CS410J: *Programming with Java*

Computer users have come to expect a graphical interface to the programs that they use. However, many tools for constructing graphical user interfaces (GUIs) have been tied to a certain operating system or windowing environment. Since its initial release, the Java™ Programming Platform has provided the ability to construct graphical user interfaces in a platform-independent manner.

Developing Graphical User Interfaces using Java

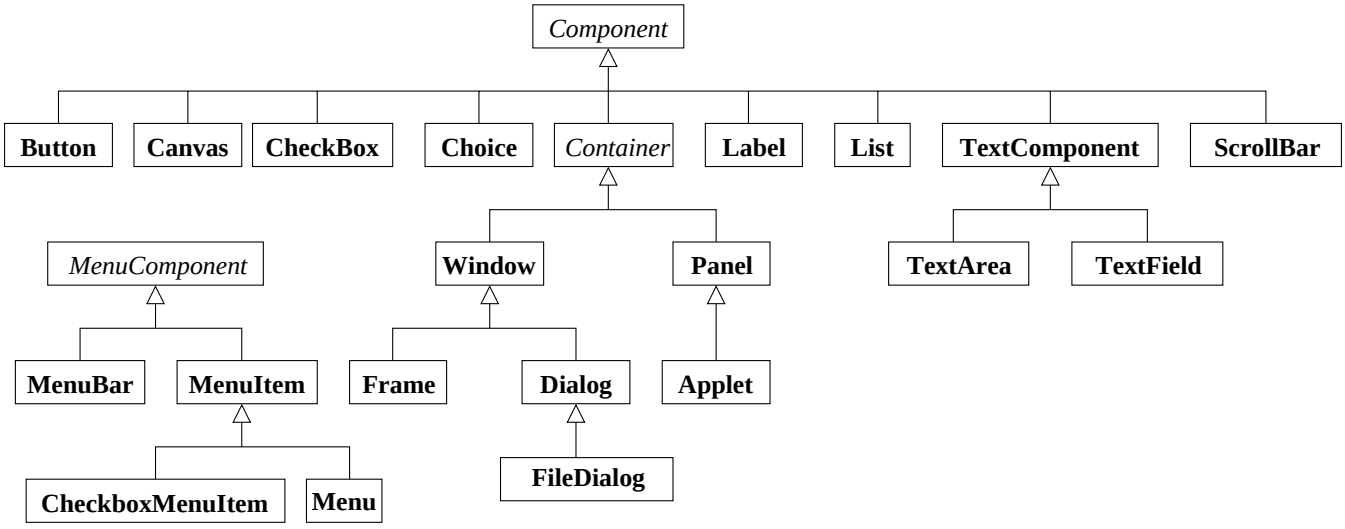
- The Abstract Windowing Toolkit
- Widget layout
- Event handling
- The Java Foundation Classes/Swing

GUI Programming with Java

From the beginning, Java has supported classes for creating graphics and graphical user interfaces

- JDK 1.0.2: The Abstract Windowing Toolkit (AWT)
  - Heavyweight GUI components based on platform-specific “peers”
  - Applets: Code that is downloaded over the internet and run by the client
- JDK 1.1: New event model (Listeners)
- JDK 1.2: The Java Foundation Classes (JFC)
  - Lightweight “swing” components
  - Pluggable look and feel (plaf)
  - Extended line graphics capabilities (Java2D)
  - Printing to printers
  - Data Transfer (cut and paste, drag and drop, MIME, etc.)

The AWT (`java.awt`) GUI Components



All GUI classes are a subclass of `java.awt.Component`

- GUI widgets: `Button`, `CheckBox`, `Label`, `List`
- `TextComponents` for entering text: `TextArea`. `TextField`
- `Containers` that hold other widgets: `Panel`, `Window`, `java.applet.Applet`, `Frame`, `Dialog`

Note that AWT components are mapped to native platform widgets

- On Windows, an AWT `Button` is really a MS Windows button

Creating a GUI

Creating a GUI involves several steps

- Create `Components` and add them to `Containers`
  - Create a `Button`, add it to a `Panel`
  - Add the `Panel` to a `Frame`
  - Often you will subclass some `Container`
- Establish event handlers to handle events
  - Beep when the button is clicked
  - An event handler thread monitors native windows events (e.g. clicking the mouse) and dispatches them to the appropriate widgets*
- Display the GUI
  - Display the `Frame`
  - Load a web page containing an `Applet`

*So, after displaying your GUI, your `main` method can return

java.awt.Button

Buttons are created with an optional String label

```
package edu.pdx.cs410J.gui;

import java.awt.*;

public class ButtonExample extends Panel {
  public ButtonExample() {
    Button ok = new Button("OK");
    Button cancel = new Button("Cancel");
    this.add(ok);
    this.add(cancel);
  }

  public static void main(String[] args) {
    Frame frame = new Frame("Button example");
    frame.add(new ButtonExample());
    frame.pack();
    frame.setVisible(true);
  }
}
```

A Frame is a top-level Window with a title bar

java.awt.Checkbox

A `Checkbox` is a label with a toggle button

```
package edu.pdx.cs410J.gui;

import java.awt.*;

public class CheckboxExample extends Panel {

  public CheckboxExample() {
    Checkbox cb1 = new Checkbox("No");
    Checkbox cb2 = new Checkbox("Yes", true);
    Checkbox cb3 = new Checkbox("Maybe");
    cb3.setState(true);
    this.add(cb1);
    this.add(cb2);
    this.add(cb3);
  }

  public static void main(String[] args) {
    Frame frame = new Frame("Checkbox example");
    frame.add(new CheckboxExample());
    frame.pack();
    frame.setVisible(true);
  }
}
```

java.awt.CheckboxGroup

A CheckboxGroup is a group of Checkboxes in which only one box can be checked at a time ("radio buttons")

```
package edu.pdx.cs410J.gui;

import java.awt.*;

public class CheckboxGroupExample extends Panel {

  public CheckboxGroupExample() {
    CheckboxGroup group = new CheckboxGroup();
    Checkbox cb1 =
      new Checkbox("UNIX", true, group);
    Checkbox cb2 =
      new Checkbox("Windows", false, group);
    Checkbox cb3 = new Checkbox("Macintosh");
    cb3.setState(false);
    cb3.setCheckboxGroup(group);
    this.add(cb1);
    this.add(cb2);
    this.add(cb3);
  }
}
```

CheckboxGroup's getSelectedCheckBox returns the Checkbox that is checked

java.awt.Choice

A `Choice` is a drop-down list of strings

```
package edu.pdx.cs410J.gui;

import java.awt.*;

public class ChoiceExample extends Panel {

  public ChoiceExample() {
    Choice choice = new Choice();
    choice.add("Monday");
    choice.add("Tuesday");
    choice.add("Wednesday");
    choice.add("Thursday");
    choice.add("Friday");
    choice.add("Saturday");
    choice.insert("Sunday", 0);
    this.add(choice);
  }
}
```

