

CENG 1004

Introduction to Object Oriented Programming

Spring 2016

WEEK 13

GUI Basics

History of UI in Java

- When Java was first released in 1995 with **Abstract Windows Toolkit (AWT)** as GUI toolkit
 - platform-dependent
 - classes available under **java.awt.*** package
 - still exists, but not used much anymore

History of UI in Java

- **Swing** was introduced as part of the Java Standard Edition since release 1.2.
 - provides a richer set of UI widgets,
 - platform-independent (lightweight)
 - classes available under **javax.swing.*** package

History of UI in Java

- **JavaFX** is introduced with the aim of replacing Swing with the release of Java 8
 - UI is implemented in an XML file separately from implementing the application logic
 - Cascading Style Sheets (CSS) are used for formatting components
 - Better support for animation

GUI terminology


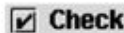



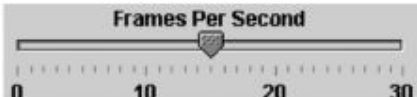



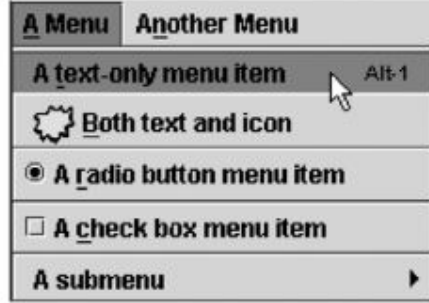
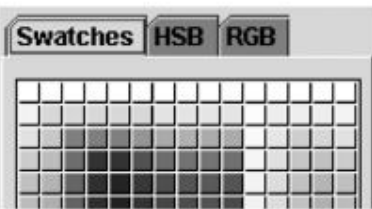



- **window:** A first-class citizen of the graphical desktop.
 - Also called a *top-level container*.
 - examples: frame, dialog box
- **component:** A GUI widget that resides in a window.
 - Also called *controls* in many other languages.
 - examples: button, text box, label
- **container:** A logical grouping for storing components.
 - examples: panel, box



Basic Workflow

- Create a window to hold your entire GUI
- Arrange UI components inside the window
 - Add individual components, or
 - Group components into containers and add them to the window
- Attach handler(s) to each UI component to handle the events they generate

Components

JButton 	JCheckBox 	JRadioButton 	JLabel  Text-Only Label
JTextField 	JSlider 	JToolBar 	
JComboBox 	JList 	JMenuBar, JMenu, JMenuItem 	
JColorChooser 	JFileChooser 	JTable 	JTree 

More Components

- **JFileChooser**: allows choosing a file
- **JLabel**: a simple text label
- **TextArea**: editable text
- **TextField**: editable text (one line)
- **Scrollbar**: a scrollbar
- **PopupMenu**: a pop-up menu
- **ProgressBar**: a progress bar
- Lots more!

Component properties

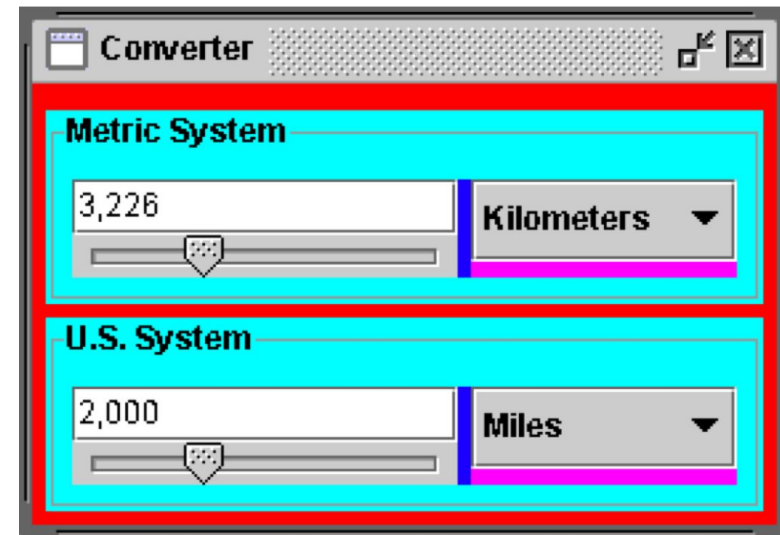
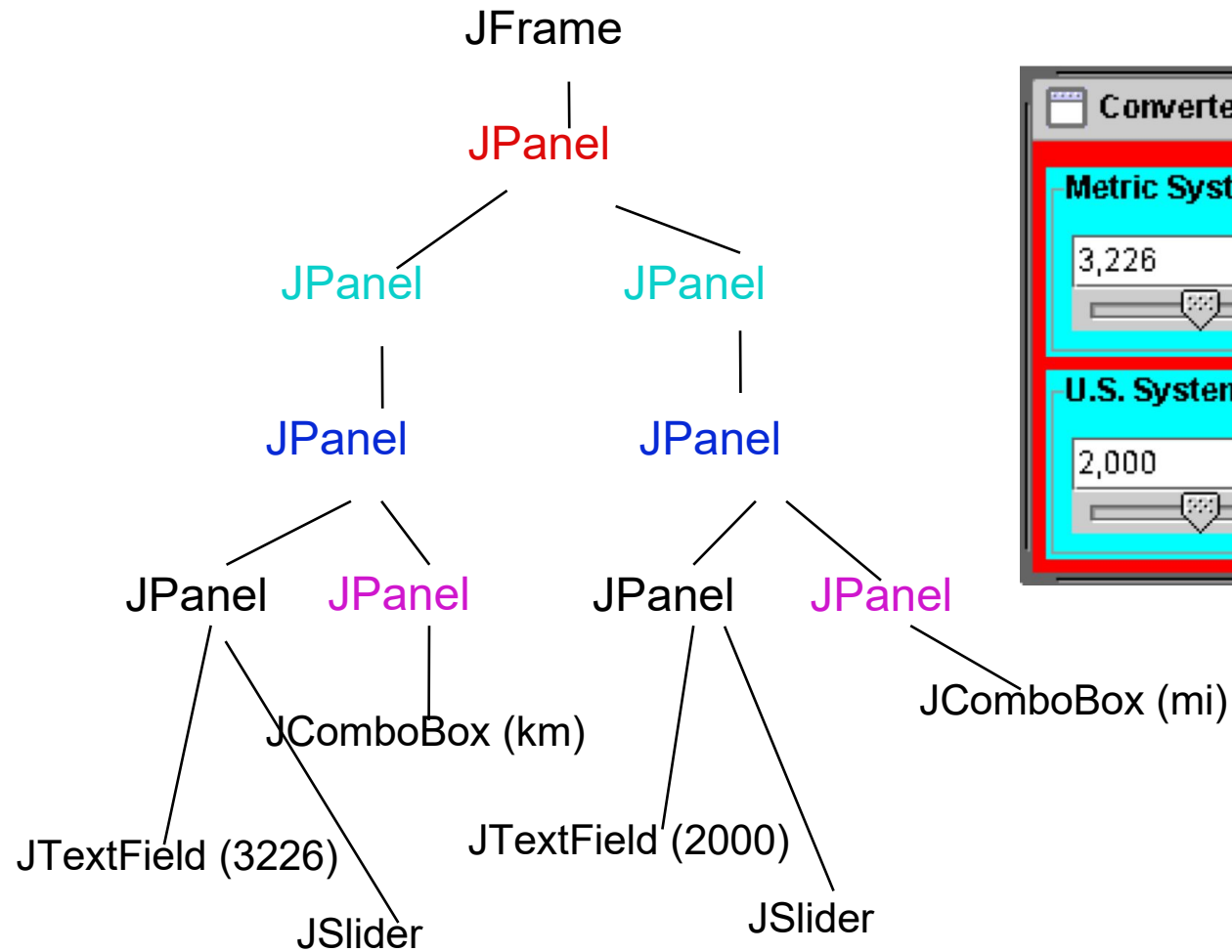
- Each has a `get` (or `is`) accessor and a `set` modifier method.
- examples: `getColor`, `setFont`, `setEnabled`, `isVisible`

