GUI

Graphical User Interfaces

Overview: AWT (Abstract Window Toolkit)

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

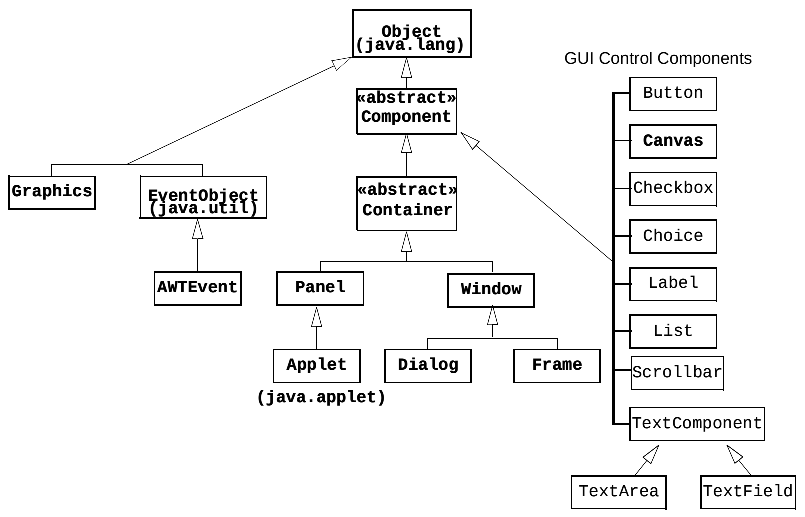
GUI-based Applications

- 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 *component 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.

Inheritance Hierarchy: `java.awt.*`



Component

- All non-menu related components for building a GUI are derived from the *abstract* class `Component`.
- 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.
- A component (and any of its child components) can be made visible or invisible by calling the method `setVisible(boolean)` method with the appropriate argument.
- Note that a component is an object of a *concrete subclass* of `Component` class.

Container

```
java.lang.Object
   |
   +----java.awt.Component (abstract)
           |
           +----java.awt.Container (abstract)
```

- A container is an object of a *concrete subclass* of `Container` class.
- *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).
- The abstract class `Container` defines methods for nesting of other component objects in a container.
  - Such a nesting defines a *component 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.

Panel

```
java.lang.Object
   |
   +----java.awt.Component (abstract)
           |
           +----java.awt.Container (abstract)
                   |
                   +----java.awt.Panel
```

- The `Panel` class is a concrete subclass of the `Container` class.
- A panel is a container and 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.

Window

```
java.lang.Object
   |
   +----java.awt.Component (abstract)
           |
           +----java.awt.Container (abstract)
                   |
                   +----java.awt.Window
```

- 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)`.

Frame

```
java.lang.Object
   |
   +----java.awt.Component (abstract)
           |
           +----java.awt.Container (abstract)
                   |
                   +----java.awt.Window
                           |
                           +----java.awt.Frame
```

- The `Frame` class can be used to create what we usually call a window on the screen.
- It has a title, menus, border, cursor, and an icon.
- A `Frame` object is usually the starting point of a GUI based application, and forms the root of a component hierarchy.
  - A `Frame` object being a container can contain other panels which in turn can contain other panels and GUI control components.

Dialog

```
java.lang.Object
   |
   +----java.awt.Component (abstract)
           |
           +----java.awt.Container (abstract)
                   |
                   +----java.awt.Window
                           |
                           +----java.awt.Dialog
```

- The `Dialog` class also defines 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 *modeless* (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.

Canvas

```
java.lang.Object
   |
   +----java.awt.Component (abstract)
           |
           +----java.awt.Canvas
```

