**JFC / Swing**

The Java GUI toolkit
(Reading: *The Sun JFC/Swing Tutorial* -- see WWW page)

**JFC - Java Foundation Classes**

JFC is a GUI toolkit. It is the successor of AWT, which is the foundation of JFC (usually have to include AWT with JFC apps).

Swing was the internal development name

Swing replaces ‘heavyweight’ AWT containers with ‘lightweight’ **components** or **widgets**.

+ **Powerful GUI toolkit**: you can do most of what you want using JFC’s built-in features.
- **Complexity**: it’s huge!
- **Efficiency**: it’s slow!

**JFC Components**

*Top-level Containers*
- Applet
- Dialog
- Frame

*General-Purpose Containers*
- Panel
- Scroll pane
- Split pane
- Tool bar
- Tabbed pane

*Special-Purpose Containers*
- Internal frame
- Layered pane
- Root pane
JFC Components (2)

Basic Controls
- Buttons
- Menu
- Label
- Slider
- Text fields

Uneditable Information Displays
- Combo box
- Progress bar
- Tool tip

Editable Displays of Formatted Information
- Color chooser
- File chooser
- Table
- Text
- Tree

Component Features & Attributes

Some features of JFC components/widgets
- Borders and layouts can be manipulated.
- They understand keyboard/mouse events.
- Provide a common style/look-and-feel.
- Properties (= resources) can be easily modified.
- Support for layout via managers.
- Flexible event handling.
- Methods to increase efficiency.

A JFC Example

Application that counts the number of button clicks

Let’s look at the design of this application and some of the code details.
Design of the Application

The application consists of:

- JFrame (window border)
- Content pane (gray rectangle)
- JPanel (gray rectangle)
- JButton (rectangle & text)
- JLabel (text)

JPanel and descendents are added to the content pane.

JFC Example (1)

```java
import javax.swing.*;  //final package name
import java.awt.*; //usually needed.
import java.awt.event.*;

cpyublic class SwingApplication {
    private static String labelPrefix =
        "Number of button clicks: ";
    private int Clicks = 0;

    public Component createComponents() {
        final JLabel label = new
            JLabel(labelPrefix + "0 ");
        JButton button = new JButton(
            "I'm a Swing button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new
            ActionListener() {
                public void
                    actionPerformed(ActionEvent e){
                    Clicks++;
                    label.setText(labelPrefix+Clicks);
                }
            });
    }
}
```

JFC Example (2)

```java
public Component createComponents() {
    final JLabel label = new
        JLabel(labelPrefix + "0 ");
    JButton button = new JButton(
        "I'm a Swing button!");
    button.setMnemonic(KeyEvent.VK_I);
    button.addActionListener(new
        ActionListener() {
            public void
                actionPerformed(ActionEvent e){
            Clicks++;
            label.setText(labelPrefix+Clicks);
        }
    });
```
JFC Example (3)

```java
label.setLabelFor(button);

// Lay component out on the screen.
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
pane.setLayout(new GridLayout(0, 1));
pane.add(button);
pane.add(label);
return pane;
}
```

JFC Example (4)

```java
public static void main(String[] args) {
//Create container and add contents
    JFrame frame = new JFrame("SwingApplication");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    swingApp app = new swingApp();
    Component contents = app.createComponent();
    frame.getContentPane().add(contents, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
}
```

Layout Managers

- Border Layout
- Box Layout
- Card Layout
- Flow Layout
- Grid Layout
- Gridbag Layout
Swing Features & Concepts

Every time a user presses a key or presses a mouse button, an event occurs.

Events must be handled. In Swing, this is done by constructing listeners, which are objects defined to deal with specific events.

Swing widgets need lots of different listeners. Two general kinds are very useful.

<table>
<thead>
<tr>
<th>Action</th>
<th>Listener Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>User clicks a button or chooses a menu item.</td>
<td>ActionListener</td>
</tr>
<tr>
<td>User closes a frame (main window)</td>
<td>WindowListener</td>
</tr>
</tbody>
</table>

Event Handling

**ActionListener**

- Used to handle JFC button clicks, menu item selection, or when a user presses return in a text field. **(Note: this is an interface! You must implement its actionPerformed method)**

**WindowListener**

- User closes a frame (main window)

**MouseListener**

- User presses a mouse button while the cursor is over a component

Event Handling (2)

**MouseMotionListener**

- User moves the mouse over a component

**ComponentListener**

- Component becomes visible where before it was not.