name	type	description
background	<code>Color</code>	background color behind component
border	<code>Border</code>	border line around component
enabled	<code>boolean</code>	whether it can be interacted with
focusable	<code>boolean</code>	whether key text can be typed on it
font	<code>Font</code>	font used for text in component
foreground	<code>Color</code>	foreground color of component
height, width	<code>int</code>	component's current size in pixels
visible	<code>boolean</code>	whether component can be seen
tooltip text	<code>String</code>	text shown when hovering mouse
size, minimum / maximum / preferred size	<code>Dimension</code>	various sizes, size limits, or desired sizes that the component may take

Containers

- A container is a component that
 - Can hold other components
 - Has a layout manager
- Heavyweight vs. lightweight
 - A heavyweight component interacts directly with the host system
 - JWindow, JFrame, and JDialog are heavyweight
 - Except for these top-level containers, Swing components are almost all lightweight
 - JPanel is lightweight
- There are three basic *top-level* containers
 - **JWindow**: top-level window with no border
 - **JFrame**: top-level window with border and (optional) menu bar
 - **JDialog**: used for dialog windows
- Another important container
 - **JPanel**: used mostly to organize objects within other containers

A Component Tree



The Main Window

- The class `javax.swing.JFrame` is used as the outermost window in a Swing GUI

`java.lang.Object`

`java.awt.Component`

`java.awt.Container`

`java.awt.Window`

`java.awt.Frame`

`javax.swing.JFrame`

JFrame

a graphical window to hold other components



- `public JFrame()`
`public JFrame(String title)`
Creates a frame with an optional title.
 - Call `setVisible(true)` to make a frame appear on the screen after creating it.
- `public void add(Component comp)`
Places the given component or container inside the frame.

More JFrame

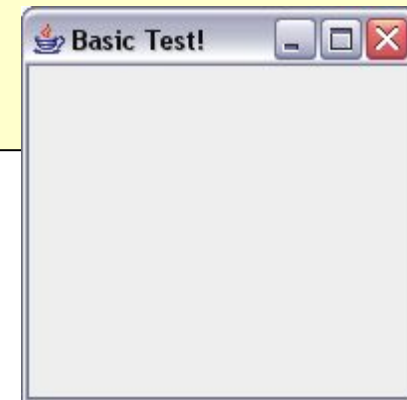


- `public void setDefaultCloseOperation(int op)`
Makes the frame perform the given action when it closes.
 - Common value passed: `JFrame.EXIT_ON_CLOSE`
 - If not set, the program will never exit even if the frame is closed.
- `public void setSize(int width, int height)`
Gives the frame a fixed size in pixels.
- `public void pack()`
Resizes the frame to fit the components inside it snugly.

Creating a Window in Swing

```
import javax.swing.*;

public class Basic1 {
    public static void main(String[] args) {
        //create the window
        JFrame f = new JFrame("Basic Test!");
        //quit Java after closing the window
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(200, 200); //set size in pixels
        f.setVisible(true); //show the window
    }
}
```