The `getSelectedItem` and `getSelectedIndex` return the selected string or its index

Initially, the first item in the `Choice` is selected

java.awt.Label

A `Label` is an immutable single line of text that usually describes what some other widget does

```
package edu.pdx.cs410J.gui;

import java.awt.*;

public class LabelExample extends Panel {

  public LabelExample() {
    Label l1 = new Label("A label");
    Label l2 =
      new Label("Left justified", Label.LEFT);
    Label l3 = new Label("Right", Label.RIGHT);
    Label l4 = new Label("Centered");
    l4.setAlignment(Label.CENTER);
    this.add(l1);
    this.add(l2);
    this.add(l3);
    this.add(l4);
  }
}
```

The text in the label maybe be justified: `Label.CENTER`, `Label.RIGHT`, `Label.LEFT`

java.awt.List

A `List`* is a scrolling list box that may allow multiple selections

```
package edu.pdx.cs410J.gui;

import java.awt.*;

public class ListExample extends Panel {

  public ListExample() {
    List list = new List(4);  // 4 items visible
    list.add("Monday");
    list.add("Tuesday");
    list.add("Wednesday");
    list.add("Thursday");
    list.add("Friday");
    list.add("Saturday");
    list.add("Sunday", 0);
    list.setMultipleMode(true);
    this.add(list);
  }
}
```

The `getSelectedItem` and `getSelectedItems` return the selected item(s) as an array of `String`s

*Not be to confused with `java.util.List`

java.awt.TextField

A `TextField` is a one-line, fixed-width widget for entering text

```
package edu.pdx.cs410J.gui;

import java.awt.*;

public class TextFieldExample extends Panel {

  public TextFieldExample() {
    TextField field =
      new TextField(20);  // 20 characters
    field.setText("Initial contents");
    this.add(field);
  }
}
```

The `setEditable` method is used to make the text immutable

The `setEchoChar` adjusts the character that is echoed (e.g. `*` for entering passwords)

The contents of a `TextField` are set with the `setText` method

java.awt.TextArea

A `TextArea` is a multi-line text field with optional scrollbars

```
package edu.pdx.cs410J.gui;

import java.awt.*;

public class TextAreaExample extends Panel {

  public TextAreaExample() {
    TextArea area =
      new TextArea("Initial Text", 5, 10,
                   TextArea.SCROLLBARS_BOTH);
    area.append("Appended text");
    area.replaceRange("Words", 8, 12);
    this.add(area);
  }
}
```

The scrollbars may be specified as
`TextArea.SCROLLBARS_BOTH`,
`TextArea.SCROLLBARS_HORIZONTAL_ONLY`,
`TextArea.SCROLLBARS_NONE`,
`TextArea,SCROLLBARS_VERTICAL_ONLY`

Some methods of `java.awt.Component`

All components have the following methods

- `getHeight` and `getWidth` that return the component's height and width in pixels
- `getLocation` that returns the location of the component relative to its container as a `java.awt.Point`
- `getLocationOnScreen` that returns the location of the component relative to the upper-left corner of the screen
- `getBounds` that returns the bounding `java.awt.Rectangle` of the component
- `setEnabled` that enables or disables (e.g. greying-out a button) the component
- `setVisible` that shows the component (most components are visible by default)
- `setBackground` that changes the `java.awt.Color` of the component
- `setFont` that changes the `java.awt.Font` used to draw text in the component

Event handling

Interaction between the user and a GUI generates events

- The mouse is moved, the mouse button is clicked, keys are pressed

`java.util.EventObject` models an event that came from a “source”

`java.awt.AWTEvent` is the superclass of all GUI events

- An `ActionEvent` is created when a `Button` is clicked
- Most event-related classes are the `java.awt.event` package

AWT events are handled by a “listener” object that is registered on a widget

- An `ActionListener`’s `actionPerformed` method handles an `ActionEvent`
- An `ActionListener` is registered by invoking a `Component`’s `addActionListener` method

Handling ActionEvents

```
package edu.pdx.cs410J.gui;

import java.awt.*;
import java.awt.event.*;

public class ActionEventExample extends Panel
  implements ActionListener {

  private Button button;
  private Label label;

  public ActionEventExample() {
    this.button = new Button("Click me");
    this.button.addActionListener(this);
    this.label = new Label("Not clicked");
    this.add(this.button);
    this.add(this.label);
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == this.button) {
      this.label.setText("Clicked");
    }
  }
}
```

The button's `ActionEvent` is delegated to its `ActionListener`

Slight Detour Through Inner Classes

Some classes have a very limited use

- Only used by one other class
- Only instantiated once

Classes can be declared inside another class

- Called “inner classes”

For instance, `java.util.Map` declares an inner interface named `Entry`.

- It is referred to as `java.util.Map.Entry`
- Objects that implement `Map.Entry` are returned by the `entrySet` method of `Map`
- The name of an inner class’s class file is encoded with a `$`: `java/util/Map$Entry.class`

Example of an inner class

From Map.java:

```
package java.util;
public interface Map {
  int size();
  boolean isEmpty();
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  Object get(Object key);
  Object put(Object key, Object value);
  Object remove(Object key);
  void putAll(Map t);
  void clear();
  public Set keySet();
  public Collection values();
  public Set entrySet();

  public interface Entry {
    Object getKey();
    Object getValue();
    Object setValue(Object value);
    boolean equals(Object o);
    int hashCode();
  }

  boolean equals(Object o);
  int hashCode();
}
```

Using an inner class

Printing out the contents of a `Map`

```
public void printMap(Map map, PrintWriter pw) {
  Iterator entries = map.entrySet().iterator();
  while (entries.hasNext()) {
    Map.Entry entry = (Map.Entry) entries.next();
    Object key = entry.getKey();
    Object value = entry.getValue();
    pw.println(key + " -> " + value);
  }
}
```

Some more notes about inner classes:

- Inner classes may not have non-`final` `static` members
- Inner classes may be marked as `static`
  - Static inner classes cannot reference the outer class's instance members
  - Static methods of an outer class can only create instances of static inner classes

Anonymous inner classes

Sometimes you only need a class once. Why bother writing it at all?

