ch*)

  - *ch* specifies the character to be echoed.

- To check a text field to see if it is in this mode, call **echoCharIsSet( )** method.

  boolean echoCharIsSet( )

- To retrieve the echo character, call **getEchoChar( )** method.

  char getEchoChar( )

- Example

TextArea

- Sometimes, a single line of text input is not enough for a given task. to handle these situations, the AWT includes a multiline editor called TextArea.
- Constructors

  TextArea( )

  TextArea(int *numLines*, int *numChars*)

  -> *numLines* specifies the height, in lines, of the text area

  -> *numChars* specifies its width, in characters.

  TextArea(String *str*)

  -> initial text can be specified by *str*

TextArea(String *str*, int *numLines*, int *numChars*)

TextArea(String *str*, int *numLines,* int *numChars*, int *sBars*)

-> *sBars* specify the scroll bars. *sBars* must be one of these values

SCROLLBARS_BOTH

SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY

SCROLLBARS_VERTICAL_ONLY

- TextArea is a subclass of **TextComponent.**
- It supports the getText( ), setText( ), getSelectedText( ), select, isEditable( ), and setEditable( )

- TextArea adds the following methods

  void append(String *str*)

  -> appends the string specified by *str* to the end of the current text.

  void insert(String *str*, int *index*)

  -> inserts the string passed in *str* at the specified index.

  void replaceRange(String *str*, int *startIndex*, int *endIndex*)

  -> replaces the characters from *startIndex* to *endIndex -1,* with the replacement text passed in *str*.

Layout managers

- Each **Container** object has a layout manager associated with it.
- A layout manager is an instance of any class that implements the **LayoutManager** interface.
- The layout manager is set by the **setLayout( )** method.
- If no call to setLayout( ) is made, then the default layout manager is used.
- The setLayout( ) method has the following form

  void setLayout(LayoutManager *layoutObj*)

  layoutObj -> reference to the desired layout manager
- Java has several predefined LayoutManager classes.

FlowLayout

- **FlowLayout** is the default layout manager.
- FlowLayout implements a simple layout style, which is similar to how words flow in a text editor.
- direction -> by default it is left to right, top to bottom
- By default, components are laid out line-by-line beginning at the upper-left corner.
- A small space is left between each component, above and below, as well as left and right.
- Constructors

FlowLayout( ) -> creates the default layout, which centers components and leaves 5 pixels of space between each component.

FlowLayout(int *how*)

-> valid values for *how* are as follows:

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT

FlowLayout.LEADING

FlowLayout.TRAILING

FlowLayout(int *how*, int *horz*, int *vert*)

-> *horz* and *vert* allows to specify the horizontal and vertical space between components

BorderLayout

- **BorderLayout** class implements a common layout style for top-level windows
- It has four narrow, fixed-width components at the edges and one large area in the center.
- The four sides are referred to as north, south, east and west. The middle area is called the center
- Constructors

  BorderLayout( ) -> default border layout

  BorderLayout(int *horz*, int *vert*)

- When adding components to border layout, call the following add( ) method

  void add(Component compObj, Object region)