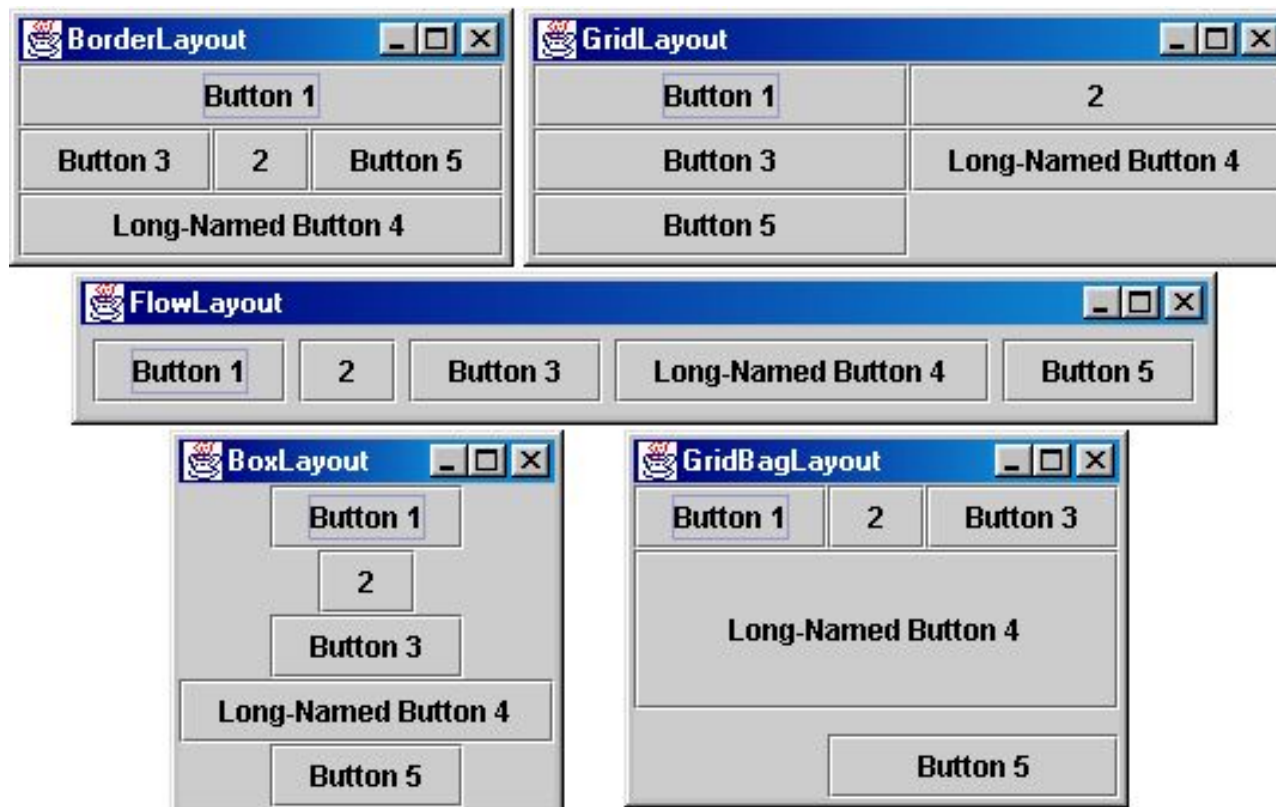


Layout

- Issue here concerns the way the components are placed on the screen
- If you do it statically (and you can), the resulting application can't be resized easily
- So GUI builders offer a more dynamic option

Containers and layout

- Place components in a container, add the container to a frame.
 - *container*: An object that stores components and governs their positions, sizes, and resizing behavior.



Sizing and positioning

How does the programmer specify where each component appears, how big each component should be, and what the component should do if the window is resized / moved / maximized / etc.?

- **Absolute positioning** (C++, C#, others):
Programmer specifies exact pixel coordinates of every component.
 - "Put this button at (x=15, y=75) and make it 70x31 px in size."
- **Layout managers** (Java):
Objects that decide where to position each component based on some general rules or criteria.
 - "Put these four buttons into a 2x2 grid and put these text boxes in a horizontal flow in the south part of the frame."

JFrame as container

A `JFrame` is a container. Containers have these methods:

- `public void add(Component comp)`
`public void add(Component comp, Object info)`
Adds a component to the container, possibly giving extra information about where to place it.
- `public void remove(Component comp)`
- `public void setLayout(LayoutManager mgr)`
Uses the given layout manager to position components.
- `public void validate()`
Refreshes the layout (if it changes after the container is onscreen).

Preferred sizes

- Swing component objects each have a certain size they would "like" to be: Just large enough to fit their contents (text, icons, etc.).
 - This is called the *preferred size* of the component.
 - Some types of layout managers (e.g. `FlowLayout`) choose to size the components inside them to the preferred size.
 - Others (e.g. `BorderLayout`, `GridLayout`) disregard the preferred size and use some other scheme to size the components.

Buttons at preferred size:



Not preferred size:



FlowLayout

```
public FlowLayout()
```

- treats container as a left-to-right, top-to-bottom "paragraph".
 - Components are given preferred size, horizontally and vertically.
 - Components are positioned in the order added.
 - If too long, components wrap around to the next line.

```
myFrame.setLayout(new FlowLayout());  
myFrame.add(new JButton("Button 1"));
```

- The default layout for containers other than `JFrame` (seen later).



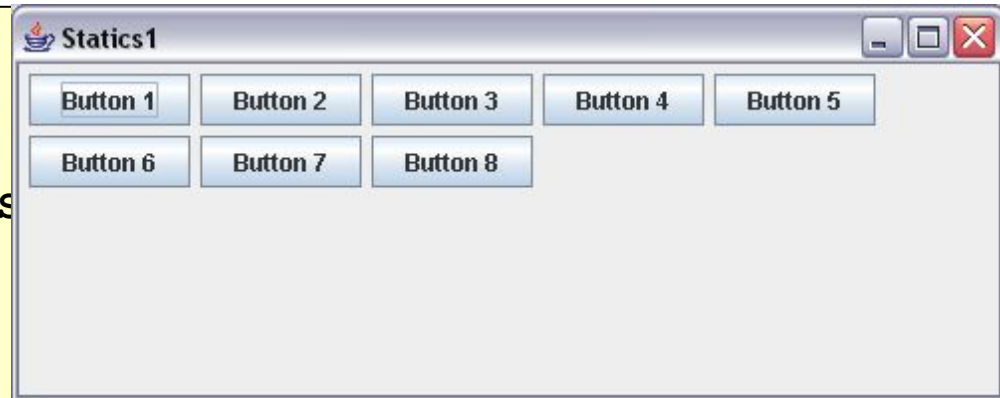
FlowLayout Example

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

public class Statics1 {
    public static void main(S
        new S1GUI();
    }
}

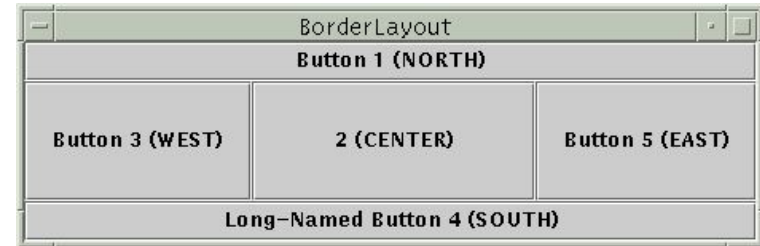
class S1GUI {
    private JFrame f;

    public S1GUI() {
        f = new JFrame("Statics1");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(500, 200);
        f.setLayout(new FlowLayout(FlowLayout.LEFT));
        for (int b = 1; b < 9; b++)
            f.add(new JButton("Button " + b));
        f.setVisible(true);
    }
}
```



BorderLayout

```
public BorderLayout()
```



- Divides container into five regions:
 - NORTH and SOUTH regions expand to fill region horizontally, and use the component's preferred size vertically.
 - WEST and EAST regions expand to fill region vertically, and use the component's preferred size horizontally.
 - CENTER uses all space not occupied by others.