Class declarations can be “inlined” into methods

“Anonymous Inner Class”

An anonymous inner class may only reference `final` variables of the method in which it is declared

- May need to qualify variable with the name of the outer class (`Outer.this`)

The names of the class file for anonymous inner classes are encoded with numbers:

```
edu/pdx/cs410J/gui/FindClassFiles$1.class
```

is the first anonymous inner class declared in `FindClassFiles`

Anonymous classes are especially handy when writing event handlers in GUI programming

- Create a specific piece of code that is only used in one place

Using Anonymous Inner Classes

```
package edu.pdx.cs410J.gui;
import java.io.*;

public class FindClassFiles {
  private static void findFiles(File dir) {
    final String suffix = ".class";

    File[] classFiles =
      dir.listFiles(new FilenameFilter() {
        public boolean accept(File dir, String name)
          return name.endsWith(suffix);
        }
      });

    for (int i = 0; i < classFiles.length; i++)
      System.out.println(classFiles[i]);

    File[] subdirs =
      dir.listFiles(new FileFilter() {
        public boolean accept(File file) {
          return file.isDirectory();
        }
      });

    for (int i = 0; i < subdirs.length; i++)
      findFiles(subdirs[i]);
  }
}
```

Using inner classes for listeners

In the action event handlind example, the example class implemented the `actionPerformed` method

- As a consequence, `button` and `label` had to be stored in fields

- This is messy, `button` and `label` should only be local variables in the constructor

Anonymous inner classes allow a class declaration to be “inlined” into a method

- Anonymous inner classes are classes that are only instantiated once

- Ideal for event handlers

Event handling with an inner class

```
package edu.pdx.cs410J.gui;

import java.awt.*;
import java.awt.event.*;

public class ActionEventExample2 extends Panel {

  public ActionEventExample2() {
    final Label label = new Label("Not clicked");
    Button button = new Button("Click me");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          label.setText("Clicked");
        }
      });

    this.add(button);
    this.add(label);
  }
}
```

I prefer using anonymous inner classes for event handling

- Puts the event handling code next to the code for creating the widget
- Fewer fields, more `final` variables

Kinds of `AWTEvent`s

There are a handful of events in `java.awt.event`

- `ComponentEvent`: a `Component` moved, resized, or shown
- `ContainerEvent`: a `Component` was added to or removed from a `Container`
- `FocusEvent`: a `Component` gained or lost keyboard focus
- `KeyEvent`: a keyboard key was pressed/released
- `MouseEvent`: the mouse was clicked, dragged, etc.
- `WindowEvent`: a window was opened, closed, iconified, etc.
- `ActionEvent`: a component-defined action occurred (e.g. a button was clicked)
- `AdjustmentEvent`: an integer value changed
- `ItemEvent`: an item was selected or deselected
- `TextEvent`: text was changed

Listeners

ActionListener: Button, List, MenuItem, TextField

- actionPerformed(ActionEvent e)

AdjustmentListener: Scrollbar

- adjustmentValueChanged(AdjustmentEvent e)

ComponentListener: Component

- componentHidden(ComponentEvent e)
- componentMoved(ComponentEvent e)
- componentResized(ComponentEvent e)
- componentShown(ComponentEvent e)

ContainerListener: Container

- componentAdded(ContainerEvent e)
- componentRemoved(ContainerEvent e)

Listeners

`FocusListener`: `Component`

- `focusGained(FocusEvent e)`
- `focusLost(FocusLost e)`

`KeyListener`: `Component`

- `keyPressed(KeyEvent e)`
- `keyReleased(KeyEvent e)`
- `keyTyped(KeyEvent e)`

`WindowListener`: `Window`

- `windowActivated(WindowEvent e)`
- `windowClosed(WindowEvent e)`
- `windowClosing(WindowEvent e)`
- `windowDeactivated(WindowEvent e)`
- `windowDeiconified(WindowEvent e)`
- `windowIconified(WindowEvent e)`
- `windowOpened(WindowEvent e)`

Listeners

MouseListener: Component

- mouseClicked(MouseEvent e)
- mouseEntered(MouseEvent e)
- mouseExited(MouseEvent e)
- mousePressed(MouseEvent e)
- mouseReleased(MouseEvent e)

MouseMotionListener: Component

- mouseDragged(MouseEvent e)
- mouseMoved(MouseEvent e)

TextListener: TextArea, TextField

- textValueChanged(TextEvent e)

ItemListener: Choice, Checkbox, CheckboxMenuItem, List

- itemStateChanged(ItemEvent e)

Adapters

Some listeners (e.g. `MouseListener`) have multiple methods

- Have to implement every method, even if you only want one

Adapter classes provide empty implementations of a listener's methods

- Only override the methods that you need

- `ComponentAdapter`, `ContainerAdapter`, `FocusAdapter`, `KeyAdapter`, `MouseAdapter`, `MouseMotionAdapter`, `WindowAdapter`

Using a `WindowAdapter`

A `WindowAdapter` can be used to gracefully end GUI applications

```
package edu.pdx.cs410J.gui;

import java.awt.*;
import java.awt.event.*;

public class WindowAdapterExample extends Panel {

  public WindowAdapterExample() {
    Label label = new Label("I don't do much");
    this.add(label);
  }

  public static void main(String[] args) {
    Frame frame = new Frame("WindowAdapter example");
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          // The frame is being closed, exit
          System.exit(0);
        }
      });
    frame.add(new WindowAdapterExample());
    frame.pack();
    frame.setVisible(true);
  }
}
```

Handling mouse and key events

```
package edu.pdx.cs410J.gui;

import java.awt.*;
import java.awt.event.*;

public class MouseAndKeyEvents extends Panel {
  public MouseAndKeyEvents() {
    final TextField x = new TextField(3);
    final TextField y = new TextField(3);
    final TextField c = new TextField(1);
    this.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
          x.setText(e.getX() + "");
          y.setText(e.getY() + "");
        }
      });
    Button button = new Button("Type Here");
    button.addKeyListener(new KeyAdapter() {
        public void keyTyped(KeyEvent e) {
          c.setText(e.getKeyChar() + "");
        }
      });
    x.setEditable(false); this.add(x);
    y.setEditable(false); this.add(y);
    c.setEditable(false); this.add(c);
    this.add(button);
  }
}
```

Dialog boxes

A dialog box is a window with a title and a border that is usually “popped-up” by an application

- A `Dialog` has a owning `Dialog` or `Frame`
- If a `Dialog` is “modal”, then no other window in the application is accessible while the dialog is visible

```
class DialogBox extends Dialog {
  DialogBox(Frame owner) {
    super(owner, "Example dialog box", true /*modal*/

    Button button = new Button("Go away");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          DialogBox.this.dispose();
        }
      });
    this.add(button);

    this.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          DialogBox.this.dispose();
        }
      });
    this.pack();
  }
}
```

Displaying a Dialog Box

```
package edu.pdx.cs410J.gui;

import java.awt.*;
import java.awt.event.*;

public class DialogExample extends Frame {

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

    Button button = new Button("Show dialog");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          (new DialogBox(DialogExample.this)).show();
        }
      });

    Panel panel = new Panel();
    panel.add(button);
    this.add(panel);
  }
}
```

Note that `DialogExample` extends `Frame` and that we created a `Panel` to hold the `Button`

Menus

Menus are implemented in the AWT with the `Menu`, `MenuBar`, `MenuItem`, and `CheckboxMenuItem`

- Specified with `Frame`'s `setMenuBar` method
- When a menu item is selected, an `ActionEvent` is issued