**FocusListener**

- Component gets the keyboard focus

**ListSelectionListener**

- Table or list selection changes
Implementing an Event Handler

Three bits of code are needed:
In declaration for event handler class, code that specifies an implementation of a listener interface, or an extension of such a class.

```java
public class Foo implements ActionListener
```

Code that registers an instance of the event handler class as a listener on one or more widgets

```java
myComponent.addActionListener(instanceOfFoo);
```

Code that implements methods in listener interface:

```java
public void actionPerformed(ActionEvent e) {
    //code reacting to the action. }
```

Example - Beeping Buttons

Suppose we have a single button and we want it to beep when the user clicks on it.

Here's a snippet of the event handling code.

```java
public class Beeper ... implements ActionListener {
    ...
    // button is a JButton object.
    button.addActionListener(this);
    ...
    public void actionPerformed(ActionEvent e) {
        // make a beeping sound }
```

Listeners (Single & Multiple)

Each event is represented by an object: it provides information about event and identifies the source.

Each event source can have multiple listeners registered on it. Conversely, a single listener can register with multiple event sources (and can handle events from each source).

*Hint: When you want to handle events from a component, first check its how-to section: it describes how to handle most common events.*
Example - Multiple Listeners

Example: two buttons.
Click on left, generate
“Cartman Rules!”
on right, generate
“Kenny Rules!”

Applet has two event sources and two listeners.

- One listener listens for events from both buttons.
  When it hears an event on either, it adds the button’s
text to the top text area.
- The second listener listens for events on right
  button. On event, it adds text to the bottom text area.

Example - MultiListeners (2)

Structure of the button event handling code:
```java
public class MultiListener implements ActionListener {
    ... // where initialization occurs:
    button1.addActionListener(this);
    button2.addActionListener(this);
    button2.addActionListener(new Eavesdropper(bottom));
    ...
    public void actionPerformed(ActionEvent e){
        top.append(e.getText()+newline); }
}
public class Eavesdropper implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e){
        myTextArea.append(e.getText()+newline); }
}
```

Data Models and Components

Preview MVC: separate data representation of
Model. JFC provides built-in representations!

Some JFC data models and components that use
them:
- BoundedRangeModel
  (JScrollBar, JProgressBar)
- ButtonModel (AbstractButton)
- ComboBoxModel, MutableComboBoxModel
  (JComboBox)
- ListModel, ListSelectionModel,
  SingleSelectionModel (JList)
- TableModel, TableColumnModel (JTable)
- TreeModel, TreeSelectionModel (JTree)

If you work with a Swing component, you will want
to work with its underlying data model.
**Bounded Range Model**

Represents an increasing set of values (e.g., a progress bar).

The data model keeps track of:
- upper & lower bound for range.
- current value and/or extent

```
+-----------------+  +-----------------+  +-----------------+
|                 |  |                 |  |                 |
| min             |  | value           |  | value + extent  |
|                 |  |                 |  | max             |

extent
```

**List Selection Model**

Represents currently selected items in a list
- interval selection
- element and multiple selection

```
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
index0 index1

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
index0 index1

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
index0 index1

last interval added
```

**Drawing Graphics**

**Graphics** class stores state
- current color, font, clipping region, pixel operation (paint/XOR), translation origin

Primitives:
- lines, ovals, polygons, rectangles, round rectangles, raised/lowered rectangles (’3d’), arcs, text, icons (= images)

AWT: `paint()` method, `repaint()` to activate

JFC: `paintComponent`

More advanced graphics in Java 2D API
JFC Graphics Example

```java
class ImagePanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g); //background
        g.drawLine(50, 50, 100, 100);
        g.setColor(Color.magenta);
        g.fill3DRect(10, 10, 50, 50);
        g.setColor(Color.black);
        g.fillArc(100, 10, 150, 80, 0, 90);
        //draw preloaded image.
        g.drawImage(image, 200, 200, this);
    }
}
```