```
myFrame.setLayout(new BorderLayout());  
myFrame.add(new JButton("Button 1"), BorderLayout.NORTH);
```

- This is the default layout for a JFrame.

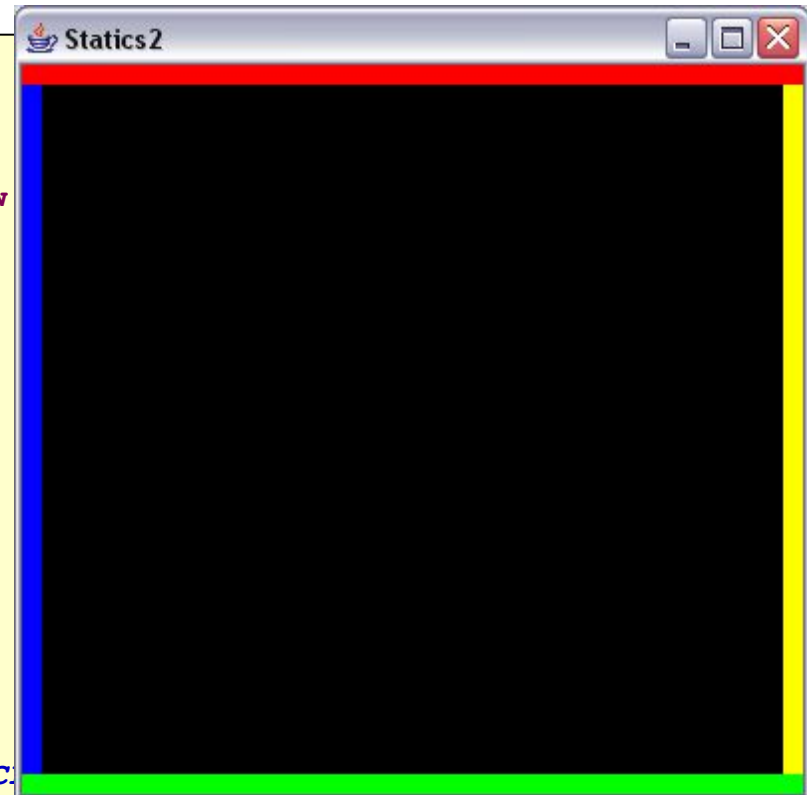
BorderLayout Example

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

public class Statics2 {
    public static void main(String[] args) { new

class ColoredJPanel extends JPanel {
    Color color;
    ColoredJPanel(Color color) {
        this.color = color;
    }
    public void paintComponent(Graphics g) {
        g.setColor(color);
        g.fillRect(0, 0, 400, 400);
    }
}

class S2GUI extends JFrame {
    public S2GUI() {
        setTitle("Statics2");
        setDefaultCloseOperation(JFrame.EXIT_ON_C
        setSize(400, 400);
        add(new ColoredJPanel(Color.RED), BorderLayout.NORTH);
        add(new ColoredJPanel(Color.GREEN), BorderLayout.SOUTH);
        add(new ColoredJPanel(Color.BLUE), BorderLayout.WEST);
        add(new ColoredJPanel(Color.YELLOW), BorderLayout.EAST);
        add(new ColoredJPanel(Color.BLACK), BorderLayout.CENTER);
        setVisible(true);
    }
}
```



GridLayout

```
public GridLayout(int rows, int columns)
```

- Treats container as a grid of equally-sized rows and columns.
- Components are given equal horizontal / vertical size, disregarding preferred size.
- Can specify 0 rows or columns to indicate expansion in that direction as needed.



Concurrency in Swing

- Careful use of concurrency is particularly important to the Swing programmer.
- A well-written Swing program uses concurrency to create a user interface that never "freezes" — the program is always responsive to user interaction, no matter what it's doing.
- To create a responsive program, the programmer must learn how the Swing framework employs threads.

Concurrency in Swing

- A Swing programmer deals with the following kinds of threads:
 - **Initial threads**, the threads that execute initial application code.
 - **The event dispatch thread**, where all event-handling code is executed. Most code that interacts with the Swing framework must also execute on this thread.
 - **Worker threads**, also known as background threads, where time-consuming background tasks are executed.

Initial Threads

- Every program has a set of threads where the application logic begins.
- In standard programs, there's only one such thread: the thread that invokes the **main method** of the program class.

Initial Threads

- In Swing programs, the initial threads don't have a lot to do. Their most essential job is to **create a Runnable** object that initializes the GUI and schedule that object for execution on the **event dispatch thread**.
- An initial thread schedules the GUI creation task by invoking `javax.swing.SwingUtilities.invokeLater` or `javax.swing.SwingUtilities.invokeAndWait`.

Initial Threads

```
SwingUtilities.invokeLater(new Runnable() {  
    public void run() {  
        createAndShowGUI();  
    }  
});
```

or

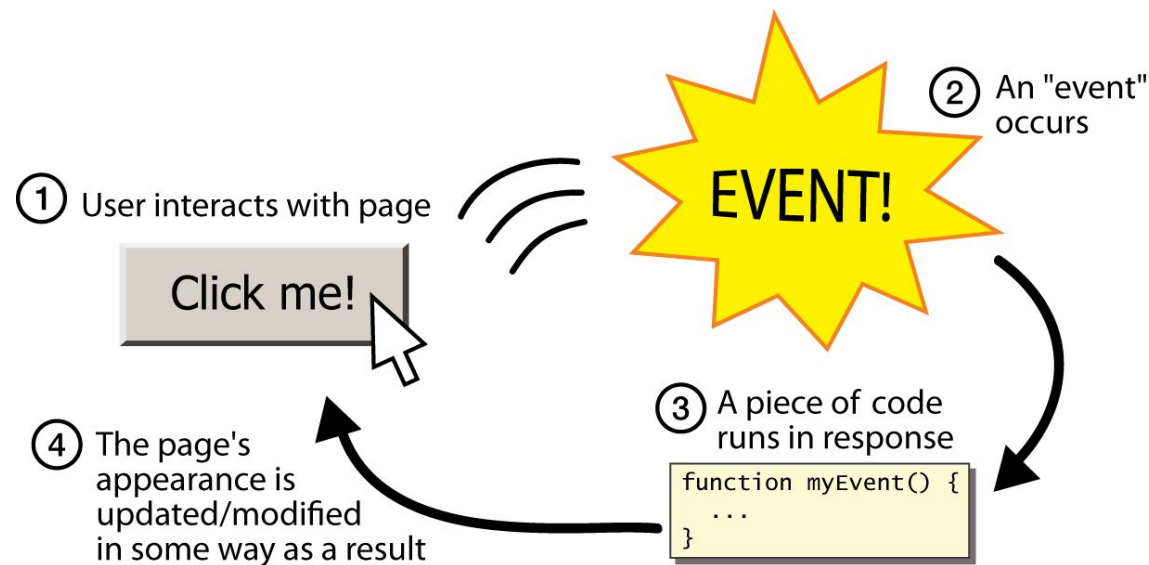
```
EventQueue.invokeLater(new Runnable() {  
    public void run() {  
        createAndShowGUI();  
    }  
});
```