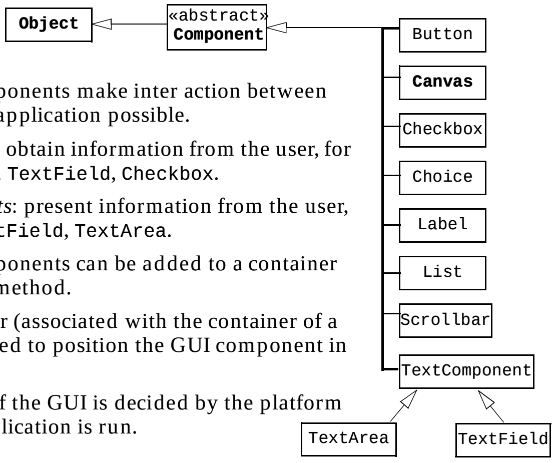
- The `Canvas` class is a subclass of `Component` class, and defines a *GUI control component*.
- Subclassing the `Canvas` class is one way of implementing new components as a canvas has no visible representation.
  - Visuals can be drawn in a canvas by overriding the `paint(Graphics gfx)` method from the `Component` class.

Applet

```
java.lang.Object
   |
   +----java.awt.Component (abstract)
           |
           +----java.awt.Container (abstract)
                   |
                   +----java.awt.Panel
                           |
                           +----java.applet.Applet
```

- 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.

GUI control components



- GUI control components make inter action 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.
- "Look and feel" of the GUI is decided by the platform on which the application is run.

Information in and out of GUI control components

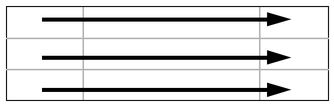
- `TextArea`, `TextField`:
  - use `getText()` which returns a `String` object.
  - use `setText(String str)` to write a `String` object in the component.
  - use `setEditable(boolean)` if the text is editable or not.
  - Text must be converted to and from the appropriate numerical data type.
- `Checkbox`:
  - 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.
- Component hierarchy is usually built by adding components to containers using the `add()` method defined for all containers.

`FlowLayout` manager

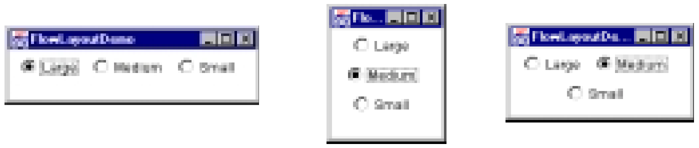
- `FlowLayout` manager is the default layout manager for a panel (and thereby all applets).
- `FlowLayout` manager inserts components *row-major* in a container, from left to right and top to bottom, starting a new row depending on the container's width and if there is not enough room for all the components.



- `FlowLayout` manager honors the preferred size of the components, but spatial relationships can change depending on the size of the container.

```
container.setLayout(new FlowLayout()); // not necessary for Panel.
container.add(component);
```

Example of `FlowLayout`

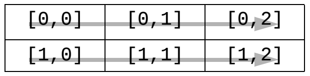


```
import java.awt.*;

public class FlowLayoutDemo extends Frame {
    FlowLayoutDemo() {

        super("FlowLayoutDemo");
        // Create a checkboxgroup
        CheckboxGroup sizeOptions = new CheckboxGroup();

        // Create 3 checkboxes and add them to the checkboxgroup.
        Checkbox cb1 = new Checkbox("Large",  sizeOptions, true);
        Checkbox cb2 = new Checkbox("Medium", sizeOptions, false);
        Checkbox cb3 = new Checkbox("Small",  sizeOptions, false);

        // Create and set a FlowLayout manager
        setLayout(new FlowLayout());
```

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

        // Show the GUI in the frame
        setSize(200, 100);
        setVisible(true);
    }

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

`GridLayout` manager

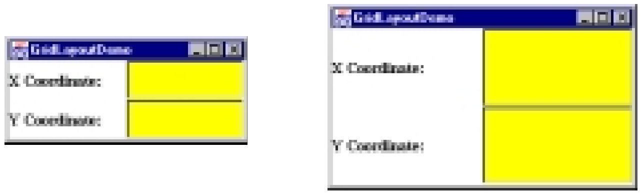
- A `GridLayout` manager divides the region of the container in to a matrix of rows and columns (i.e. a rectangular grid).