A `Menu` is created with a given name

`MenuItem`s are then added to the `Menu`

- Note that `Menu` itself is a `MenuItem`, so menus can be nested
- `Menu`'s `addSeparator` method generates a separating line in the menu
- A menu item can be `Disabled` (i.e. greyed-out)

`Menu`s are added to a `MenuBar`

Using Menus

```
package edu.pdx.cs410J.gui;

import java.awt.*;
import java.awt.event.*;

public class MenuExample extends Frame {

  public MenuExample(String title) {
    super(title);
    final Label label = new Label("Your text here");
    Menu menu = new Menu("Colors");
    MenuItem item = new MenuItem("Blue");
    menu.add(item);
    item.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          label.setBackground(Color.blue);
        }
      });
    // Similarly for "Red" and "Yellow"

    MenuBar menuBar = new MenuBar();
    menuBar.add(menu);
    this.setMenuBar(menuBar);
    Panel panel = new Panel();
    panel.add(label);
    this.add(panel);
  }
}
```

java.awt.CheckboxMenuItem

A `CheckboxMenuItem` can be toggled on and off

```
public class CheckboxMenuItemExample extends Frame {
  public CheckboxMenuItemExample(String title) {
    super(title);
    final Label label = new Label("Your text here");
    // Make "Colors" menu like before...

    Menu styleMenu = new Menu("Style");

    CheckboxMenuItem cbitem =
      new CheckboxMenuItem("Bold", false);
    styleMenu.add(cbitem);
    cbitem.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent e) {
          int style = label.getFont().getStyle();
          if (e.getStateChange() ==
             ItemEvent.SELECTED) {
            style |= Font.BOLD;
          } else if (e.getStateChange() ==
                    ItemEvent.DESELECTED) {
            style &= ~Font.BOLD;
          }
          label.setFont(label.getFont().
                        deriveFont(style));
        }
      });
```

Keyboard Shortcuts

A `MenuShortcut` can be used to associate the selection of a menu item with a key sequence

- The shortcut is activated with the control key on UNIX and Windows and the command key on a Macintosh
- A flag to the `MenuShortcut`'s constructor specifies whether or not the shift key is part of the shortcut
- The key is specified using one of the "virtual key" fields of `KeyEvent`
- A `MenuShortcut` is specified with `MenuItem`'s `setShortcut` method

```
item.setShortcut(new MenuShortcut(KeyEvent.VK_R));
```

See `edu.pdx.cs410J.gui.MenuShortcutExample`

Popup menus

Regular menus must be anchored to a `Frame`

A `PopupMenu` needs no anchor

- Usually displayed in response to a mouse click
- Even though it may not be displayed right away, a `PopupMenu` must be added to a `Container`

From `edu.pdx.cs410J.gui.PopupMenuExample`:

```
final Label label = new Label("Your text here");
label.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
      if (!e.isPopupTrigger())
        return;
      PopupMenu menu = new PopupMenu("Change text");
      MenuItem item = new MenuItem("One");
      menu.add(item);
      item.addActionListener(...);
      // Similarly for "Two" and "Three"

      PopupMenuExample.this.add(menu);
      menu.show(e.getComponent(),
                e.getX(), e.getY());
    }
  });
```

Arranging Components in a Container

When constructing GUIs, the programmer needs to specify where components should be placed, and how they should be resized

A layout manager encapsulates an algorithm for positioning and sizing widgets

- Factors the code for handling resizing events into one place
- You don’t have to write this code!

`java.awt.LayoutManager` contains methods for

- Getting size and position information for the layout manager and the widgets it lays out
- Adjusting the layout when a widget is added or removed from a container

The Size of a `Component`

`Component` has three methods that returns it size as a `java.awt.Dimension`

- `getPreferredSize`: desired size
- `getMinimumSize`: smallest desired size
- `getMaximumSize`: largest desired size

Layout managers use these sizes when layout out components

If you want to specify these sizes for AWT widgets, you have to subclass the widget and override the method

Swing components have methods like `setPreferedSize`

Layout Managers and `Container`s

A `Container` uses a `LayoutManager` to position and size the `Component`s it contains

- The layout manager computes the size and position of each component and invokes the component's `setBounds`, `setLocation`, and `setSize` methods before the container is redrawn

`LayoutManager` is associated with a `Container` via its `setLayout` method

```
Panel p = new Panel();
p.setLayout(new BorderLayout());
```

Adding Components to a Container

A container has several `add` methods for adding components

- `addComponent(Component comp)`
- `addComponent(Component comp, int index)`

  Specifies an ordering of components in the container

- `addComponent(Component comp, Object constraints)`

  A constraint could be “at the top of the container”

- `addComponent(Component comp, Object constraints, int index)`

These methods, in turn, invoke the `add` method of the container’s layout manager

java.awt.FlowLayout

`FlowLayout` has rather simple rules:

- Respect the preferred size of all components in the container
- Layout components left to right, start a new row when the right edge of the container is reached
- Will only display as many components as can fit in the container

```
package edu.pdx.cs410J.gui;
import java.awt.*;

public class FlowLayoutExample  {
  public static void main(String[] args) {
    Frame f = new Frame("FlowLayoutExample");
    f.setLayout(new FlowLayout());
    f.add(new Label("One"));
    f.add(new Label("Two"));
    f.add(new Label("Three"));
    f.add(new Label("Four"));
    f.add(new Label("Five"));
    f.add(new Label("Six"));
    f.pack();
    f.setVisible(true);
  }
}
```

java.awt.FlowLayout

The alignment of a `FlowLayout` can be set of `FlowLayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`

Additionally, you can set the horizontal and vertical gap between components (`hgap` and `vgap`)

Preferred height of the container is maximum height of any component plus `vgap`

Preferred width of the container is sum of width of components plus `hgap` on either side of components

See `edu.pdx.cs410J.gui.NestedFlowLayouts`

When using a `FlowLayout` keep in mind:

- Don't nest a `FlowLayout`-managed container inside a component whose layout manager respects preferred height
- `FlowLayout` hides components that do not fit in the container
- `FlowLayout` is good when you only have a small number of components

java.awt.BorderLayout

BorderLayout displays components in the north, south, east, west, and center of a container (also has hgap and vgap)

