GUI

Graphical User Interfaces

Reference:
Chapters 12, 13, 14, and 17

Overview: JFC - Java Foundation Classes

- Containers and Components
- containment hierarchy:
  - Component layout using layout managers
  - Event driven programming: events, listeners and sources
- Implementing listeners using:
  - Adapter Classes
  - Anonymous Classes

GUI-based Applications

- Developing GUI based application requires understanding of:
  - Structure of the inheritance hierarchy which defines the behavior and attributes of the components in the GUI of the application.
  - Structure of the containment hierarchy which defines how components are put together in the GUI of the application.
  - Handling of events during the interaction between the GUI and the user.

JFC: Java Foundation Classes

- Java provides JFC for developing GUI based application.
  - AWT (Abstract Window Toolkit) package (java.awt) which mostly uses heavy-weight components.
  - Swing (javax.swing) which mostly uses light-weight components.
- JFC makes it easier to develop GUI based applications
  - Use container and layout managers to design the GUI
  - Use event delegation model to handle events.

JFC: Java Foundation Classes

- Java provides JFC for developing GUI based application.
Components

- The Component class (and its subclasses) provides support for handling events, changing a component's size, control of color and fonts, and painting of components and their contents.
- Note that a component is an object of a concrete subclass of Component class.
- We distinguish between two kinds of components:
  - GUI control components: user interaction is via such components (buttons, checkboxes, menus, etc.)
  - Containers: used for building the GUI, and can be adorned with other components.

Containers

- A container is a component which can contain other components (and thereby can contain other containers since a component can be a container because of inheritance).
- Such a nesting of components defines a containment hierarchy (in contrast to the inheritance hierarchy).
- The add(Component comp) method can be used by a container to nest a component.
- A container uses a layout manager to position the components inside it.
  - The setLayout(…) method can be used to associate a layout manager with a container.
- In the AWT, subclasses of the class Container are all containers.
- In Swing, the class JPanel defines a generic container.

Panels

- A panel is a container (and also a component).
- It is a window which has no title, menus or borders, but can contain other components.
- It is an ideal candidate for grouping components.
  - The inherited add() method can be use to add a component to a panel.
  - A panel uses the FlowLayout manager as the default layout manager.
- Classes java.awt.Panel and javax.swing.JPanel are typical containers that are "stuffed" with other components to build a GUI.
Windows
- The `Window` class can be used to create a *top-level* window which has does not have a title, menus or borders.
- A *top-level* window cannot be incorporated/nested in a container.
- A *top-level* window (and its components) must be *explicitly* made visible by the call `setVisible(true)`, and *explicitly* made invisible by the call `setVisible(false)`.

Frames
- A frame is a container used to create what we usually call a window on the screen.
- It has a title, menus, border, cursor, and an icon.
- A frame is the starting point of a *GUI based application*, and forms the root of a *containment hierarchy*.
  - A frame being a container can contain other panels which in turn can contain other panels and GUI control components.
- Typical frames can be created from `java.awt.Frame` and `javax.swing.JFrame` classes.

Dialog Boxes
- The `java.awt.Dialog` and `java.swing.JDialog` classes can define a *top-level window*, often called a *dialog box*.
- However it has only title and border, and no menus or icon.
- A dialog box can be *modal* (i.e. no other window can be accessed while this window is visible) or *non-modal* (i.e. other windows can be accessed while this window is visible).
- A dialog box can contain other panels which in turn can contain other panels and GUI control components.
- A dialog box is usually used to get input from the user or show information to the user.

Applets
- An applet is a specialized panel which can be embedded in an *applet-context* (for example, a Web browser).
- Since an applet has the `Component` class as superclass:
  - It can draw onto itself by overriding the `paint()` method.
- Since an applet has the `Panel` class as superclass:
  - Other components can be embedded in an applet to create a full-fledged GUI.
- The `java.applet.Applet` and `java.swing.JApplet` classes can be used to implement applets.

GUI control components in AWT
- GUI control components make interaction between the user and the application possible.
- **Input-components**: obtain information from the user, for example, `Button`, `TextField`, `Checkbox`.
- **Output-components**: present information from the user, for example, `TextField`, `TextArea`.
- GUI control components can be added to a container using the `add()` method.
- A layout manager (associated with the container of a component) is used to position the GUI component in the container.
- For AWT, the "look and feel" (LAF) of the GUI is decided by the platform on which the application is run.
- For Swing, LAF can be set by the application.
Information in and out of GUI control components

- **TextArea, TextField:**
  - use `getText()` which returns a String object.
  - use `setText(String str)` to write a String object in the component.
  - use `setEditable(boolean)` if the text is editable or not.
  - Text must be converted to and from the appropriate numerical data type.