  compObj – component to be added

  region – specifies where the component will be added

- BorderLayout defines the following constants that specify the regions

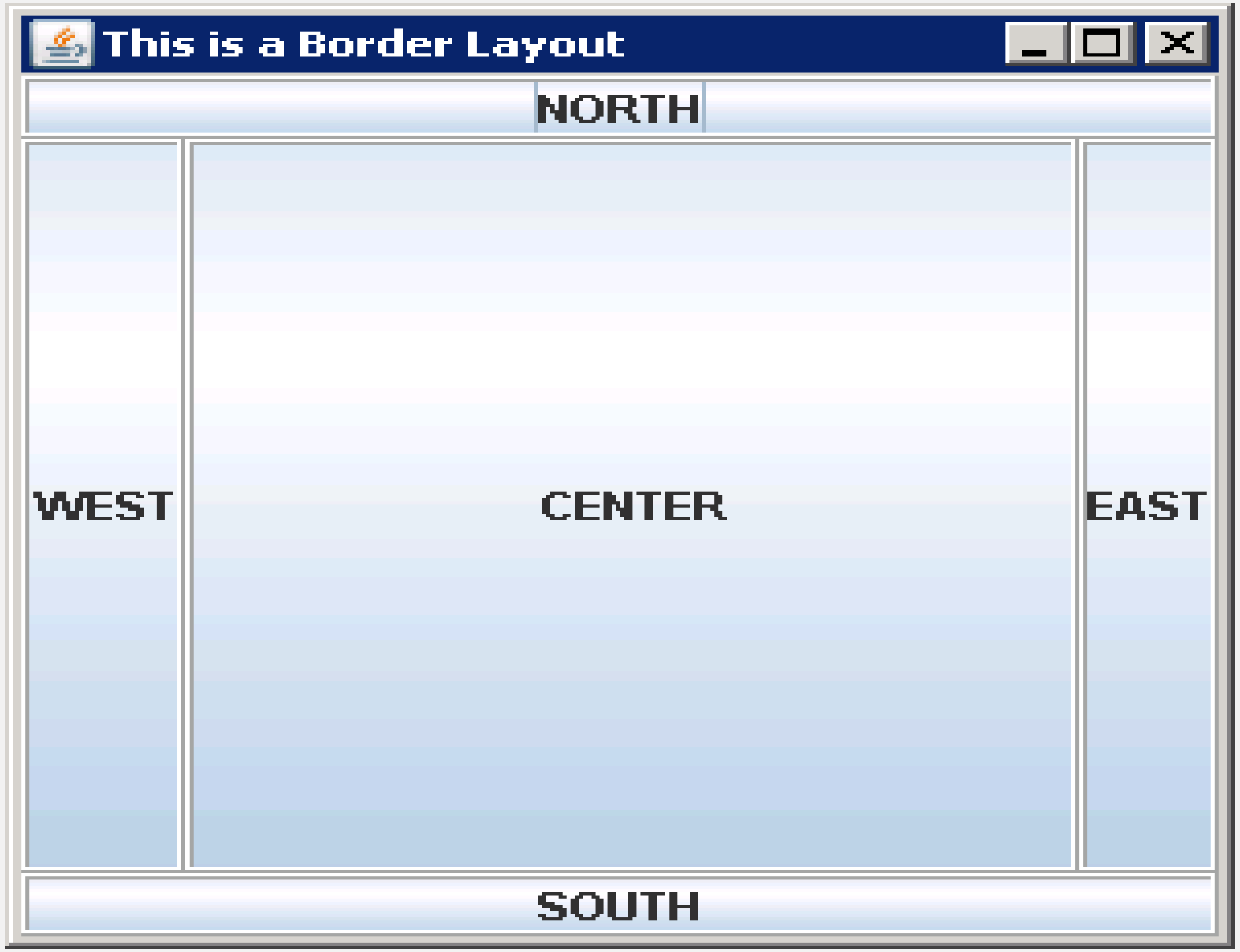
  BorderLayout.CENTER

  BorderLayout.SOUTH

  BorderLayout.EAST

  BorderLayout.WEST

  BorderLayout.NORTH


This is a Border Layout
NORTH
WEST
CENTER
EAST
SOUTH

<u>GridLayout</u>

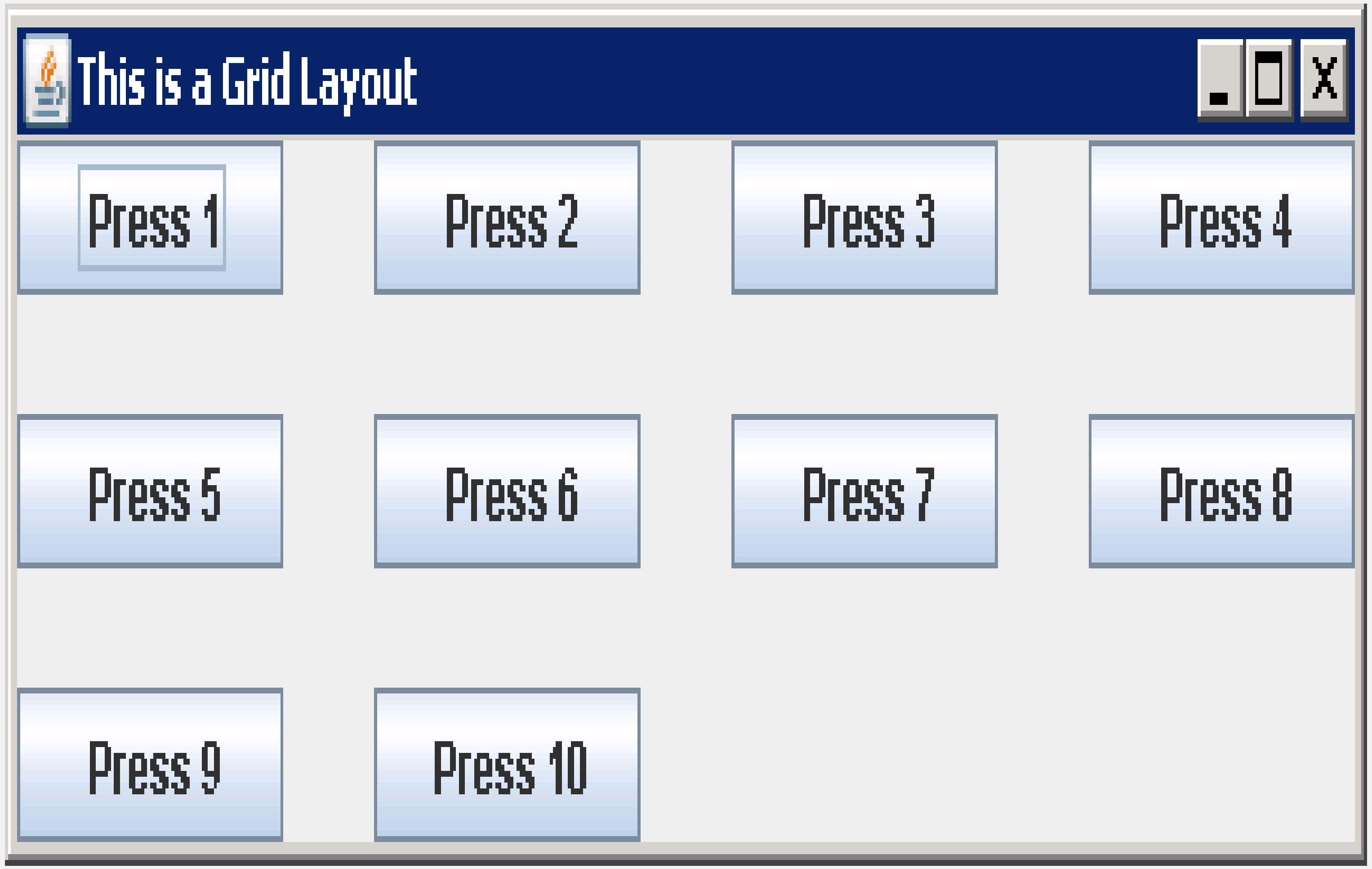
- **GridLayout** lays out components in a two-dimensional grid.
- Constructors

  GridLayout( ) – creates a single-column grid layout

  GridLayout(int *numRows*, int *numColumns*)

  -> creates a grid layout with the specified number of rows and columns

  GridLayout(int *numRows*, int *numColumns,* int *horz*, int *vert*)
- Specifying *numRows* as zero allows for unlimited-length columns
- Specifying *numColumns* as zero allows for unlimited-length rows


This is a Grid Layout
Press 1
Press 2
Press 3
Press 4
Press 5
Press 6
Press 7
Press 8
Press 9
Press 10

SWING

- Swing was a response to deficiencies present in Java’s original GUI system: AWT
- Limitations of AWT
  - The look and feel of a component in AWT is defined by the platform, not by Java.
  - AWT components are heavyweight, because they use native code resources

Swing is built on the AWT

- Swing eliminates many limitations in AWT, but does not replace it.
- Instead, Swing is built on the foundation of the AWT.
- Swing also uses the same event handling mechanism as the AWT.

Two key Swing features

1. **Swing components are lightweight**
   - Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers.
   - So the look and feel of each component is determined by Swing, not by the underlying operating system.
   - This means that each component will work in a consistent manner across all platforms.
2. **Swing supports a Pluggable Look and Feel(PLAF)**
   - It is 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.

The MVC Connection

- In general, a visual component is a composite of three distinct parts
  - The state information associated with the component **(Model)**

    -> For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.
  - The way that the component looks when rendered on the screen **(View)**
  - The way the component reacts to the user **(Controller)**

    -> 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.

- Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the **UI delegate.**
- Swing's approach is called the **Model-Delegate architecture** or the **Separable Model architecture.**
- Because the view(look) and controller(feel) are separate from model, the look and feel can be changed without affecting how the component is used within a program.
- Most Swing components contain two objects
  - The first represents the **model.** Models are defined by **interfaces.** eg. Model for a button is defined by ButtonModel interface.
  - The second represents the **UI delegate**(view & controller). UI delegates are **classes** that inherit ComponentUI. eg. UI delegate for a button is ButtonUI

Components and Containers

- A Swing GUI consists of two key items: components and containers.
- A component is an independent visual control, such as a push button or slider.
- A container holds a group of components.