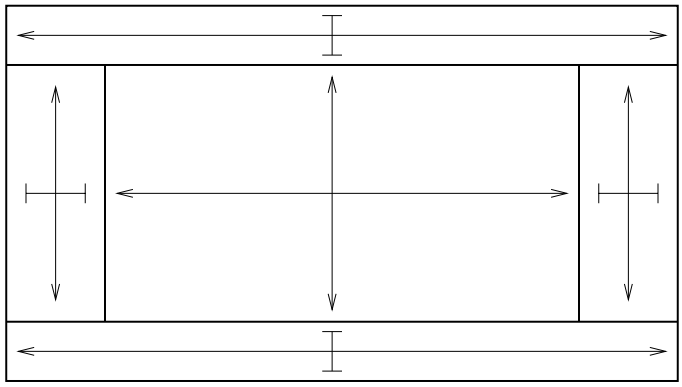
```
package edu.pdx.cs410J.gui;
import java.awt.*;

public class BorderLayoutExample  {

  public static void main(String[] args) {
    Frame f = new Frame("BorderLayout Example");
    f.setLayout(new BorderLayout());
    f.add(new Label("North"), BorderLayout.NORTH);
    f.add(new Label("South"), BorderLayout.SOUTH);
    f.add(new Button("East"), BorderLayout.EAST);
    f.add(new Button("West"), BorderLayout.WEST);
    f.add(new Label("Center"), BorderLayout.CENTER);

    f.pack();
    f.setVisible(true);
  }
}
```

Probably, the most useful layout manager!

java.awt.BorderLayout

BorderLayout follows these rules:

- NORTH and SOUTH components: Respects their preferred height, sets the width to width of container
- EAST and WEST components: Respects their preferred width, sets height to remaining height of the container
- CENTER component: Sets height and width to fill remaining space



Note how a FlowLayout interacts with a BorderLayout

- FlowLayoutInSouth: FlowLayout never has more than one row
- FlowLayoutInEast: Still only one row (BorderLayout respects preferred width)
- FlowLayoutInCenter: All components are displayed

A `BorderLayout` Example

```
package edu.pdx.cs410J.gui;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class PostfixCalculator {
  public static void main(String[] args) {
    Panel p = new Panel();
    p.setLayout(new BorderLayout());
    final Button b = new Button("Calculate");
    p.add(b, BorderLayout.WEST);
    final TextField expr = new TextField(15);
    p.add(expr, BorderLayout.CENTER);

    // Event hanling code...

    Frame f = new Frame("Postfix Calculator");
    f.setLayout(new BorderLayout());
    f.add(p, BorderLayout.CENTER);

    f.pack();
    f.setVisible(true);
  }
}
```

Why doesn't the window resize nicely? What can we do to fix it?

The Preferred Size of with `BorderLayout`

`BorderLayout` tries to respect the preferred size of each of its components

- Preferred width is maximum of:
  - Preferred width of NORTH component
  - Preferred width of SOUTH component
  - Sum of the preferred widths of EAST, CENTER, and WEST components plus `hgap`s
- Preferred height is sum of `vgap`s, the preferred height of NORTH component, SOUTH component, and the maximum of
  - Preferred height of EAST component
  - Preferred height of CENTER component
  - Preferred height of WEST component

java.awt.GridLayout

`GridLayout` arranges components in a grid, each component has the same width and height

```
package edu.pdx.cs410J.gui;
import java.awt.*;

public class GridLayoutExample  {
  public static void main(String[] args) {
    Panel p = new Panel();
    int rows = 4;
    int columns = 3;
    p.setLayout(new GridLayout(rows, columns));

    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < columns; j++) {
        p.add(new Label("(" + i + ", " + j + ")"));
      }
    }
  }
}
```

The above example creates a grid with 12 elements. You can create a four-column grid with an unlimited number of rows with `new GridLayout(0, 4)`

java.awt.GridLayout

Like the other layout managers, `GridLayout` has a `hgap` and `vgap`

- Non-zero `hgap` and `vgap` tend to be used more with `GridLayout`, otherwise components are adjacent

The preferred width is the maximum preferred width of **all** components times the number of columns, plus the `hgaps`

The preferred height is the maximum preferred height of `all` components times the number of rows, plus the `vgaps`

See `edu.pdx.cs410J.gui.AdjustGridLayoutGaps`

Note that a `FlowLayout` puts gaps **around** components while `GridLayout` puts gaps **between** components

`java.awt.CardLayout`

Unlike the other layout managers, `CardLayout` only displays one component at a time

- The components in the container are indexed
- Components can also have a `String` name associated with them
- Lets you flip through the components like a deck of cards
- `CardLayout` has methods like `previous`, `next`, and `show`. These methods need a reference to the container.
- The preferred width of the container is the maximum of the preferred widths of the components
- The preferred height of the container is the maximum of the preferred heights of the components

Using CardLayout

```
package edu.pdx.cs410J.gui;
import java.awt.*;

public class DavesSocialLife {
  public static void main(String[] args) {
    final Panel calendar = new Panel();
    final CardLayout layout = new CardLayout();
    calendar.setLayout(layout);
    final List list =
      new List(daysOfTheWeek.length, false);
    for (int i = 0; i < daysOfTheWeek.length; i++) {
      String day = daysOfTheWeek[i];
      list.add(day);

      Panel p = new Panel();
      p.setLayout(new BorderLayout());
      p.add(new Label(day, Label.CENTER),
                        BorderLayout.NORTH);
      calendar.add(p, day);
    }
    list.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent e) {
          String day = list.getSelectedItem();
          layout.show(calendar, day);
        }
      });   // Set up rest of GUI...
  }
}
```

`java.awt.GridBagLayout`

`GridBagLayout` (a.k.a. “barf bag layout”) is quite difficult to use

- Divide the GUI into a grid of cells
- Assign relative sizes and padding to the cells
- Tools that generate GUIs use `GridBagLayout`, everyone else should run away

Applets

Applets are small applications that subclass `java.applet.Applet` that are usually downloaded over the internet and are run on the client side

An Applet doesn't have a `main` method; its more like having a GUI component that you "start up"

Applets are embedded in an HTML file and obtain parametric information from `PARAM` tags in the HTML file

Applets have a number of security restrictions on them

- Can't write to local file system
- Can only establish socket to machine it was downloaded from

Embedding an Applet in an HTML page

Applets are embedded in HTML using the `<APPLET>` tag

```
<APPLET
  CODEBASE="Directory containing class files"
  ARCHIVE="Jar files"
  CODE="Fully-qualified name of Applet class"
  ALT="Alternate text if browser can't run Java"
  NAME="An identifying name of the applet"
  WIDTH="Width of applet on HTML page (pixels)"
  HEIGHT="Height of applet (pixels)"
  ALIGN="Alignment of applet (like IMG tag)"
>
<PARAM NAME="name" VALUE="value">
<PARAM NAME="name" VALUE="value">

HTML to be displayed if applet can't be run
</APPLET>
```

The `CODEBASE`, `CODE`, `WIDTH`, and `HEIGHT` attributes are required, others are optional.