- **Checkbox:**
  - use `getState()` if the Checkbox is selected or not.
  - use `setState(boolean state)` to set the state of a Checkbox object equal to the value of the parameter.

Layout Management

- A container uses a layout manager to position components inside it.
- The three most common layout managers are
  - `FlowLayout` manager
  - `BorderLayout` manager
  - `GridLayout` manager
- A layout manager is associated with a container by calling the `setLayout(…)` method.
- Containment hierarchy is usually built by adding components to containers using the `add(…)` method defined for all containers.

FlowLayout manager

- `FlowLayout` manager is the default layout manager for a panel (and thereby all applets).
- `FlowLayout` manager inserts components row-major in a container, from left to right and top to bottom, starting a new row depending on the container’s width and if there is not enough room for all the components.

- `FlowLayout` manager honors the preferred size of the components (i.e. it won’t “stretch” it), but spatial relationships can change depending on the size of the container.

```java
import java.awt.*;

public class FlowLayoutDemo extends Frame {
    FlowLayoutDemo() {
        super("FlowLayoutDemo");
        // Create a checkbox group
        CheckboxGroup sizeOptions = new CheckboxGroup();

        // Create 3 checkboxes and add them to the checkbox group.
        Checkbox cb1 = new Checkbox("Large", sizeOptions, true);
        Checkbox cb2 = new Checkbox("Medium", sizeOptions, false);
        Checkbox cb3 = new Checkbox("Small", sizeOptions, false);
        
        container.setLayout(new FlowLayout()); // not necessary for Panel.
        container.add(component);
    }
}
```
// Create and set a FlowLayout manager
setLayout(new FlowLayout);

// Add the checkboxes
add(cb1);
add(cb2);
add(cb3);

// Show the GUI in the frame
pack();
setVisible(true);

public static void main(String args[]) {
    new FlowLayoutDemo();
}

public GridLayout

A GridLayout manager divides the region of the container into a matrix of rows and columns (i.e. a rectangular grid).

Components have row-major allocation, where each component occupies a cell.

All the cells in the grid have the same size, i.e. same width and height.

The cell size is dependent on the number of components to be placed in the container and the container's size.

A GridLayout manager ignores a component's preferred size, and a component is stretched if possible to fill the cell.

A common practice to avoid components being stretched is first to stick the component in a panel (using FlowLayout manager) and then adding the panel to the container, as the components in the panel will not stretch.

container.setLayout(new GridLayout(2,3)); // 2x3 grid
container.add(comp1);
container.add(comp2);
container.add(comp3);

Example of GridLayout

import java.awt.*;
public class GridLayoutDemo extends Frame {
    GridLayoutDemo() {
        super("GridLayoutDemo");
        // Create 2 labels and 2 text fields
        Label xLabel = new Label("X Coordinate:");
        Label yLabel = new Label("Y Coordinate:");
        TextField xInput = new TextField(5);
        TextField yInput = new TextField(5);
        // Set the font the background color
        xLabel.setFont(new Font("Serif", Font.BOLD, 14));
        xInput.setBackground(Color.yellow);
        yLabel.setFont(new Font("Serif", Font.BOLD, 14));
        yInput.setBackground(Color.yellow);
        // Create and set a GridLayout with 2 x 2 grid
        layout = new GridLayout(2,2);
        // Add the components
        add(xLabel); // [0,0]        add(xInput); // [0,1]
        add(yLabel); // [1,0]        add(yInput); // [1,1]
        // Show the GUI in the frame
        pack();
        setVisible(true);
    }
    public static void main(String args[]) {
        new GridLayoutDemo();
    }
}
**BorderLayout manager**

