GUI and Event-Driven Programming

Dasar – Dasar Pemrograman 2

Slide Acknowledgment: Y. Daniel Liang

Pudy Prima

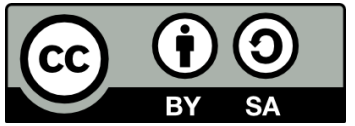
GUI Basic

Motivations

The design of the API for Java GUI programming is an excellent example of how the object-oriented principle is applied. In the chapters that follow, you will learn the framework of Java GUI API and use the GUI components to develop user-friendly interfaces for applications and applets.

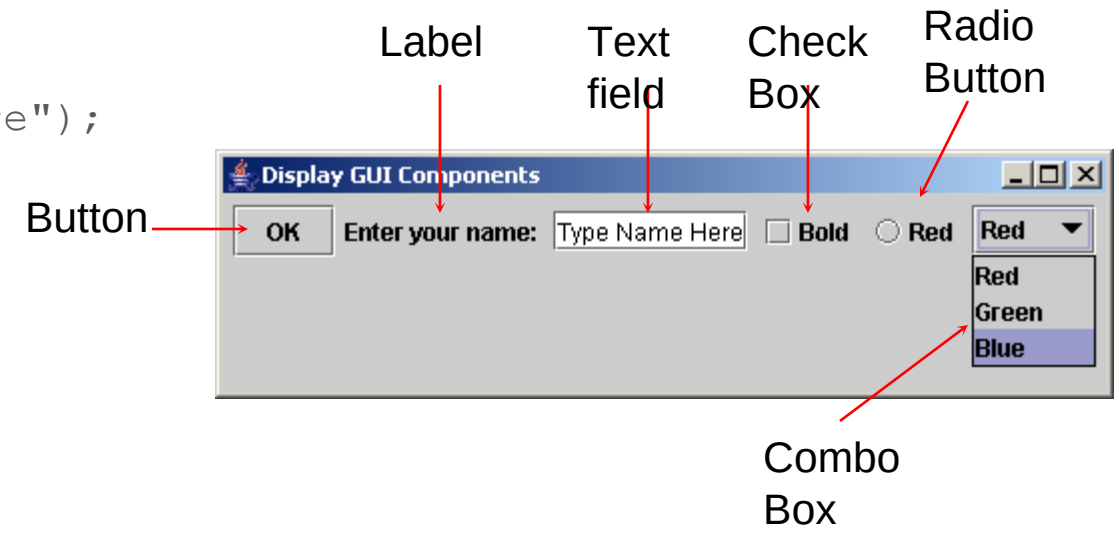
Objectives

- To distinguish between Swing and AWT (§12.2).
- To describe the Java GUI API hierarchy (§12.3).
- To create user interfaces using frames, panels, and simple GUI components (§12.4).
- To understand the role of layout managers and use the **FlowLayout**, **GridLayout**, and **BorderLayout** managers to lay out components in a container (§12.5).
- To use **JPanel** to group components in a subcontainer (§12.6).
- To create objects for colors using the **Color** class (§12.7).
- To create objects for fonts using the **Font** class (§12.8).
- To apply common features such as borders, tool tips, fonts, and colors on Swing components (§12.9).
- To decorate the border of GUI components (§12.9).
- To create image icons using the **ImageIcon** class (§12.10).
- To create and use buttons using the **JButton** class (§12.11).
- To create and use check boxes using the **JCheckBox** class (§12.12).
- To create and use radio buttons using the **JRadioButton** class (§12.13).
- To create and use labels using the **JLabel** class (§12.14).
- To create and use text fields using the **JTextField** class (§12.15).



Creating GUI Objects

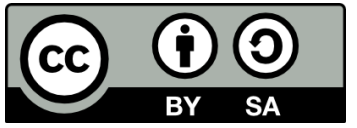
```
// Create a button with text OK
JButton jbtOK = new JButton("OK");

// Create a label with text "Enter your name: "
JLabel jlblName = new JLabel("Enter your name: ");

// Create a text field with text "Type Name Here"
JTextField jtfName = new JTextField("Type Name Here");

// Create a check box with text bold
JCheckBox jchkBold = new JCheckBox("Bold");

// Create a radio button with text red
JRadioButton jrbRed = new JRadioButton("Red");

// Create a combo box with choices red, green, and blue
JComboBox jcboColor = new JComboBox(new String[]{"Red",
  "Green", "Blue"});
```