The Event Dispatch Thread

- Swing event handling code runs on a special thread known as the **event dispatch thread**.
- Most code that invokes Swing methods also runs on this thread.
- This is necessary because most Swing object methods **are not "thread safe"**: invoking them from multiple threads risks thread interference or memory consistency errors.

Graphical events

- **event:** An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components.
- **listener:** An object that waits for events and responds to them.
 - To handle an event, attach a *listener* to a component.
 - The listener will be notified when the event occurs (e.g. button click).



GUI event example

```
public class MyGUI {
    private JFrame frame;
    private JButton btn1;
    private JTextField textfield;

    public MyGUI() {
        ...
        btn1.addActionListener(new Btn1Listener() );
    }
    ...

    // When button is clicked, doubles the field's text.
    private class Btn1Listener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            String text = textfield.getText();
            textfield.setText(text + text);
        }
    }
}
```

Procedural vs. Event-Driven Programming

- *Procedural programming* is executed in procedural order.
- In event-driven programming, code is executed upon activation of events.

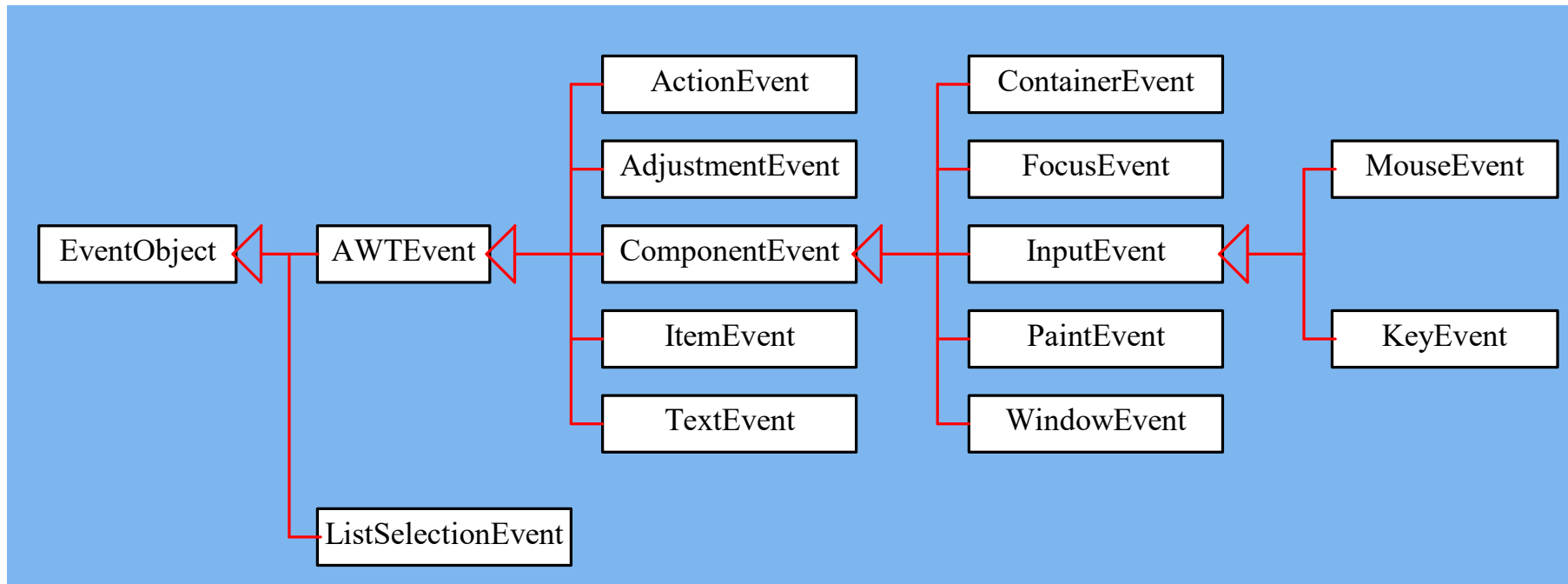
Event-driven Programming

- A programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads.
- Event-driven programming is the dominant paradigm used in graphical user interfaces

Events

- An *event* can be defined as a type of signal to the program that something has happened.
- The event is generated by
 - external user actions such as mouse movements, mouse clicks, and keystrokes, or
 - by the operating system, such as a timer.