- `BorderLayout` manager is the default layout manager for a `Frame`.
- `BorderLayout` manager inserts components in the four compass directions ("North", "South", "East", "West") and in the center ("Center") of the container.

```java
import java.awt.*;
public class BorderLayoutDemo extends Frame {
    BorderLayoutDemo() {
        super("BorderLayoutDemo");
        // Create a text field
        TextField msg = new TextField("MESSAGE DISPLAY");
        msg.setEditable(false);
        // Create a button
        Button drawButton = new Button("Draw");
        // Create a label
        Label banner = new Label("Get yourself a Java Jolt!", Label.CENTER);
        banner.setBackground(Color.white);
        banner.setForeground(Color.blue);
        banner.setFont(new Font("Serif", Font.BOLD, 18));
        // Create 2 vertical scrollbars
        Scrollbar sb1 = new Scrollbar(Scrollbar.VERTICAL,0, 10,-50,100);
        Scrollbar sb2 = new Scrollbar(Scrollbar.VERTICAL,0, 10,-50,100);
        // Create and set border layout
        setLayout(new BorderLayout());
        // Add the components in designated regions
        add(drawButton, BorderLayout.NORTH);
        add(msg, BorderLayout.SOUTH);
        add(banner, BorderLayout.CENTER);
        add(sb1, BorderLayout.WEST);
        add(sb2, BorderLayout.EAST);
        // Show the GUI in the frame
        pack();
        setVisible(true);
    }
    public static void main(String args[]) {
        new BorderLayoutDemo();
    }
}
```

**Example of BorderLayout**

```java
// Create a label
Label banner = new Label("Get yourself a Java Jolt!", Label.CENTER);
banner.setBackground(Color.white);
banner.setForeground(Color.blue);
banner.setFont(new Font("Serif", Font.BOLD, 18));
// Create 2 vertical scrollbars
Scrollbar sb1 = new Scrollbar(Scrollbar.VERTICAL,0, 10,-50,100);
Scrollbar sb2 = new Scrollbar(Scrollbar.VERTICAL,0, 10,-50,100);
// Create and set border layout
setLayout(new BorderLayout());
// Add the components in designated regions
add(drawButton, BorderLayout.NORTH);
add(msg, BorderLayout.SOUTH);
add(banner, BorderLayout.CENTER);
add(sb1, BorderLayout.WEST);
add(sb2, BorderLayout.EAST);
// Show the GUI in the frame
pack();
setVisible(true);
}
```

**Events and the AWT Thread**

- GUI based applications are **event-driven**.
- A special thread, called the **AWT thread**, is responsible for interaction with the user.
- **Events** are generated and sent to the application during interaction with the user.
  - An event can give information to the application on what action the user has performed (pressed a mouse button, moved the mouse cursor, pressed a key, closed the window, moved the window, scrolled up, made a menu choice, etc.), and/or how its context has changed (window uncovered, etc.)
- **Event-handling** is done by **event handlers**:
  - **Event-handlers** in the application are responsible for correct handling of events. In Java, these are called **listeners**.
  - A listener is notified of the events it is interested in.
  - A listener should not hoard the AWT thread.
  - A listener should do computation intensive tasks in a separate thread, allowing the AWT thread to continue monitoring the user interaction.
- **Note** that events can occur in an arbitrary sequence, and are usually user initiated.
Events
- EventObject class encapsulates all the information about an event.
- AWTEvent class is the superclass of all classes that represent categories of events generated by components.
- The method getSource() on an event can be used to identify the source of the event.
- Objects of these event classes encapsulate additional information that identifies the exact nature of the event.
- For example, the MouseEvent class categorizes events relating to a mouse-button being clicked (MouseEvent.MOUSE_CLICKED), the mouse being dragged (MouseEvent.MOUSE_DRAGGED).
- These values (MouseEvent.MOUSE_CLICKED, MouseEvent.MOUSE_DRAGGED) constitute an ID for the event. The class java.awt.AWTEvent provides a method that returns an event's ID: The method getID() on an event returns the ID (type) of the event.

Event Hierarchy
java.awt.event.

Some Semantic Event Classes
**ActionEvent**
- This event is generated by an action performed on a component.
- GUI components that generate these events:
  - Button - when a button is clicked.
  - List - when a list item is double-clicked.
  - MenuItem - when the item is selected.
  - TextField - when ENTER key is hit in the text field.
- JButton, JToggleButton and JCheckBox - when a button is clicked.
- JMenuItem, JMenu, JComboBox, JMenuItem, JRadioButtonMenuItem - when the item is selected.
- JTextField - when ENTER key is hit in the text field.