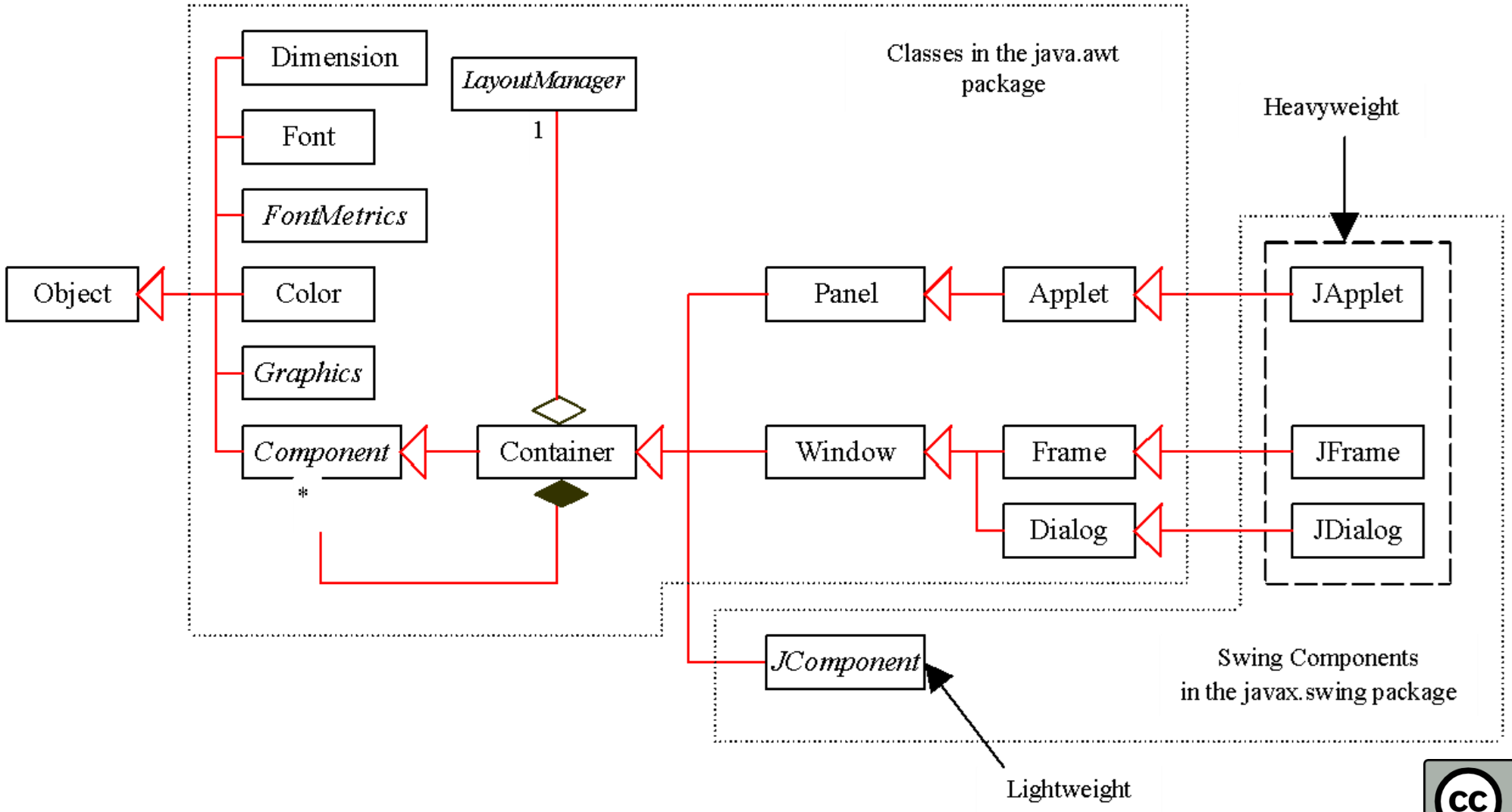
Swing vs. AWT

So why do the GUI component classes have a prefix *J*? Instead of JButton, why not name it simply Button? In fact, there is a class already named Button in the java.awt package.