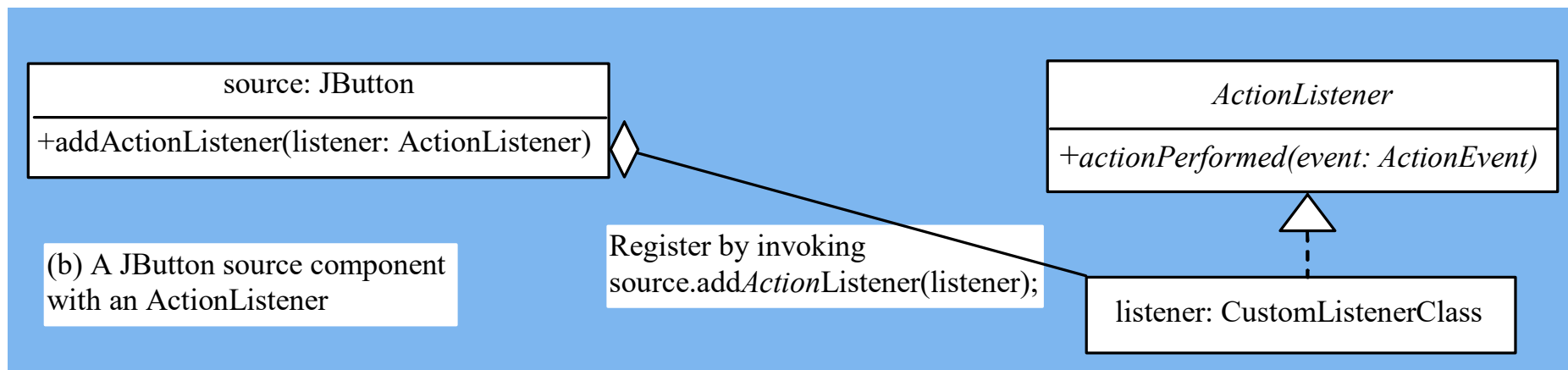
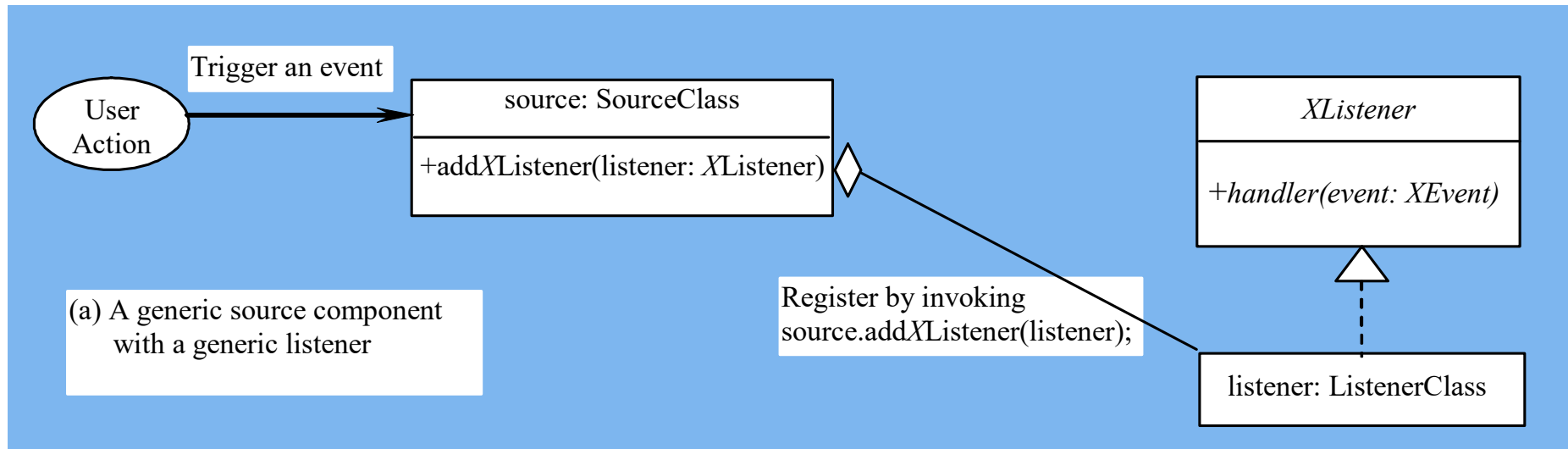
Event Classes



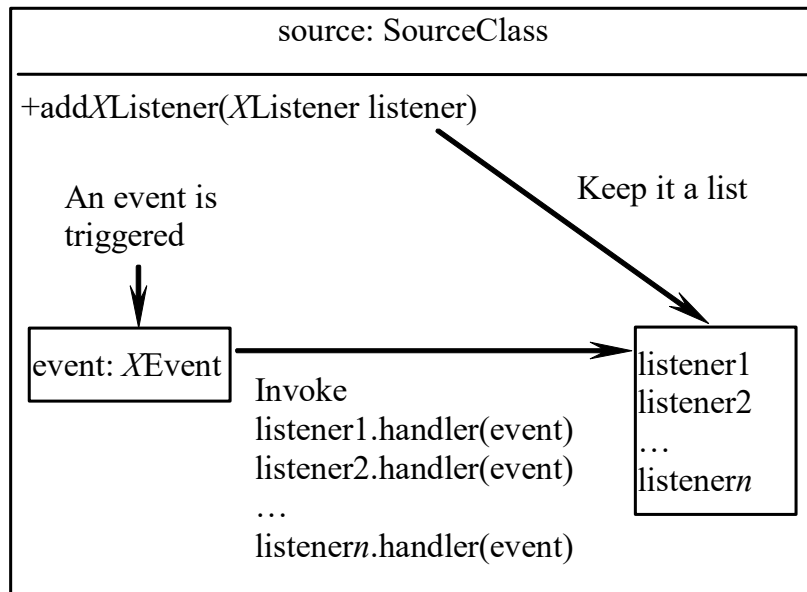
Event Information

- You can identify the source object of the event using the getSource() instance method in the EventObject class.
- The subclasses of EventObject deal with special types of events, such as button actions, window events, component events, mouse movements, and keystrokes.