The ActionEvent class provides the following useful methods:
- public String getActionCommand()
  - Returns the command name associated with this action. The command name is a button label, a list-item name, a menu-item name or text depending on whether the component was a Button, List, MenuItem or TextField object.

**ItemEvent**
- This event is generated when an item is selected or deselected in an ItemSelectable component.
- GUI components that generate these events:
  - Checkbox - when the state of a checkbox changes.
  - CheckboxMenuItem - when the state of a checkbox associated with a menu item changes.
  - Choice - when an item is selected or deselected in a choice-list.
  - List - when an item is selected or deselected from a list.
- JButton, JToggleButton and JCheckBox - when the state of the item changes.
- JMenuItem, JMenu, JComboBox, JMenuItem, JRadioButtonMenuItem - when the state of the menu item changes.

The ItemEvent class provides the following useful methods:
- public Object getSelectedItem()
  - The object returned is actually a String object containing the label of the item.
- public int getStateChange()
  - The returned value indicates whether it was a selection or a de-selection that took place, given by the two constants from the ItemEvent class:
    - public static final int SELECTED
    - public static final int DESELECTED
Some Low-level Events

**KeyEvent**
- This class is a subclass of the abstract `InputEvent` class.
- This event is generated when the user presses or releases a key, or types (i.e., both presses and releases) a character.

**MouseEvent**
- This class is a subclass of the abstract `InputEvent` class.
- This event is generated when the user moves the mouse or presses a mouse button.

**WindowEvent**
- This event is generated when an important operation is performed on a window. These operations are identified by the following constants in the `WindowEvent` class:
  - `public static final int WINDOW_OPENED`
  - `public static final int WINDOW_CLOSING`
  - `public static final int WINDOW_CLOSED`
  - `public static final int WINDOW_ICONIFIED`
  - `public static final int WINDOW_DEICONIFIED`
  - `public static final int WINDOW_ACTIVATED`
  - `public static final int WINDOW_DEACTIVATED`
- The inherited `getID()` method returns the specific type of the event denoted by one of the constants given above.
- These events are generated by the `Window` class and its subclasses.

Event Delegation Model

![Event Delegation Model Diagram]

Setting up Sources and Listeners

- A **source** is an object which can generate events.
- A **listener** is an object which is interested in being informed when certain events occur.
  - **STEP 1**: A listener must first **register** itself with the source(s) which can generate these events.
  - Sources inform listeners when events occur, sending the necessary information about the events.
  - A source of a particular event calls a special method in all the listeners registered for receiving notification about this event.
  - **STEP 2**: The listener must guarantee that the method exists by undertaking to implement a **listener interface** for this event.
- Any object can be a listener as long as it implements the right interface (XListener) for the specific event (XEvent), and registers itself (addListener()) with a source that generates this event.

Note that subclasses of a component can generate the same events as the superclass component because of inheritance.

Registering and Removing Listeners of Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Source</th>
<th>Methods which the source provides to register and remove listeners who are interested in the event generated by the source.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>Button, List, Menu, TextField</td>
<td>addActionListener, removeActionListener ActionListener</td>
</tr>
<tr>
<td></td>
<td>JComponent, JMenuBar, JMenu, JCheckBox, JRadioButton, JToggleButton, JMenuComponent</td>
<td>addActionListener, removeActionListener ActionListener</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>Choice, Checkbox, List, Menu, List</td>
<td>addItemListener, removeItemListener ItemListener</td>
</tr>
<tr>
<td></td>
<td>JComponent, JMenuBar, JMenu, JCheckBox, JRadioButton, JMenuComponent</td>
<td>addItemListener, removeItemListener ItemListener</td>
</tr>
</tbody>
</table>
### Registering and Removing Listeners of Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Source</th>
<th>Methods which the source provides to register and remove listeners who are interested in the event generated by the source.</th>
<th>Interface which a listener for a particular event must implement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyEvent</td>
<td>Component</td>
<td><code>addKeyListener</code>, <code>removeKeyListener</code></td>
<td>KeyListener</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>Component</td>
<td><code>addMouseListener</code>, <code>removeMouseListener</code>&lt;br&gt;<code>addMouseMotionListener</code>, <code>removeMouseMotionListener</code></td>
<td>MouseListener</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>Window</td>
<td><code>addWindowListener</code>, <code>removeWindowListener</code></td>
<td>WindowListener</td>
</tr>
</tbody>
</table>