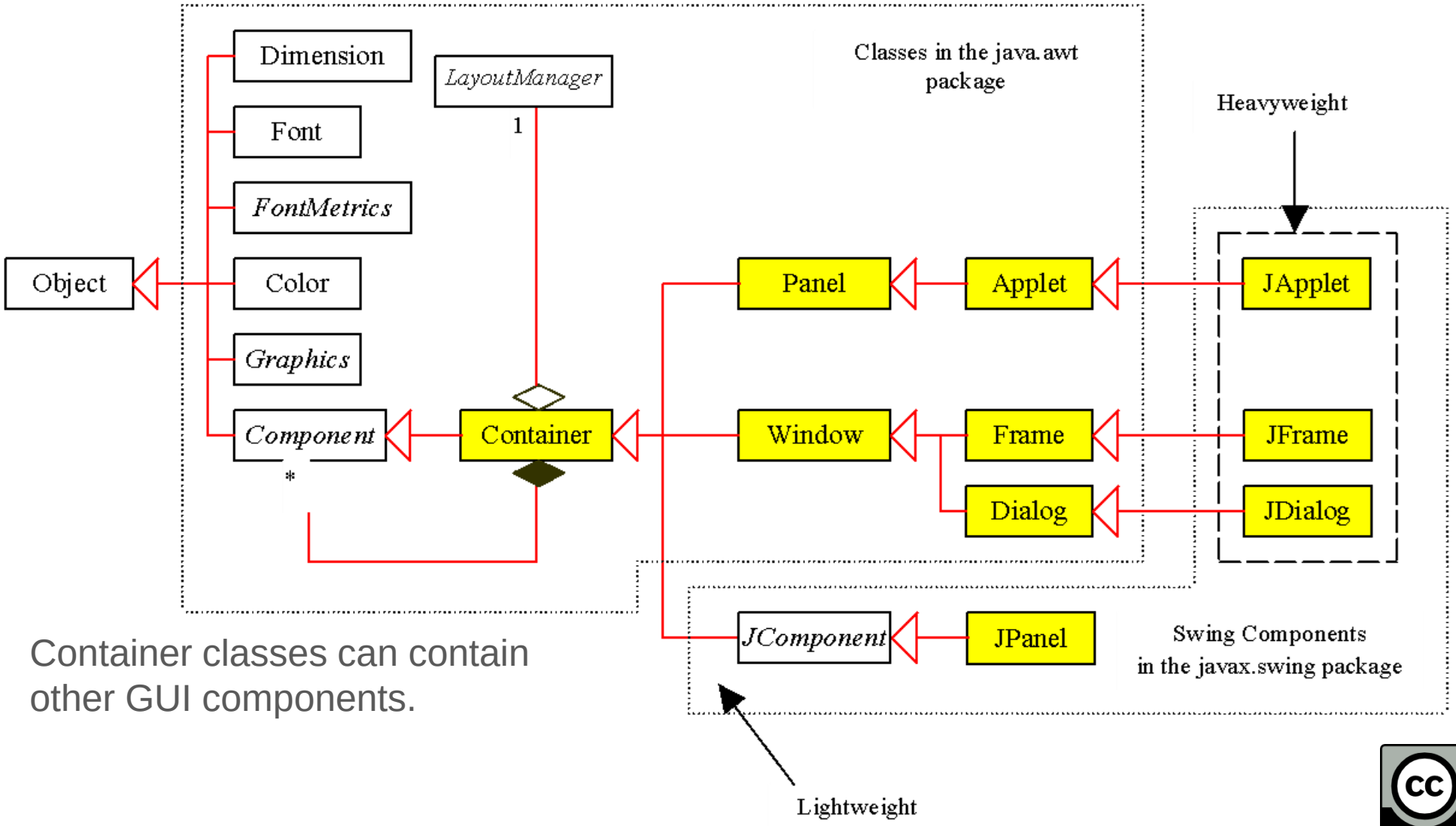
When Java was introduced, the GUI classes were bundled in a library known as the Abstract Windows Toolkit (AWT). For every platform on which Java runs, the AWT components are automatically mapped to the platform-specific components through their respective agents, known as *peers*. AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects. Besides, AWT is prone to platform-specific bugs because its peer-based approach relies heavily on the underlying platform. With the release of Java 2, the AWT user-interface components were replaced by a more robust, versatile, and flexible library known as *Swing components*. Swing components are painted directly on canvases using Java code, except for components that are subclasses of java.awt.Window or java.awt.Panel, which must be drawn using native GUI on a specific platform. Swing components are less dependent on the target platform and use less of the native GUI resource. For this reason, Swing components that don't rely on native GUI are referred to as *lightweight components,* and AWT components are referred to as *heavyweight components*.