Components

- In general, Swing components are derived from **JComponent** class.
- JComponent inherits the AWT classes **Component** and **Container.**
- All of Swing's components are represented by classes defined within the package **javax.swing**
- Swing Component classes (refer book table)

<u>Containers</u>

- Swing defines two types of containers.
  1. **Top-level containers**

     - **JFrame, JApplet, JWindow and JDialog**.

     - Inherit the AWT classes **Component** and **Container**

     - The most commonly used container for applications is JFrame.

     - The one used for applets is JApplet.
  2. **Lightweight containers**

     - Lightweight containers do inherit **JComponent.**

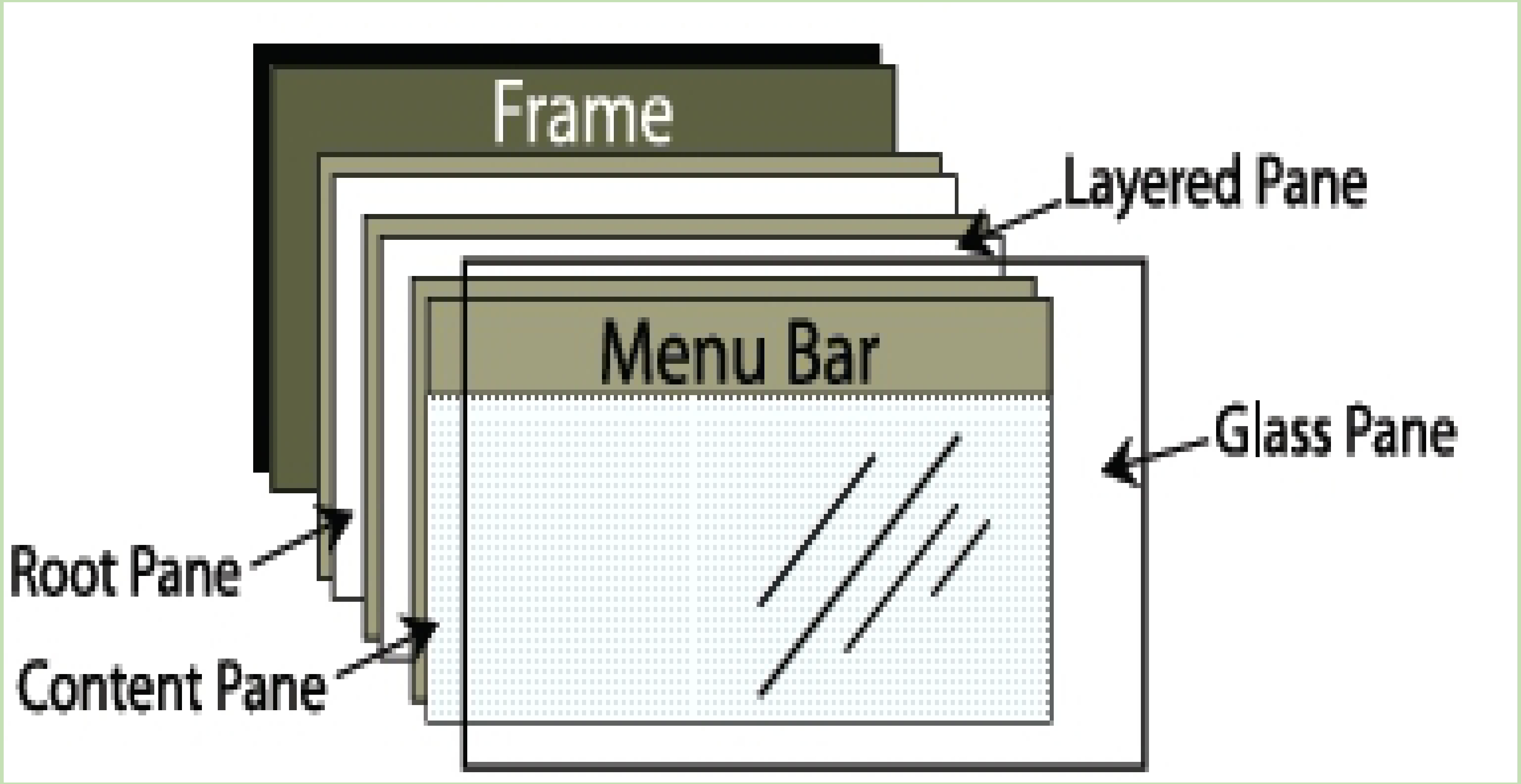
     - An example of a light weight container is **JPanel.**