### Listener Interfaces

<table>
<thead>
<tr>
<th>Listener Interfaces</th>
<th>Methods in listener interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyListener</td>
<td><code>keyPressed(KeyEvent e)</code>&lt;br&gt;<code>keyReleased(KeyEvent e)</code>&lt;br&gt;<code>keyTyped(KeyEvent e)</code></td>
</tr>
<tr>
<td>MouseListener</td>
<td><code>mouseClicked(MouseEvent e)</code>&lt;br&gt;<code>mouseEntered(MouseEvent e)</code>&lt;br&gt;<code>mouseExited(MouseEvent e)</code>&lt;br&gt;<code>mousePressed(MouseEvent e)</code>&lt;br&gt;<code>mouseReleased(MouseEvent e)</code></td>
</tr>
<tr>
<td>MouseMotionListener</td>
<td><code>mouseDragged(MouseEvent e)</code>&lt;br&gt;<code>mouseMoved(MouseEvent e)</code></td>
</tr>
<tr>
<td>WindowListener</td>
<td><code>windowActivated(WindowEvent e)</code>&lt;br&gt;<code>windowClosed(WindowEvent e)</code>&lt;br&gt;<code>windowClosing(WindowEvent e)</code>&lt;br&gt;<code>windowDeactivated(WindowEvent e)</code>&lt;br&gt;<code>windowIconified(WindowEvent e)</code>&lt;br&gt;<code>windowIconRestored(WindowEvent e)</code>&lt;br&gt;<code>windowOpened(WindowEvent e)</code></td>
</tr>
</tbody>
</table>

### Example One: Simple Event Handling

Application consists of a simple window which has a "Quit" button. The application terminates when the button is clicked.

**Listener Registration:**

```java
SimpleWindow.addActionListener(quitter)
```

**Event Handling:**

```java
quitter:QuitHandler
```

```java
quitter:QuitListener
```

```java
addActionListener(quitter)
```

```java
 listeners.add(quitter)
```

```java
quitter.actionPerformed(ActionEvent e)
```
Steps to create a GUI based application:

Button quitButton;
QuitHandler quitter;

• Set up the GUI:
  // Create a button
  quitButton = new Button("Quit");
  // Set a layout manager, and add the button to the window.
  setLayout(new FlowLayout(FlowLayout.CENTER));
  add(quitButton);

• Register listener with the source:
  – Create a listener:
    quitter = new QuitHandler(this);            // (1)
    Note that we pass the window reference to the listener which has access to information from the window in order to handle the event.
  – Register the listener (quitter) with the source (button quitButton):
    quitButton.addActionListener(quitter);         // (2)
    Note that the source (Button) generates ActionEvent when the button is clicked, so that we use the addActionListener method from the Button class to register the listener.

Make sure that the listener implements the right XListener interface.

// Definition of the Listener
class QuitHandler implements ActionListener {                   // (3)
  private SimpleWindow application;    // The associated application

  public QuitHandler(SimpleWindow window) {
    application = window;
  }

  // Invoked when the user clicks the quit button.
  public void actionPerformed(ActionEvent evt) {              // (4)
    if (evt.getSource() == application.quitButton) {        // (5)
      System.out.println("Quitting the application.");
      application.dispose();                              // (6)
      System.exit(0);                                     // (7)
    }
  }
}

Example One: Simple Event Handling

/** A simple application to demonstrate the Event Delegation Model */
import java.awt.*;import java.awt.event.*;public class SimpleWindow extends Frame {
  Button quitButton;                 ...
  QuitHandler quitter;               ...
  public SimpleWindow() {
    super("SimpleWindow");
    // Create one button
    quitButton = new Button("Quit");
    // Set a layout manager, and add the button to the window.
    setLayout(new FlowLayout(FlowLayout.CENTER));
    add(quitButton);
    // Create and add the listener to the button
    quitter = new QuitHandler(this);            // (1)
    quitButton.addActionListener(quitter);         // (2)
    // Set the window size and pop it up.
    pack();
    setVisible(true);
  }
  /** Create an instance of the application */
  public static void main(String args[]) { new SimpleWindow(); }
}

Example Two: Event Handling

Application terminates when the close button of the window is clicked.