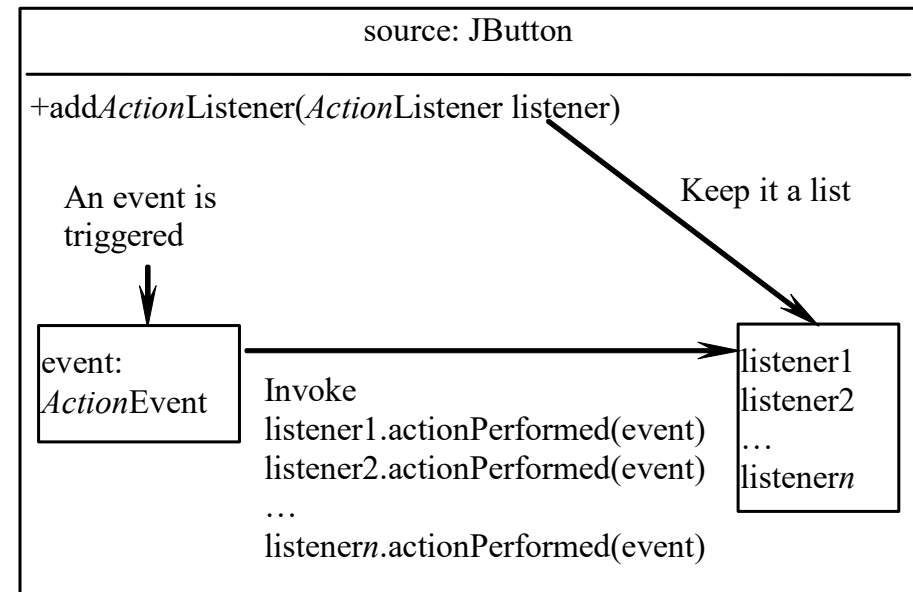
The Delegation Model



Internal Function of a Source Component



(a) Internal function of a generic source object

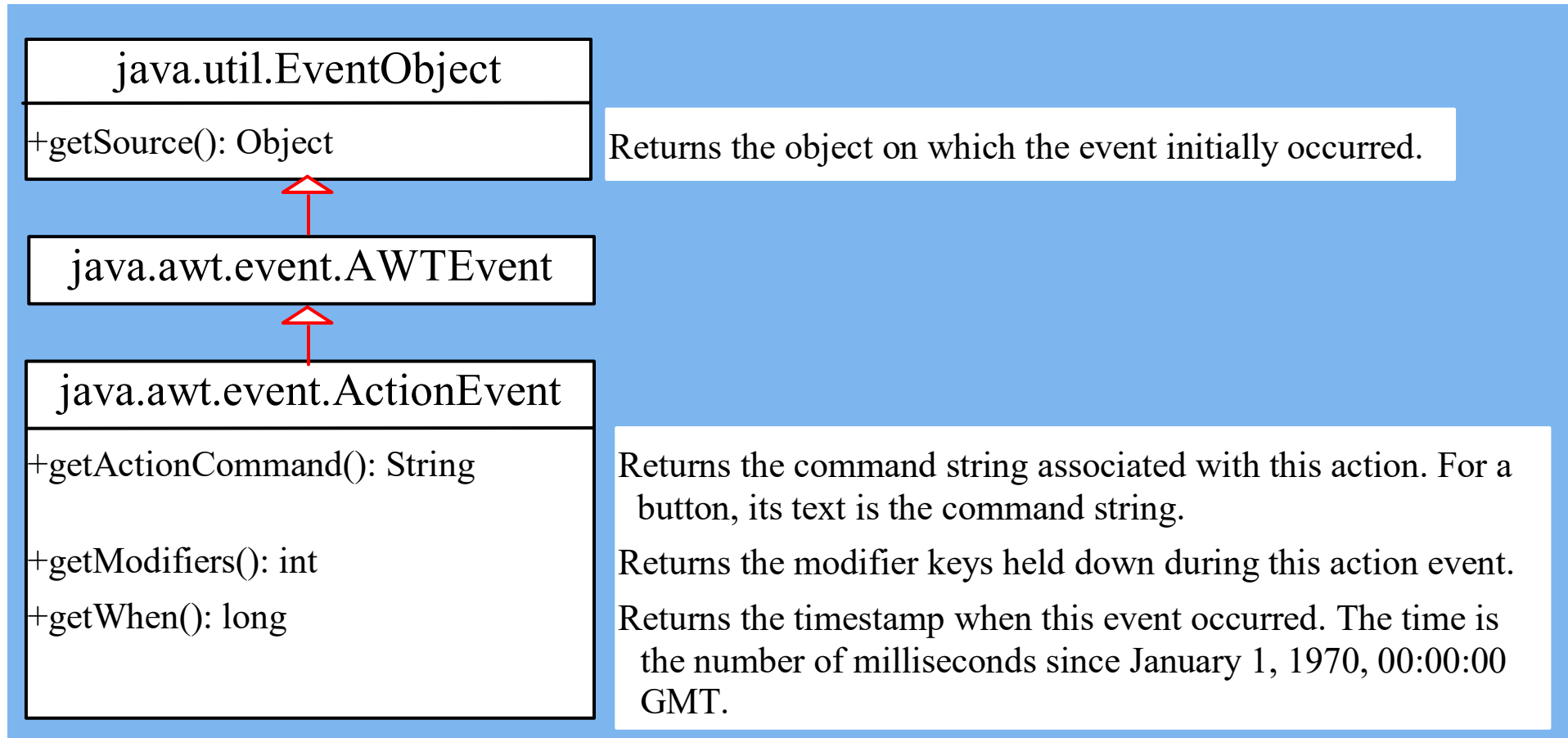


(b) Internal function of a JButton object

The Delegation Model: Example

```
JButton jbt = new JButton("OK");  
ActionListener listener = new OKListener();  
jbt.addActionListener(listener);
```

java.awt.event.ActionEvent



java.awt.event.ActionEvent

- `getActionCommand`
 - Returns the string associated with this action. Generally, this is a label on the component -- or on the selected item within the component -- that generated the action.
- `getModifiers`
 - Returns an integer representing the modifier keys the user was pressing when the action event occurred.
 - You can use the constants `ActionEvent.SHIFT_MASK`, `ActionEvent.CTRL_MASK`, `ActionEvent.META_MASK`, and `ActionEvent.ALT_MASK` to determine which key was pressed.
 - For example, if the user Shift-selects a menu item, then the following expression is nonzero:

```
actionEvent.getModifiers() & ActionEvent.SHIFT_MASK
```

Inner Class Listeners

- A listener class is designed specifically to create a listener object for a GUI component (e.g., a button).
- It will not be shared by other applications. So, it is appropriate to define the listener class inside the frame class as an inner class.

Inner Classes

- Inner class: A class is a member of another class.
- An inner class can reference the data and methods defined in the outer class in which it nests, so you do not need to pass the reference of the outer class to the constructor of the inner class.

Inner Classes (cont.)

- Inner classes can make programs simple and concise.
- An inner class supports the work of its containing outer class and is compiled into a class named *OuterClassName\$InnerClassName.class*.
- For example, the inner class InnerClass in OuterClass is compiled into *OuterClass\$InnerClass.class*.

Anonymous Inner Classes (cont.)

Inner class listeners can be shortened using anonymous inner classes. An *anonymous inner class* is an inner class without a name. It combines declaring an inner class and creating an instance of the class in one step. An anonymous inner class is declared as follows:

```
new SuperClassName/InterfaceName() {  
    // Implement or override methods in superclass or interface  
    // Other methods if necessary  
}
```

Run

Alternative Ways of Defining Listener Classes

- There are many other ways to define the listener classes. For example
 - creating just one listener, register the listener with the buttons, and let the listener detect the event source, i.e., which button fires the event.

Alternative Ways of Defining Listener Classes

- You may also define outer class that implements ActionListener.