- 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);
```

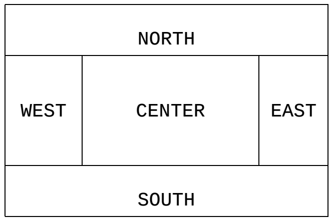
Example of `GridLayout`



```
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));
        yLabel.setFont(new Font("Serif", Font.BOLD, 14));
        xInput.setBackground(Color.yellow);
        yInput.setBackground(Color.yellow);

        // Create and set a GridLayout with 2 x 2 grid
        setLayout(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
        setSize(200, 100);
        setVisible(true);
    }

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

`BorderLayout` manager

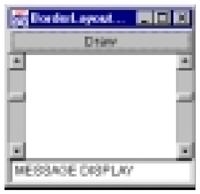
- `BorderLayout` manager is the default layout manager for a `Frame`.
- `BorderLayout` manager inserts components in the four compass directions (`"North"`, `"South"`, `"East"`, `"West"`) and in the center (`"Center"`) of the container.

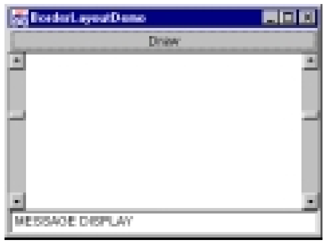


- `BorderLayout` manager does *not* honor the preferred size of the components, but spatial relationships remain the same regardless of the change in the size of the container.

```
container.setLayout(new BorderLayout()); // not necessary for Frame.
container.add(comp1, BorderLayout.NORTH);
container.add(comp2, BorderLayout.SOUTH);
```

Example of `BorderLayout`





```
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 canvas
        Canvas drawRegion = new Canvas();
        drawRegion.setSize(150,150);
        drawRegion.setBackground(Color.white);
        // 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(drawRegion, BorderLayout.CENTER);
        add(sb1, BorderLayout.WEST);
        add(sb2, BorderLayout.EAST);
        // Show the GUI in the frame
        setSize(200, 100);
        setVisible(true);
    }

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

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. I 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.

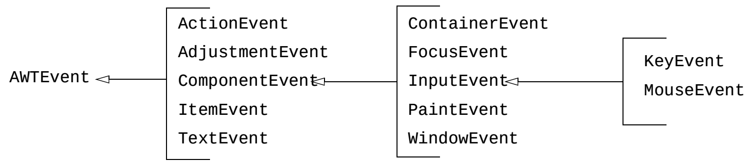
Events

```
java.lang.Object
   |
   +----java.util.EventObject
           |
           +----java.awt.AWTEvent (abstract)
```

- `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.*`



Semantic Event Classes

- The following event classes represent semantic events:

`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 a menu item is selected. • `TextField - 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.
`AdjustmentEvent`	This event is generated when adjustments are made to an adjustable component like a scroll bar. GUI component that generates these events: • `Scrollbar` - when any adjustment is made using the slider or the end-arrows of the scroll bar. The `AdjustmentEvent` class provides the following useful method: • `public int getValue()` Returns the *current value* denoted by the adjustable component.

`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.

The `ItemEvent` class provides the following useful methods:

- `public Object getItem()`
  The object returned is actually a `String` object containing the label of the checkbox or the `CheckMenuItem`, or the label of the item in a choice or a list.
- `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
```

`TextEvent`

This event is generated when contents of a text component are changed.
GUI component that generates these events are subclasses of the `TextComponent` class:

- `TextArea`
- `TextField`

Low-level Events

- The following event classes generate 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.
These situation are characterized by the following constants in the `KeyEvent` class:

- `public static final int KEY_PRESSED`
  This event is delivered when a key is pressed.
- `public static final int KEY_RELEASED`
  This event is delivered when a key is released.
- `public static final int KEY_TYPED`
  This event is delivered when a key is typed, i.e. that a key is pressed and then released to signify typing a character.

The inherited `getID()` method returns the specific type of the event denoted by one of the constants given above.
In addition, the inherited method `getWhen()` from the parent class `InputEvent` can be used to get the time when the event took place.
These events are generated by the `Component` class and its subclasses.
The `KeyEvent` class provides the following useful methods:

- `public int getKeyCode()`
  For `KEY_PRESSED` or `KEY_RELEASED` `KeyEvent`s, this method can be used to get the actual key (called *virtual key*) that was pressed or released.
- `public char getKeyChar()`
  For `KEY_TYPED` `KeyEvent`s, the method can be used to get the Unicode character that resulted from hitting a key.