Listener Registration:

addWindowListener(quitter)

EventHandler:

windowClosing(WindowEvent e)
windowOpened(WindowEvent e)
windowIconified(WindowEvent e)
...
• Adding the extra functionality.

```java
public class SimpleWindowTwo extends Frame {
    // ...
    QuitHandlerTwo quitter;  // The listener
    public SimpleWindowTwo() {
        // ...
        // Add the listener to the window
        addWindowListener(quitter);  // (3)
        // ...
    }
    // ...
}

class QuitHandlerTwo implements ActionListener, WindowListener {   // (4)
    private SimpleWindowTwo application; // The associated application
    public QuitListenerTwo(SimpleWindowTwo window) {
        application = window;    }
    // Terminate the application.
    private void terminate() {                                  // (5)
        System.out.println("Quitting the application.");
        application.dispose();        System.exit(0);
    }

    // Invoked when the user clicks the close-box
    public void windowClosing(WindowEvent evt) {                // (6)
        terminate();
    }
    // Unused methods of the WindowListener interface.             (7)
    public void windowOpened(WindowEvent evt) {}
    public void windowIconified(WindowEvent evt) {}
    public void windowDeiconified(WindowEvent evt) {}
    public void windowDeactivated(WindowEvent evt) {}
    public void windowClosed(WindowEvent evt) {}
    public void windowActivated(WindowEvent evt) {}
}
```

For a given event category, the listener receives notification of all the different types of events in the category represented by the methods of the corresponding listener interface.

- The listener QuitHandlerTwo implements the interface WindowListener which has seven methods. Some methods are just stubs.
- The listener QuitHandlerTwo receives notification of seven types of events typified by the methods of the WindowListener interface.

### Example Two: Event Handling

```java
/* SimpleWindowTwo: A simple setup for Event Delegation Model */
import java.awt.*;
import java.awt.event.*;
/** A simple application to demonstrate the Event Delegation Model */
public class SimpleWindowTwo extends Frame {
    Button quitButton;                   // The source
    quitButton.addActionListener(quitter);              // (2)
    // Add the listener to the window
    addWindowListener(quitter);                         // (3)
    // Set the window size and pop it up.
    pack();        setVisible(true);
}
```

```java
/** Create an instance of the application */
public static void main(String args[]) { new SimpleWindowTwo(); }
```
Event Listener Adapters

- Event listener adapters can be used to simplify implementation of event listeners.
- Such adapter classes provide a default implementation (which is just a stub) of the methods in a listener interface, so that a listener can extend an adapter and override the appropriate listener methods.

```java
// Definition of the Listener
class QuitHandlerTwo extends WindowAdapter implements ActionListener { // (4)
    // ...
    // Overrides the appropriate interface method
    public void windowClosing(WindowEvent evt) {                // (6)
        terminate();
    }
}

Note that the QuitHandlerTwo class now cannot extend any other class.
```

Anonymous Classes

- An anonymous class is an inner class which is without a name.
- Anonymous classes combine the process of definition and instantiation into one step.
- As these classes do not have a name, an instance of the class can only be created together with the definition.
- Anonymous classes are defined at the location they are instantiated using additional syntax with the new operator.
  - An object of such a class can access methods in its enclosing context.

```java
class AnonClassExample {
    // ...
    A extendA() {
        return new A() { // (1)
            void print() {
                super.print();
                System.out.println(b);
            }
        };
    }

    public static void main (String args[]){
        AnonClassExample e = new AnonClassExample();
        A a = e.extendA();
        a.print(); // (2)
    }
}
```

Extending an existing class

```java
new <superclass name> (<optional argument list>)
{ <class body> }
```

- Optional arguments can be specified which are passed to the superclass constructor.
  - Thus the superclass must provide a corresponding non-default constructor if any arguments are passed.
  - An anonymous class cannot define constructors (as it does not have a name), an instance initializer can be used.
  - <superclass name> is the name of the superclass extended by the anonymous class.
  - Note no extends-clause is used in the syntax.

```java
class A {
    int a = 5;
    int b = 10;
    void print() {
        System.out.println(a);
    }
}
```

```java
class AnonClassExample {
    // ...
    A extendA() {
        return new A() { // ()
            void print() {
                super.print();
                System.out.println(b);
            }
        };
    }

    public static void main (String args[]){
        AnonClassExample e = new AnonClassExample();
        A a = e.extendA();
        a.print(); // (2)
    }
}
```

- Note that at (1) the anonymous class overrides the inherited method print() which is invoked at (2).
- Usually it makes sense to either overrides methods from the superclass or implement abstract methods from the superclass.
- As references to an anonymous class cannot be declared, its functionality is only available through superclass references.
Implementing an interface

new <interface name>() { <class body> }

- An anonymous class provides a single interface implementation, and no arguments are passed.
- The anonymous class implicitly extends the Object class.
- Note that no implements-clause is used in the syntax.
- A typical usage is implementing adapter classes.