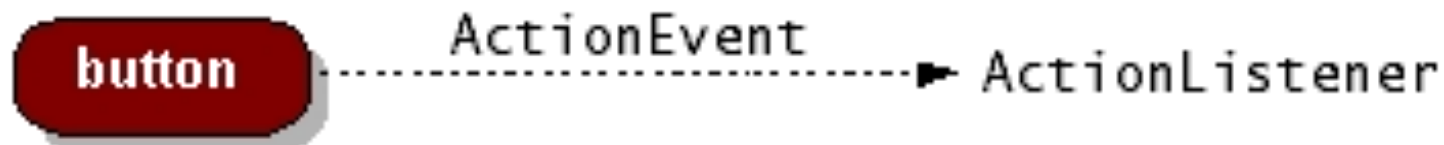
Event hierarchy

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

- EventObject
 - AWTEvent (AWT)
 - **ActionEvent**
 - TextEvent
 - ComponentEvent
 - FocusEvent
 - WindowEvent
 - InputEvent
 - KeyEvent
 - MouseEvent
- EventListener
 - AWTEventListener
 - **ActionListener**
 - TextListener
 - ComponentListener
 - FocusListener
 - WindowListener
 - KeyListener
 - MouseListener

Action events

- **action event:** An action that has occurred on a GUI component.
 - The most common, general event type in Swing. Caused by:
 - button or menu clicks,
 - check box checking / unchecking,
 - pressing Enter in a text field, ...
 - Represented by a class named `ActionEvent`
 - Handled by objects that implement interface `ActionListener`



Implementing a listener

```
public class name implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        code to handle the event;  
    }  
}
```

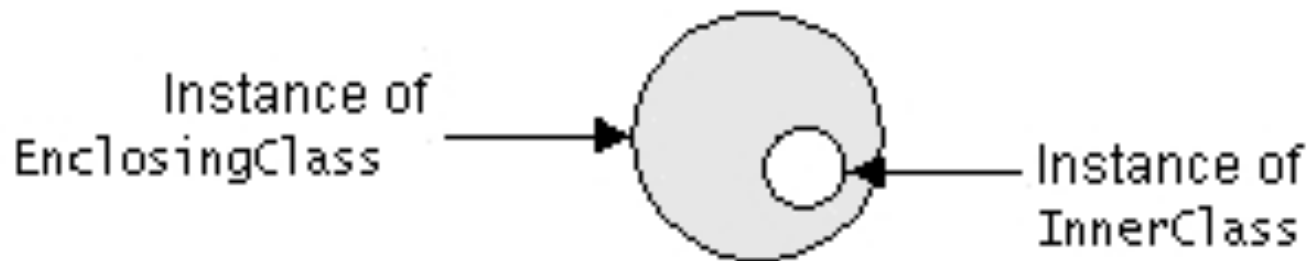
- JButton and other graphical components have this method:

- public void addActionListener(ActionListener al)

- Attaches the given listener to be notified of clicks and events that occur on this component.

Nested classes

- **nested class:** A class defined inside of another class.
- Usefulness:
 - Nested classes are hidden from other classes (encapsulated).
 - Nested objects can access/modify the fields of their outer object.
- Event listeners are often defined as nested classes inside a GUI.



Nested class syntax

```
// enclosing outer class
public class name {
    ...

    // nested inner class
    private class name {
        ...
    }
}
```

- Only the outer class can see the nested class or make objects of it.
- Each nested object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
 - If necessary, can refer to outer object as **OuterClassName.this**

Static inner classes

```
// enclosing outer class
public class name {
    ...

    // non-nested static inner class
    public static class name {
        ...
    }
}
```

- Static inner classes are *not* associated with a particular outer object.
- They cannot see the fields of the enclosing class.
- *Usefulness:* Clients can refer to and instantiate static inner classes:
Outer.Inner name = new **Outer.Inner**(**params**) ;

MouseEvent

java.awt.event.InputEvent

+getWhen(): long
+isAltDown(): boolean
+isControlDown(): boolean
+isMetaDown(): boolean
+isShiftDown(): boolean

Returns the timestamp when this event occurred.

Returns whether or not the Alt modifier is down on this event.

Returns whether or not the Control modifier is down on this event.

Returns whether or not the Meta modifier is down on this event

Returns whether or not the Shift modifier is down on this event.

java.awt.event.MouseEvent

+getButton(): int
+getClickCount(): int
+getPoint(): java.awt.Point
+getX(): int
+getY(): int

Indicates which mouse button has been clicked.

Returns the number of mouse clicks associated with this event.

Returns a Point object containing the x and y coordinates.

Returns the x-coordinate of the mouse point.

Returns the y-coordinate of the mouse point.

Handling Mouse Events

- Java provides two listener interfaces, `MouseListener` and `MouseMotionListener`, to handle mouse events.
- The `MouseListener` listens for actions such as when the mouse is pressed, released, entered, exited, or clicked.
- The `MouseMotionListener` listens for actions such as dragging or moving the mouse.

Handling Mouse Events

java.awt.event.MouseListener

+mousePressed(e: MouseEvent): void

Invoked when the mouse button has been pressed on the source component.

+mouseReleased(e: MouseEvent): void

Invoked when the mouse button has been released on the source component.

+mouseClicked(e: MouseEvent): void

Invoked when the mouse button has been clicked (pressed and released) on the source component.

+mouseEntered(e: MouseEvent): void

Invoked when the mouse enters the source component.

+mouseExited(e: MouseEvent): void

Invoked when the mouse exits the source component.

java.awt.event.MouseMotionListener

+mouseDragged(e: MouseEvent): void

Invoked when a mouse button is moved with a button pressed.

+mouseMoved(e: MouseEvent): void

Invoked when a mouse button is moved without a button pressed.

Example: Moving Message Using Mouse

Objective: Create a program to display a message in a panel. You can use the mouse to move the message. The message moves as the mouse drags and is always displayed at the mouse point.



Handling Keyboard Events

To process a keyboard event, use the following handlers in the `KeyListener` interface:

- `keyPressed(KeyEvent e)`

Called when a key is pressed.

- `keyReleased(KeyEvent e)`

Called when a key is released.

- `keyTyped(KeyEvent e)`

Called when a key is pressed and then released.

References

- <http://www.javatpoint.com/java-awt>
- <https://courses.cs.washington.edu/courses/cse331/11sp/lectures/slides/14a-gui.ppt>
- <http://www.cs.cornell.edu/Courses/cs2110/2013sp/L11-GUI%20Statics/L11cs2110sp13.ppt>