JApplet

- Fundamental to Swing is the JApplet class, which extends Applet.
- JApplet support various “panes” such as the,
  - content pane
  - glass pane
  - root pane
- When adding a component, instead of invoking the add( ) method of the applet, **call add( ) for the content pane of the JApplet class**

  Container getContentPane( )
- The add( ) method of Container can be used to add a component to a content pane

  void add(comp)


Frame
Layered Pane
Menu Bar
Glass Pane
Root Pane
Content Pane

Icons and Labels

ImageIcon class

- Constructors

  ImageIcon(String *filename*)

  ImageIcon(URL *url*)

- Methods

  int getIconHeight( )

  int getIconWidth( )

  void paintIcon(Component *comp*, Graphics *g*, int *x*, int *y*)

JLabel class

- extends **JComponent**
- can display text/or an icon
- Constructors

    JLabel(Icon *i*)

    Label(String *s*)

    JLabel(String *s*, Icon *i*, int *align*)

        align -> either LEFT, RIGHT, CENTER, LEADING orTRAILING

- Methods

    Icon getIcon( )

    String getText( )

    void setIcon(Icon *i*)

    void setText(String *s*)

```
import java.awt.*;
import javax.swing.*;
/*
<applet code = “JLabelDemo” width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet
{
        public void init( ) {
                Container cp = getContentPane( );
                ImageIcon ii = new ImageIcon(“France.gif”);
                JLabel jl = new JLabel(“France”, ii, JLabel.CENTER);
                cp.add(jl);
        }
}
```

Text Fields

- **JTextComponent** class
- extends JComponent
  - one of the subclass of JComponent -> **JTextField**
- constructors
  - JTextField( )
  - JTextField(int *cols*) -> no.of columns in the text field
  - JTextField(String *s*, int *cols*)
  - JTextField(String *s*)
- Eg. JTextField jtf = new JTextField(15);

  contentPane.add(jtf);

Buttons

- Swing Buttons are subclasses of the **AbstractButton** class.
- AbstractButton class
  - extends **JComponent**.
  - contains methods to control the behaviour of buttons, checkboxes and radio buttons.
  - the text associated with a button can be read and written via

    String getText( )

    void setText(String *s*)

The JButton class

- Provides the functionality of a push button.
- Allows an icon, a string or both to be associated with the push button.
- Constructors
    - JButton(Icon *i*)
    - JButton(String *s*)
    - JButton(String *s*, Icon *i*)

Check Boxes

- **JCheckBox** class
  - concrete implementation of AbstractButton
  - immediate superclass is JToggleButton
    - provides support for two-state buttons
- Constructors
  - JCheckBox(Icon *i*)
  - JCheckBox(Icon *i*, boolean *state*)
  - JCheckBox(String *s*)
  - JCheckBox(String *s*, boolean *state*)
  - JCheckBox(String *s*, Icon *i*)
  - JCheckBox(String *s*, Icon *i*, boolean *state*)

- The state of the check box can be changed via the following method

  void setSelected(boolean *state*)

- When a checkbox is selected or deselected, an item event is generated.
  - This event is handled by itemStateChanged( ) method.
  - Inside itemStateChanged( ), the getItem( ) method gets the JCheckBox object that generated the event.
  - Next a call to getStateChange( ) determines if the box was selected or cleared.

    selected -> SELECTED is returned

    deselected -> DESELECTED is returned

Radio Buttons

- **JRadioButton** class
- Constructors
  - JRadioButton(Icon *i*)
  - JRadioButton(Icon *i*, boolean *state*)
  - JRadioButton(String *s*)
  - JRadioButton(String *s*, boolean *state*)
  - JRadioButton(String *s*, Icon *i*)
  - JRadioButton(String *s*, Icon *i*, boolean *state*)
- Radio Buttons must be configured into a group.
- Only one of the buttons in that group can be selected at any time.

- The Button Group is instantiated to create a button group.
- Elements are added to the button group via the following method.

  void add(AbstractButton *ab*)

  ab -> reference to the button to be added to the group.
- Radio button presses generate action events that are handled by actionPerformed( )
- The getActionCommand( ) method gets the text that is associated with a radio button and uses it to set the text field.