// ...
Button b;
b.addActionListener(    new ActionListener() { // (1)
    public void actionPerformed(ActionEvent e) {
        System.out.println("Action performed.");
    }
    }
    );

Example: Listeners using Anonymous Classes

Listener Registration:
- addActionListener($1)
- addWindowListener($2)

Event Handling:
- actionPerformed(ActionEvent e)
- windowClosing(WindowEvent e)

Example: Listeners as Anonymous Classes

/*
*/
import java.awt.*;
import java.awt.event.*;
public class SimpleWindowThree extends Frame {
    Button quitButton;
    public SimpleWindowThree() {
        super("SimpleWindowThree");
        // Create one button
        quitButton = new Button("Quit");
        // Set a layout manager, and add the button to the window.
        setLayout(new FlowLayout(FlowLayout.CENTER));
        add(quitButton);
        // Create and add the listener to the button
        quitButton.addActionListener(new ActionListener() { // (1) $1
            public void actionPerformed(ActionEvent e) {
                // (2) terminate();
            }
        });
        // Create and add the listener to the window
        addWindowListener(new WindowAdapter() { // (3) $2
            // (4) terminate();
        });
        pack();
        setVisible(true);
    }
    private void terminate() {
        System.out.println("Quitting the...");
    }
    /** Create an instance of the application */
    public static void main(String args[]) {
        new SimpleWindowThree();
    }
}
Programming Model for GUI-based Applications

The programming model comprises of three parts:
2. Registration of listeners with sources.
3. Listeners must implement the appropriate listener interfaces, i.e. actions to be performed when events occur.

Steps in developing a GUI Application

- Draw the GUI design first.
  - Group components into panels, with a Frame object as root of the containment hierarchy.
- For the root window, decide a layout manager.
  - use the method setLayout(aLayoutManager)
- For each panel, decide a layout manager.
  - use the method setLayout(someOtherLayoutManager)
- For each panel, add the relevant components to it.
  - use the method add(guiComponent)
  - Add each child component to the parent container, and these containers to their parents upwards in the containment hierarchy.
- Set up event handling:
  - Add listeners to the sources using the addXListener(listener) method for handling XEvent.
- Set preferred size of the root window and make it (and rest of the containment hierarchy) visible.
  - use the method pack()
  - use the method setVisible(true)

Example: Modal Dialog Boxes

- When the user clicks on the "Read an Integer" button in the main window (a), a input window (b) is created to read the number.
- The input window to read the number is modal, so that the user cannot access other windows while this window is showing.
- Data validation: value read must be checked to ensure that only a valid integer is registered.
- Clicking the Ok button in the input window results in the value being validated, and only if it is legal, it is passed to the main window and then only the input window is closed.
- The user can close the input window by clicking the close box, but then no value is passed to the main window.

```java
// Modal dialog boxes.
import java.awt.*;
import java.awt.event.*;

public class ModalDialogDemo extends Frame implements ActionListener {
    private Button intButton;
    private TextField intTF;

    ModalDialogDemo() {
        super("ModalDialogDemo");
        intTF = new TextField("0", 10);
        intTF.setEditable(false);
        intButton = new Button("Read an Integer");

        intTF.addXListener(new XAdapter() {                
            @Override
            public void xValueChanged(XEvent event) {                
                System.out.println("xValue changed: ", event.getValue());                
            }
        });

        intButton.addActionListener(this);

        add(intTF); add(intButton);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked.");
    }
}
```

Example: Modal Dialog Boxes (cont.)

- Setup the main window.
  - Register window as listener with button.
Example: Modal Dialog Boxes (cont.)

```java
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent ev) {
        terminate();
    }
});
pack();
setVisible(true);
```

Register listener for closing the window

Necessary to free the resources.

Data Validation

Necessary for making the window visible.

Register listener for closing the window.

Constructor must have parent.