`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. The exact action is identified by the following constants in the `MouseEvent` class:

- `public static final int MOUSE_PRESSED`
  This event is delivered when a mouse button is pressed.
- `public static final int MOUSE_RELEASED`
  This event is delivered when a mouse button is released.
- `public static final int MOUSE_CLICKED`
  This event is delivered when a mouse button is pressed and released without any intervening mouse dragging.
- `public static final int MOUSE_DRAGGED`
  This event is delivered when the mouse is dragged, i.e. moved while a mouse button is pressed.
- `public static final int MOUSE_MOVED`
  This event is delivered when the mouse is moved without any mouse button pressed.
- `public static final int MOUSE_ENTERED`
  This event is delivered when the mouse crosses the boundary of a component and enters it.
- `public static final int MOUSE_EXITED`
  This event is delivered when the mouse crosses the boundary of a component and exits it.

The inherited `getID()` method returns the specific type of the event denoted by one of the constants given above.
In addition, the inherited method `getWhen()` from the parent class `InputEvent` can be used to get the time when the event took place.
These events are generated by the `Component` class and its subclasses.

`MouseEvent` (cont.)

The `MouseEvent` class provides the following useful methods:

- `public int getX()`
- `public int getY()`
- `public Point getPoint()`
  These methods can be used to get the x- and/or y-position of the event relative to the source component.
- `public synchronized void translatePoint(int x, int y)`
  Translates the coordinate position of the event by x, y.
- `public int getClickCount()`
  Return the number of mouse clicks associated with the event. This is useful for detecting such events as double clicks.

`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` This event is delivered only once for a window when it is created, opened and made visible the first time. • `public static final int WINDOW_CLOSING` This event is delivered when the user action dictates that the window should be closed. The application should explicitly call either `setVisible(false)` or `dispose()` on the window because of this event. • `public static final int WINDOW_CLOSED` This event is delivered after the window has been closed as the result of a call to `setVisible(false)` or `dispose()`. • `public static final int WINDOW_ICONIFIED` This event is delivered when the window is iconified. • `public static final int WINDOW_DEICONIFIED` This event is delivered when the window is de-iconified. • `public static final int WINDOW_ACTIVATED` This event is delivered when the window is activated. • `public static final int WINDOW_DEACTIVATED` This event is delivered when the window is de-activated. 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. The `ComponentEvent` class provides the following useful method: • `public Window getWindow()` This method returns the `Window` object that caused the event to be generated.

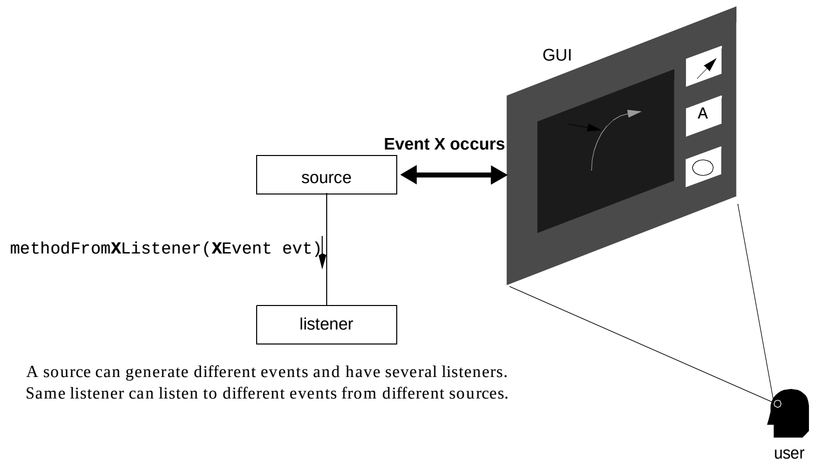
Components and Events

Component	Event
`Button`	`ActionEvent`
`Checkbox`	`ItemEvent`
`CheckboxMenuItem`	`ItemEvent`
`Choice`	`ItemEvent`
`List`	`ActionEvent` `ItemEvent`
`MenuItem`	`ActionEvent`
`Scrollbar`	`AdjustmentEvent`
`TextArea`	`TextEvent`
`TextField`	`ActionEvent` `TextEvent`

`Component`	`ComponentEvent`
	`FocusEvent`
	`KeyEvent`
	`MouseEvent`

`Container`	`ContainerEvent`
`Window`	`WindowEvent`

Event Delegation Model



A source can generate different events and have several listeners.
Same listener can listen to different events from different sources.