Note that with applets, `System.out` and `System.err` are displayed on a "Java console"

An example APPLET tag

```
<html>
  <head>
    <title>Family Tree Applet</title>
  </head>

  <body>
    <h1>Family Tree Applet</h1>

    <P>Here is a Java Applet that uses components
    of the Family Tree GUI.  Note that the XML
    files are loaded from URLs.</P>

<APPLET CODE="edu.--.familyTree.FamilyTreeApplet"
 CODEBASE="http://www.cs.pdx.edu/~whitlock/classes"
 ARCHIVE="http://www.cs.pdx.edu/~whitlock/jars/jaxp.j
 WIDTH=500 HEIGHT=300>

<PARAM NAME = "xmlFile"
  VALUE ="http://www.cs.pdx.edu/~whitlock/.../davesFa
</APPLET>

</html>
```

I’ve found that getting the CODEBASE/ARCHIVE tags to work correctly is a little tricky.

Viewing Applets

The JDK includes the `appletviewer` tool for use during development

Working with applets was traditionally problematic because Java support in web browsers varied greatly

- To take the burden of adding a JVM to a web browser, Sun developed the “Java Plugin”
- Treats a Java program just like any other browser plugin
- The conversion utility post-processes the `APPLET` tag and adds plugin-related HTML understood by Netscape and IE

More recently, Java Web Start has emerged as the method of choice for deploying client-side Java application

Swing

The AWT is nice, but...

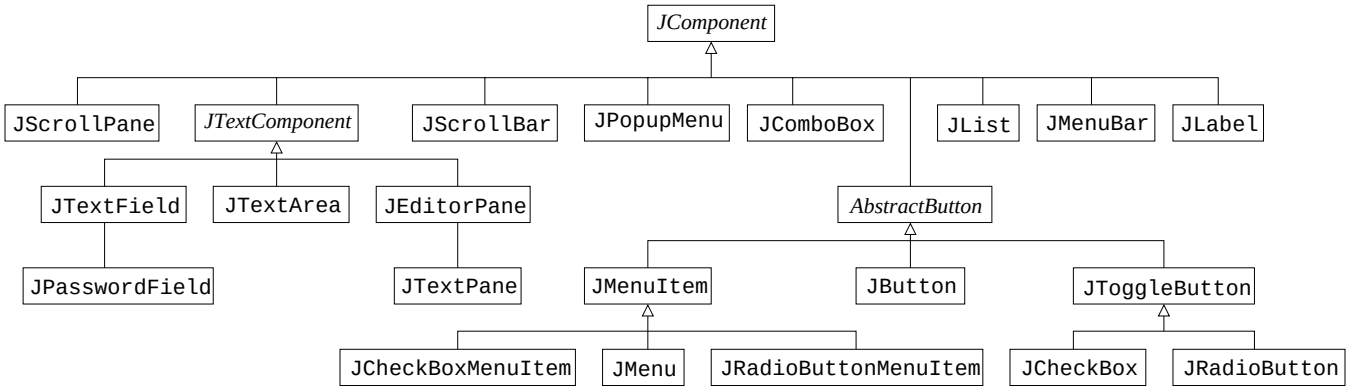
- By no means “fully-featured”
- Strongly tied to the native platform, difficult to port
- Just doesn’t look that good

In 1997, Sun, IBM, and Netscape began to develop the Java Foundation Classes (JFC) also known as “Swing”

- New GUI widgets
- Pluggable Look at Feel (Motif widgets on a Windows machine)
- Accessibility API for non-traditional interfaces
- Expanded 2D Graphics API (Java2D)
- Drag and Drop

Swing widgets are implemented entirely in Java, no native code!

Swing Components



Most of the swing components reside in the `javax.swing` package

Similar to AWT components, but their names start with “J”

- Swing buttons and lables can display HTML in addition to text
- Borders can be added easily to any component
- Swing components use a “model” to maintain their state
  - A `JSlider` uses a `BoundedRangeModel` to enforce its boundaries and hold its current value
- Do not mix Swing and AWT components!

javax.swing.JButton

A JButton can do everything a Button can do

- A key mnemonic (“shortcut”) can be set with setMnemonic
- An Icon can be associated with a JButton
  - setDisabledIcon, setDisabledSelectedIcon, setPressedIcon, setRolloverIcon, setRolloverSelectedIcon, setSelectedIcon
- setVerticalAlignment and setHorizontalAlignment anchors text inside button
- Uses an ActionListener to handle events
- Can use tab and space bar to navigate widgets

JButton example

```
package edu.pdx.cs410J.gui;

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

public class JButtonExample extends JPanel {
  public JButtonExample() {
    JButton b1 = new JButton("Press me!");
    b1.setMnemonic(KeyEvent.VK_P);
    b1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          System.out.println("I was pressed!");
        }
      });
    this.add(b1);

    JButton b2 =
      new JButton("<html>No, press <i>me</i>!</html>"
    b2.setMnemonic(KeyEvent.VK_N);
    this.add(b2);

    JToggleButton b3 =
      new JToggleButton("Toggle Button");
    this.add(b3);
  }
}
```

Manipulating Text with Swing

`javax.swing.JTextComponent` is a base class of widgets that view and edit text

- Supports `cut`, `copy`, `paste`, `selectAll`, etc.
- Can `getText` or `getSelectedText`

`JTextField` displays one line of text

`JTextPassword` is used for entering passwords

- Use `getPassword` instead of `getText`
- Password is returned as an array of `char`
- Text in a `JTextPassword` cannot be copied nor cut
- The echo character cannot be unset like an AWT `TextField`

`JTextArea` displays multiple lines of unformatted text

Example of Swing Text components

```
package edu.pdx.cs410J.gui;

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

public class JTextExample extends JPanel {
  public JTextExample() {
    this.setLayout(new BorderLayout());

    JPanel p = new JPanel();
    p.add(new JLabel("User name:"));
    final JTextField name = new JTextField(10);
    p.add(name);

    p.add(new JLabel("Password:"));
    final JPasswordField password =
      new JPasswordField(10);
    p.add(password);
    this.add(p, BorderLayout.NORTH);

    final JTextArea text = new JTextArea(10, 30);
    this.add(text, BorderLayout.CENTER);
```

Example of Swing Text components

```
JButton submit = new JButton("Submit");
  submit.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String s = "User name: " + name.getText() +
          " (" + name.getSelectedText() +
          " selected)" + "\n" + "Password: " +
          new String(password.getPassword());
        text.setText(s);
      }
    });
  this.add(submit, BorderLayout.SOUTH);
}
```

Editing Complex Text and HTML

`javax.swing.JEditorPane` can display and edit, text, HTML 3.2, and RTF

- Not meant to be a fully-fledged web browser
- Good for displaying help text
- Can be created to view a `URL`
- Uses a `javax.swing.event.HyperlinkListener` to handle hyper links

Because the contents of the `JEditorPane` can get large, it is often embedded in a `javax.swing.JScrollPane`