Setup for Dialog window.

Example: Dialog boxes (cont.)

```java
class IntegerInputDialog extends Dialog {
    ModalDialogDemo app;
    private TextField intTF;
    private Button okButton;
    IntegerInputDialog(ModalDialogDemo f) {
        super(f, "IntegerDialogBox", true);
        app = f;
        // GUI
        okButton = new Button("OK");
        intTF = new TextField(20);
        intTF.setEditable(true);
        add(intTF, BorderLayout.NORTH);
        add(okButton, BorderLayout.SOUTH);
        // Listeners
        okButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                app.setInteger(intTF.getText());
                removeDialogBox();
            }
        });
    }
}
```

Register listeners with sources.

Create a Dialog-window with the main window as parent.

Necessary to free the resources.

Data Validation

Necessary for making the window visible.

Register listener for closing the window.

Constructor must have parent.

Setup for Dialog window.

Example: Dialog boxes (cont.)

```java
public static void main(String args[]) {
    new ModalDialogDemo();
}
```

Modal Dialog Boxes - Swing version

- Notice in Swing, the root of the containment hierarchy is the content pane of the top-level window.

```java
// Get the content pane of the top level window
Container contentPaneForMainWindow = getContentPane();
contentPaneForMainWindow.setLayout(new FlowLayout());
contentPaneForMainWindow.add(intTF);
```

- Set the layout manager for the content pane

```java
// Add components to the content pane
contentPaneForMainWindow.add(intButton);
```
// Modal dialog boxes - Swing version
// Components are added to the content pane of the top level window,
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DialogBoxDemoSwing extends JFrame implements ActionListener {
    private JButton intButton;
    private JTextField intTF;
    DialogBoxDemoSwing() {
        ...      intTF = new JTextField("0", 10);
        intTF.setEditable(false);
        intButton = new JButton("Read an Integer");

        // Get the content pane
        Container contentPaneForMainWindow = getContentPane();
        // Set the layout manager for the content pane
        contentPaneForMainWindow.setLayout(new FlowLayout());
        // Add components to the content pane
        contentPaneForMainWindow.add(intTF);
        contentPaneForMainWindow.add(intButton);
        intButton.addActionListener(this);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                terminate();
            }
        });
        pack();
        setVisible(true);
    }
    public void setInteger(String str) {
        intTF.setText(str);
    }
    public void actionPerformed(ActionEvent ev) {
        new IntegerDialogBoxSwing(this);
    }
    public void terminate() {
        setVisible(false);
        dispose();
        System.exit(0);
    }
    public static void main(String args[]) {
        new DialogBoxDemoSwing();
    }
}

class IntegerDialogBoxSwing extends JDialog implements ActionListener {
    private DialogBoxDemoSwing app;
    private JTextField intTF;
    private JButton okButton;
    IntegerDialogBoxSwing(DialogBoxDemoSwing f) {
        super(f, "IntegerDialogBoxSwing", true); // modal
        app = f;
        // GUI
        okButton = new JButton("OK");
        intTF = new JTextField(20);
        intTF.setEditable(true);
        // Get the content pane
        Container contentPaneForDialogWindow = getContentPane();
        // Add components to the content pane
        contentPaneForDialogWindow.add(intTF, BorderLayout.NORTH);
        contentPaneForDialogWindow.add(okButton, BorderLayout.SOUTH);
        // Listeners
        okButton.addActionListener(this);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                removeDialogBox();
            }
        });
        pack();
        setVisible(true);
    }
    public boolean isLegalInteger(JTextField tf) {
        try { Integer.parseInt(tf.getText()); }
        catch (NumberFormatException ex) { return false; }
        return true;
    }
    public void removeDialogBox() {
        setVisible(false);
        dispose();
    }
}

// Listeners
okButton.addActionListener(this);
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent ev) {
        removeDialogBox();
    }
});
pack();
setVisible(true);

public void actionPerotmed(EventArgs ev) {
    if (isLegalInteger(intTF)) {
        app.setInteger(intTF.getText());
        removeDialogBox();
    }
    public boolean isLegalInteger(JTextField tf) {
        try { Integer.parseInt(tf.getText()); } catch (NumberFormatException ex) { return false; } return true;
    }
    public void removeDialogBox() {
        setVisible(false);
        dispose();
    }
}
// end class IntegerDialogBoxSwing