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 (`addXListener()`) 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

Event	Source	Methods which the source provides to register and remove listeners who are interested in the event generated by the source.	Interface which a listener for a particular event must implement.
ComponentEvent	Component	add**Component**Listener remove**Component**Listener	**Component**Listener
ContainerEvent	Container	addContainerListener removeContainerListener	ContainerListener
FocusEvent	Component	addFocusListener removeFocusListener	FocusListener
KeyEvent	Component	addKeyListener removeKeyListener	KeyListener
MouseEvent	Component	add**Mouse**Listener remove**Mouse**Listener add**MouseMotion**Listener remove**MouseMotion**Listener	**Mouse**Listener **MouseMotion**Listener
WindowEvent	Window	addWindowListener removeWindowListener	WindowListener

Registering and Removing Listeners of Events (cont.)

Event	Source	Methods which the source provides to register and remove listeners who are interested in the event generated by the source.	Interface which a listener for a particular event must implement.
ActionEvent	Button List MenuItem TextField	add**Action**Listener remove**Action**Listener	**Action**Listener
AdjustmentEvent	Scrollbar	addAdjustmentListener removeAdjustmentListener	AdjustmentListener
ItemEvent	Choice Checkbox CheckboxMenuItem List	addItemListener removeItemListener	ItemListener
TextEvent	TextArea TextField	addTextListener removeTextListener	TextListener

Listener Interfaces

Listener Interfaces	Methods in listener interfaces

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

ContainerListener	componentAdded(ContainerEvent e)
	componentRemoved(ContainerEvent e)

FocusListener	focusGained(FocusEvent e)
	focusLost(FocusEvent e)

KeyListener	keyPressed(KeyEvent e)
	keyReleased(KeyEvent e)
	keyTyped(KeyEvent e)

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

Listener Interfaces (cont.)

Listener Interfaces	Methods in listener interfaces

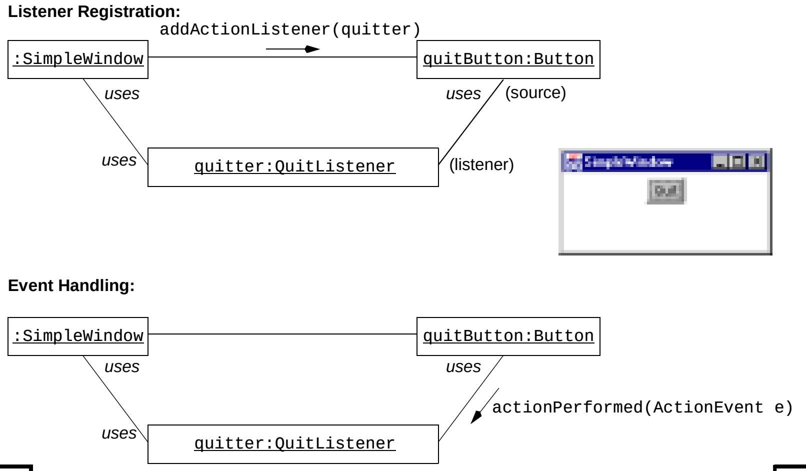
MouseMotionListener	mouseDragged(MouseEvent e)
	mouseMoved(MouseEvent e)

WindowListener	windowActivated(WindowEvent e)
	windowClosed(WindowEvent e)
	windowClosing(WindowEvent e)
	windowDeactivated(WindowEvent e)
	windowDeiconified(WindowEvent e)
	windowIconified(WindowEvent e)
	windowOpened(WindowEvent e)

ActionListener	actionPerformed(ActionEvent e)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent e)
ItemListener	itemStateChanged(ItemEvent e)
TextListener	textValueChanged(TextEvent e)

Example One: Simple Event Handling

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



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 QuitListener(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 `add`*`Action`*`Listener` method from the `Button` class to register the listener.