See `edu.pdx.cs410J.gui.JWebBrowser`

javax.swing.JSlider

A `JSlider` is a "knob" that adjusts some value

- Has major and minor "tick spacing"
- A `ChangeListener` is used to handle adjustment events
- A `JLabel` can be associated with a given value
- The `JSlider` can be either `HORIZONTAL` or `VERTICAL`

```
package edu.pdx.cs410J.gui;

public class HeatIndex extends JPanel {
  public HeatIndex() {
    //...
    final JSlider temp = new JSlider(-20, 120, 65);
    temp.setMajorTickSpacing(20);
    temp.setMinorTickSpacing(5);
    temp.setPaintTicks(true);
    temp.setLabelTable(temp.createStandardLabels(20))
    temp.setPaintLabels(true);
```

`JSlider` example continued

```
final JSlider heatIndex = new JSlider(50, 150);
heatIndex.setOrientation(JSlider.VERTICAL);
heatIndex.setMajorTickSpacing(20);
heatIndex.setMinorTickSpacing(5);
heatIndex.setPaintTicks(true);
Hashtable labels = new Hashtable();
labels.put(new Integer(80),
           new JLabel("Caution"));
labels.put(new Integer(91),
           new JLabel("Extreme Caution"));
labels.put(new Integer(105),
           new JLabel("Danger"));
labels.put(new Integer(129),
           new JLabel("Extreme Danger"));
heatIndex.setLabelTable(labels);
heatIndex.setPaintLabels(true);
double hi =
  computeHeatIndex((double) temp.getValue(),
                   (double) humidity.getValue());
heatIndex.setValue((int) hi);
this.add(heatIndex, BorderLayout.EAST);

temp.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
      int hi = ...;  // Compute index
      heatIndex.setValue(hi);
    }
  });
```

javax.swing.JProgressBar

A JProgressBar is similar to a JSlider

```
package edu.pdx.cs410J.gui;

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

public class JProgressBarExample extends JPanel {
  public JProgressBarExample() {
    this.setLayout(new BorderLayout());

    final JProgressBar progress =
      new JProgressBar(0, 25);
    progress.setBorderPainted(true);
    progress.setStringPainted(true);
    this.add(progress, BorderLayout.NORTH);

    JButton button = new JButton("Click me 25 times");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          progress.setValue(progress.getValue() + 1);
        }
      });
    this.add(button, BorderLayout.SOUTH);
  }
}
```

javax.swing.JComboBox

A `JComboBox` is similar to an AWT `Choice`

- Can be “editable” so that other entries can be added

```
package edu.pdx.cs410J.gui;
import javax.swing.*;

public class JComboBoxExample extends JPanel {
  public JComboBoxExample() {
    String[] flavors =
      { "Vanilla", "Strawberry", "Chocolate",
        "Fudge Swirl", "Pistachio" };
    JComboBox combo = new JComboBox(flavors);
    combo.addItem("Rocky Road");
    combo.setEditable(true);
    combo.setMaximumRowCount(3);
    this.add(combo);
  }
}
```

A `javax.swing.JList` is like a `List`

- Doesn’t automatically scroll, have to wrap a `JScrollPane` around it

- Can select multiple items, etc.

Borders

Every `JComponent` may have a `Border` associated with it

- `javax.swing.border` contains a number of `Border` implementations

- `BevelBorder`, `EmptyBorder`, `LineBorder`, `TitledBorder`, etc.

The easiest way to get a `Border` is to use `javax.swing.BorderFactory`

- Has lots of static factory methods: `createEmptyBorder`, `createTitledBorder`, etc.

Borders are often combined

- An `EmptyBorder` for padding, then surround it with a `TitleBorder`

- Use a `CompoundBorder`

Border example

```
package edu.pdx.cs410J.gui;
import java.awt.Color;
import javax.swing.*;
import javax.swing.border.Border;

public class BorderExample extends JPanel {
  public BorderExample() {
    JButton button = new JButton("Click me");
    Border b = BorderFactory.createMatteBorder(2, 3,
                                    4, 5, Color.RED));
    button.setBorder(b);
    this.add(button);

    String[] entries =
      { "Twenty-seven", "Thirty-nine", "Two-hundred",
        "Four", "Five" };
    JList list = new JList(entries);
    JScrollPane scroll = new JScrollPane(list);
    b = BorderFactory.createTitledBorder("Numbers");
    scroll.setBorder(b);
    this.add(scroll);
  }
}
```

Swing menus

Menus in Swing are the same as the AWT, except:

- `JMenuItem`, `JMenu`, and `JMenuBar` are subclasses of `JComponent`
- `JRadioButtonMenuItem`s can be placed in a `javax.swing.ButtonGroup`
- `JMenuItem`s can have `Icon`s
- A `JMenu` can have a `JSeparator` to group menu items
- A `JApplet` can have a `JMenuBar`

A `JPopupMenu` works the same as a `PopupMenu`

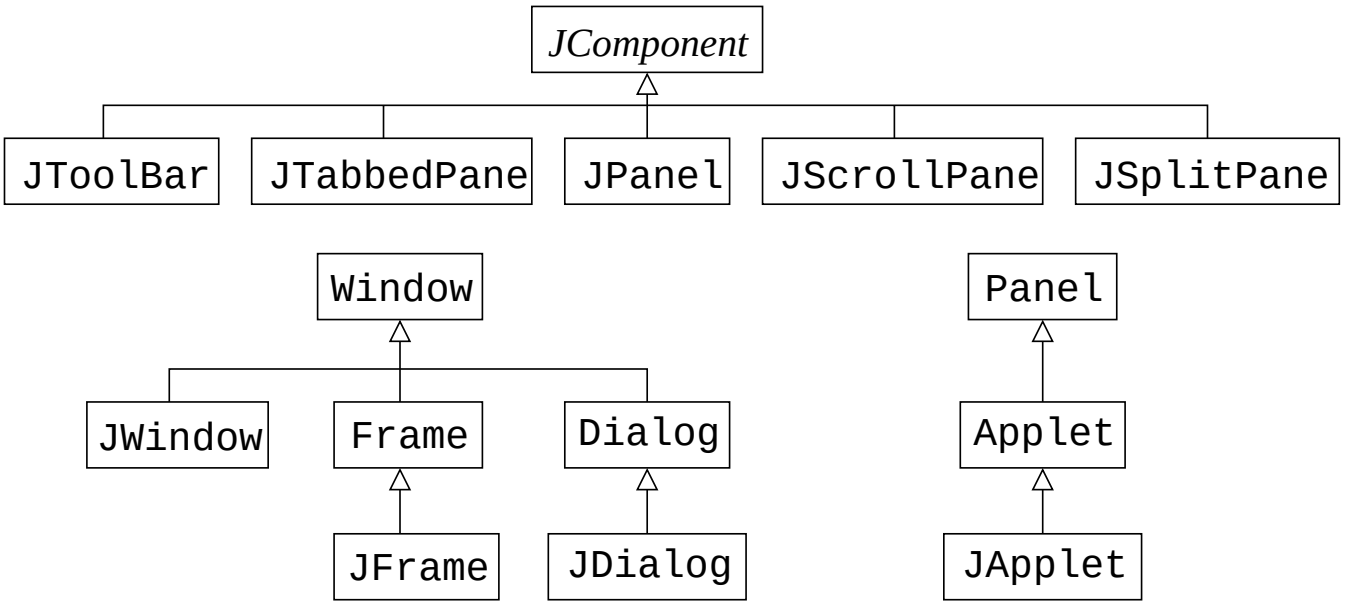
Tool Tips

Swing supports “Tool Tips” that are displayed when the mouse is moved over a widget