- Make sure that the listener implements the right `XListener` interface.

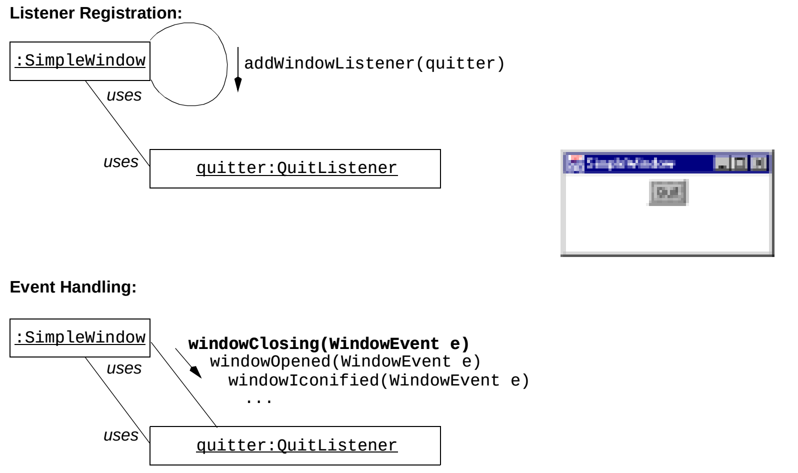
```
// Definition of the Listener
class QuitHandler implements ActionListener {                   // (3)

    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;                  // The source
    QuitHandler quitter;                // The listener

    public SimpleWindow() {

        // Create a window
        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.
        setSize(200,100);
        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.



- Adding the extra functionality.

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

class QuitHandler implements ActionListener, WindowListener {   // (4)
    // ...
    // 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 `QuitHandler` implements the interface `WindowListener` which has seven methods. Some methods are just *stubs*.
  - The listener `QuitHandler` receives notification of seven types of events typified by the methods of the `WindowListener` interface.

Example Two: Event Handling

```
/* 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
    QuitHandler quitter;             // The listener

    public SimpleWindowTwo() {

        // Create a window
        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)

        // Add the listener to the window
        addWindowListener(quitter);                          // (3)

        // Set the window size and pop it up.
        setSize(200,100);
        setVisible(true);
    }

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

}
```

```
// Definition of the Listener

class QuitHandler implements ActionListener, WindowListener {   // (4)

    private SimpleWindowTwo application; // The associated application

    public QuitHandler(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 quit button.
    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() == application.quitButton) {
            terminate();
        }
    }
```

```
    // 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) {}
}
```

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.

```
// Definition of the Listener
class QuitHandler extends WindowAdapter implements ActionListener { // (4)

    // ...

    // Overrides the appropriate interface method
    public void windowClosing(WindowEvent evt) {                // (6)
        terminate();
    }
}
```

Note that the `QuitHandler` *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.
  - Note however that an anonymous class can access `final` local variables, `final` method-parameters and `final` `catch`-block parameters in the scope of the local context.

Extending an existing class

`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.

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

```
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)
    }
}
```

- 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.

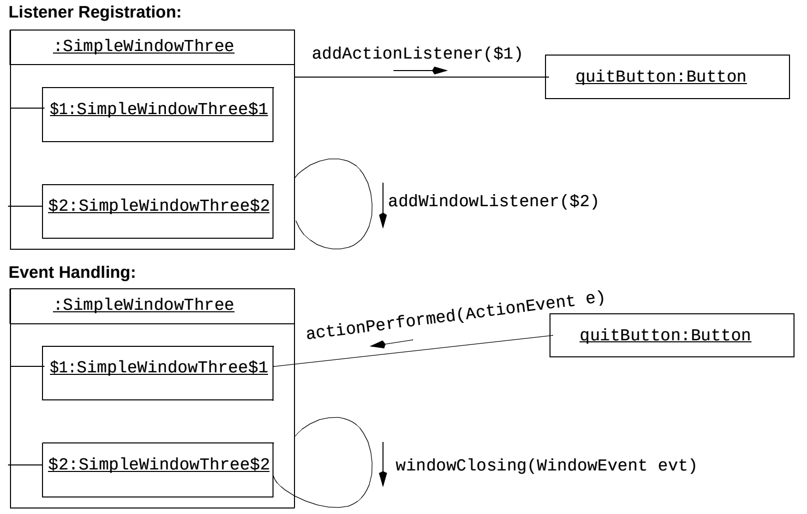
Implementing an interface

`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



Example: Listeners as Anonymous Classes

```
/*
   SimpleWindowThree: A simple setup for Event Delegation Model
   using Anonymous Classes.
 */
import java.awt.*;
import java.awt.event.*;

public class SimpleWindowThree extends Frame {

    Button quitButton;

    public SimpleWindowThree() {

        // Create a window
        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
            // Invoked when the user clicks the quit button.
            public void actionPerformed(ActionEvent evt) {
                if (evt.getSource() == quitButton)
                    terminate();                                 // (2)
            }
        });
        // Create and add the listener to the window
        addWindowListener(new WindowAdapter() {                  // (3) $2
            // Invoked when the user clicks the close-box.
            public void windowClosing(WindowEvent evt) {
                terminate();                                     // (4)
            }
        });
        // Set the window size and pop it up.
        setSize(200,100);
        setVisible(true);
    }
    private void terminate() {
        System.out.println("Quitting the application.");
        dispose();
        System.exit(0);
    }
    /** Create an instance of the application */
    public static void main(String args[]) { new SimpleWindowThree(); }
}
```

Programming Model for GUI-based Applications

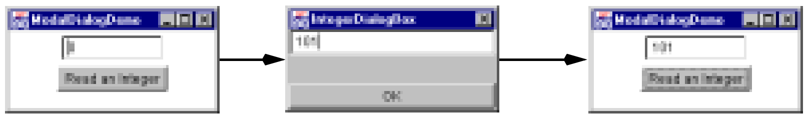
The programming model comprises of three parts:

1. Construction of the GUI: component hierarchy and layout.
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 component hierarchy.
- For the root window, decide a layout manager.
  - use the method `setLayout(aLayoutManger)`
- For each panel, decide a layout manager.
  - use the method `setLayout(someOtherLayoutManger)`
- 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 component 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 component hierarchy) visible.
  - use the method `setSize(width, height)`
  - use the method `setVisible(true)`

Example: Modal Dialog Boxes



(b) Input window to read an integer

(a) Main window before reading an integer

(c) Main window after reading an integer

- 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.

Example: Modal Dialog Boxes (cont.)

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

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");

        setLayout(new FlowLayout());
        add(intTF);
        add(intButton);

        intButton.addActionListener(this);
```

Implements listener interface.

Setup the main window.

Register window as listener with button.

Example: Modal Dialog Boxes (cont.)

```
        addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                terminate();
            }
        });

        setSize(200, 100);
        setVisible(true);
    }
    public void setInteger(String str) {
        intTF.setText(str);
    }
    public void terminate() {
        setVisible(false);
        dispose();
        System.exit(0);
    }
    public void actionPerformed(ActionEvent e) {
        new IntegerInputDialog(this);
    }

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

} // end class ModalDialogDemo
```

Register listener for closing the window.

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

Example: Dialog boxes (cont.)

```
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) {
                if (isLegalInteger(intTF)) {
                    app.setInteger(intTF.getText());
                    removeDialogBox();
                }
            }
        });
```

Constructor must have parent.

Setup for Dialog window.

Register listeners with sources.

Example: Dialog boxes (cont.)

```
        addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                removeDialogBox();
            }
        });

        setSize(200, 100);
        setVisible(true);
    }

    public boolean isLegalInteger(TextField tf) {
        try { Integer.parseInt(tf.getText()); }
        catch (NumberFormatException ex) { return false; }
        return true;
    }

    public void removeDialogBox() {
        setVisible(false);
        dispose();
    }
}
```

Register listener for closing the window.

Necessary for making the window visible.

Data Validation

Necessary to free the resources.