- Tool tips can be explicitly created as a `JToolTip`
- `JComponent` has a `setToolTipText` method

```
package edu.pdx.cs410J.gui;
import javax.swing.*;

public class JToolTipExample extends JPanel {

  public JToolTipExample() {
    JButton button = new JButton("Mitchell");
    button.setToolTipText("Push the button, Frank");
    this.add(button);

    JTextField name = new JTextField(8);
    name.setToolTipText("Your name here");
    this.add(name);
  }
}
```

Swing Containers



Swing has its own set of container widgets

- Because `JFrame`, `JDialog`, et. al. subclass AWT containers, they are heavyweight (have native counterparts)
- Widgets are not added directly to a Swing container
- Instead, they are added to its “content pane” (itself a `Container`)

```
JPanel panel = new JPanel();
JFrame frame = new JFrame("A JFrame");
BorderLayout layout = new BorderLayout();
frame.getContentPane().setLayout(layout);
frame.getContentPane().add(panel);
```

`javax.swing.JFrame` and `JApplet`

`JFrame` extends the functionality of `Frame`

- `JFrame` can have a `JMenuBar`
- `JFrame`'s `setDefaultCloseOperation` method specifies what happens when the window is closed:
  - `DO_NOTHING_ON_CLOSE`: `Frame` behavior
  - `HIDE_ON_CLOSE` (default): `JFrame` is hidden, `setVisible(true)` to show it again
  - `DISPOSE_ON_CLOSE`: `JFrame` is disposed
  - `EXIT_ON_CLOSE`: `System.exit()` is called
  - You don't need to write a `WindowAdapter`

`JApplet` lets you use Swing widgets in an applet

- `JApplet` may have a `JMenuBar`
- `JApplet` has a content pane
- By default, `JApplet` has a `BorderLayout` whereas `Applet` has a `FlowLayout`

javax.swing.JToolBar

A `JToolBar` is a container that arranges its contents in a single row or column

- May be named, “floatable”, and have separators

```
package edu.pdx.cs410J.gui;
import javax.swing.*;

public class JToolBarExample extends JPanel {
  public JToolBarExample() {
    JToolBar bar =
      new JToolBar("Tools", JToolBar.HORIZONTAL);
    bar.add(new JButton("A button"));
    bar.add(new JLabel("A label"));
    bar.setFloatable(true);
    this.add(bar);

    bar = new JToolBar();
    bar.setOrientation(JToolBar.VERTICAL);
    bar.setFloatable(false);
    bar.add(new JButton("Another Button"));
    this.add(bar);
  }
}
```

javax.swing.JTabbedPane

A JTabbedPane is a container that consists of multiple components, only one of which is viewable at a time

- “Tabs” along the top select which component is viewable
- Tabs can be at the LEFT, RIGHT, TOP, or BOTTOM
- In the same vein as CardLayout
- Each tab is titled, may be disabled, and may have a tool tip
- No event handling code is necessary

JTabbedPane **example**

```
package edu.pdx.cs410J.gui;

import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.*;

public class JTabbedPaneExample extends JPanel {
  public JTabbedPaneExample() {
    this.setLayout(new BorderLayout());

    JTabbedPane pane =
      new JTabbedPane(JTabbedPane.TOP);
    pane.addTab("Shutdown", new JButton("Shutdown"));

    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(3, 1));
    panel.add(new JCheckBox("Take shower"));
    panel.add(new JCheckBox("Make coffee"));
    panel.add(new JCheckBox("Get the paper"));
    pane.addTab("Options", panel);

    this.add(pane, BorderLayout.CENTER);
  }
}
```

javax.swing.JSplitPane

A JSplitPane provides an adjustable container for displaying two components

- A “divider” splits the JSplitPane with either a HORIZONTAL_SPLIT or a VERTICAL_SPLIT
- The initial position of the divider is specified by a number between 0.0 and 1.0
- With setOneTouchExpandable, arrows on the divider will flush the divider to the edge
- In general, JSplitPane will respect the minimumSize of its components and not clip them

JSplitPane **Example**

```
package edu.pdx.cs410J.gui;
import java.awt.BorderLayout;
import javax.swing.*;

public class JSplitPaneExample extends JPanel {

  public JSplitPaneExample() {
    this.setLayout(new BorderLayout());
    JSplitPane pane =
      new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);

    JPanel left = new JPanel();
    left.add(new JButton("Left"));
    pane.setLeftComponent(left);

    JPanel right = new JPanel();
    right.add(new JButton("Right"));
    pane.setRightComponent(right);

    pane.setOneTouchExpandable(true);
    pane.setDividerLocation(0.75);

    this.add(pane, BorderLayout.CENTER);
  }
}
```

Swing Layouts

Swing added several new layouts, the most interesting of which is `BoxLayout`

- Components are laid out in either a horizontal (x-axis) or vertical (y-axis) line
- `BoxLayout` attemps to respect its components' preferred height and width
- You must let a `BoxLayout` know which container it is laying out

A `Box` is a container whose default layout manager is a `BoxLayout`

- Has static methods for creating `Components` useful with `BoxLayout`
  - `createVerticalStrut`: Returns a `Box` with a fixed height
  - `createHorizontalGlue`: Returns a `Box` whose width will expand to fill any empty space (good for spacing components evenly)

BoxLayout **Example**

```
package edu.pdx.cs410J.gui;
import javax.swing.*;

public class BoxLayoutExample extends JPanel {

  public BoxLayoutExample() {
    this.setLayout(new BoxLayout(this,
                                 BoxLayout.Y_AXIS));

    this.add(new JButton("One"));
    this.add(Box.createVerticalStrut(20));
    this.add(new JLabel("After strut"));
    this.add(Box.createVerticalGlue());
    this.add(new JButton("After glue"));
  }
}
```

Note how the glue expands so that the second button is always at the bottom

And so much more...

What we've seen here is just the beginning of Java's facilities for developing GUIs

- Java2D: Line graphics, affine transformations, image rendering
- Document rendering: Document layout, printing facilities
- Accessibility: Non-traditional I/O devices
- Java3D: 3D Graphics
- Drag and Drop
- Internationalization
- JavaBeans for component-based software development

Summary

Java provides a number of classes for developing widget-based GUIs:

- The Abstract Windowing Toolkit has heavy-weight widgets
- Swing provides lighter-weight widgets an a great deal of functionality
- Widgets generate events that are handled by listeners
- Components are arranged in containers using layouts