GUI Class Hierarchy (Swing)

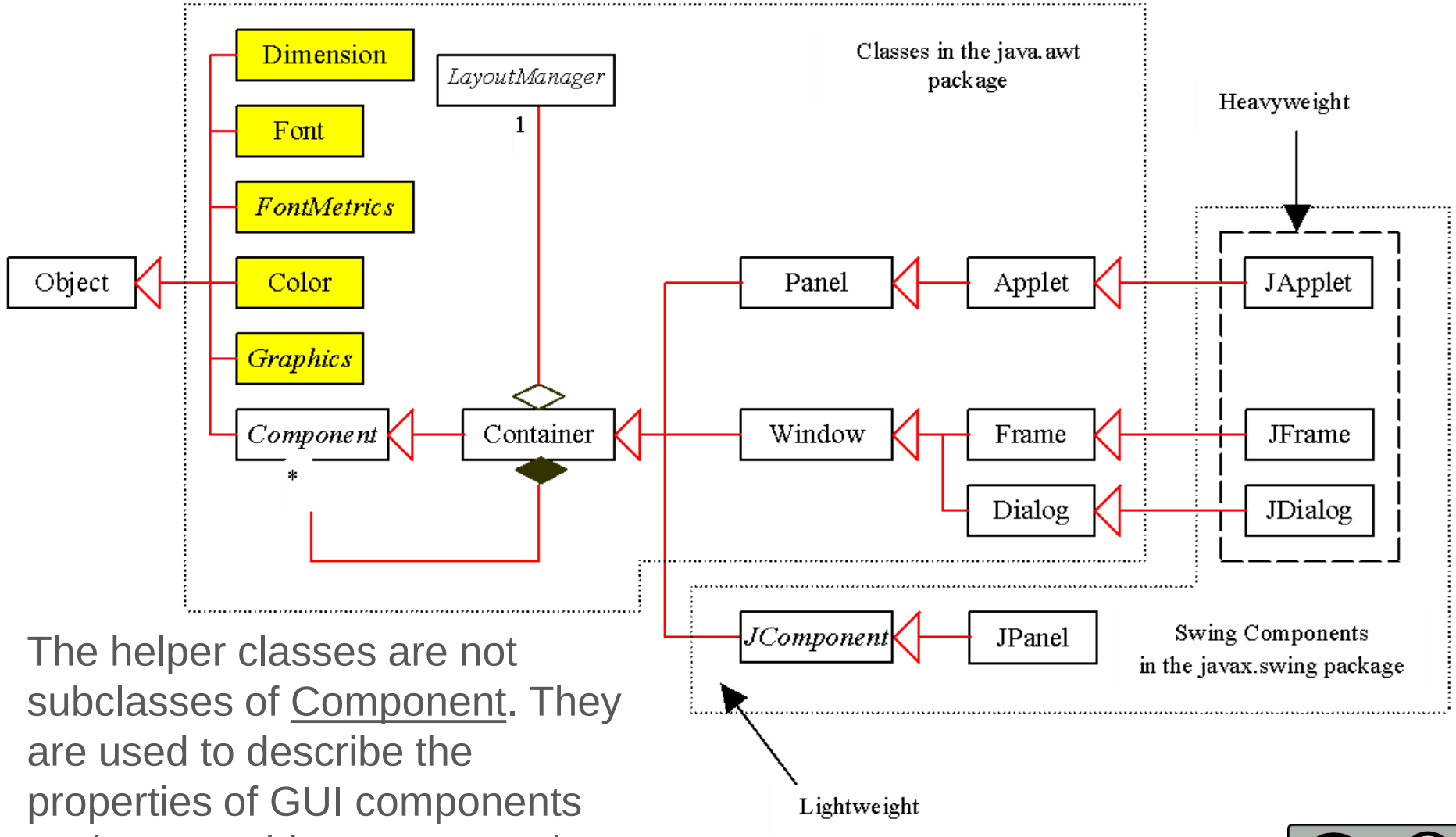


Container Classes



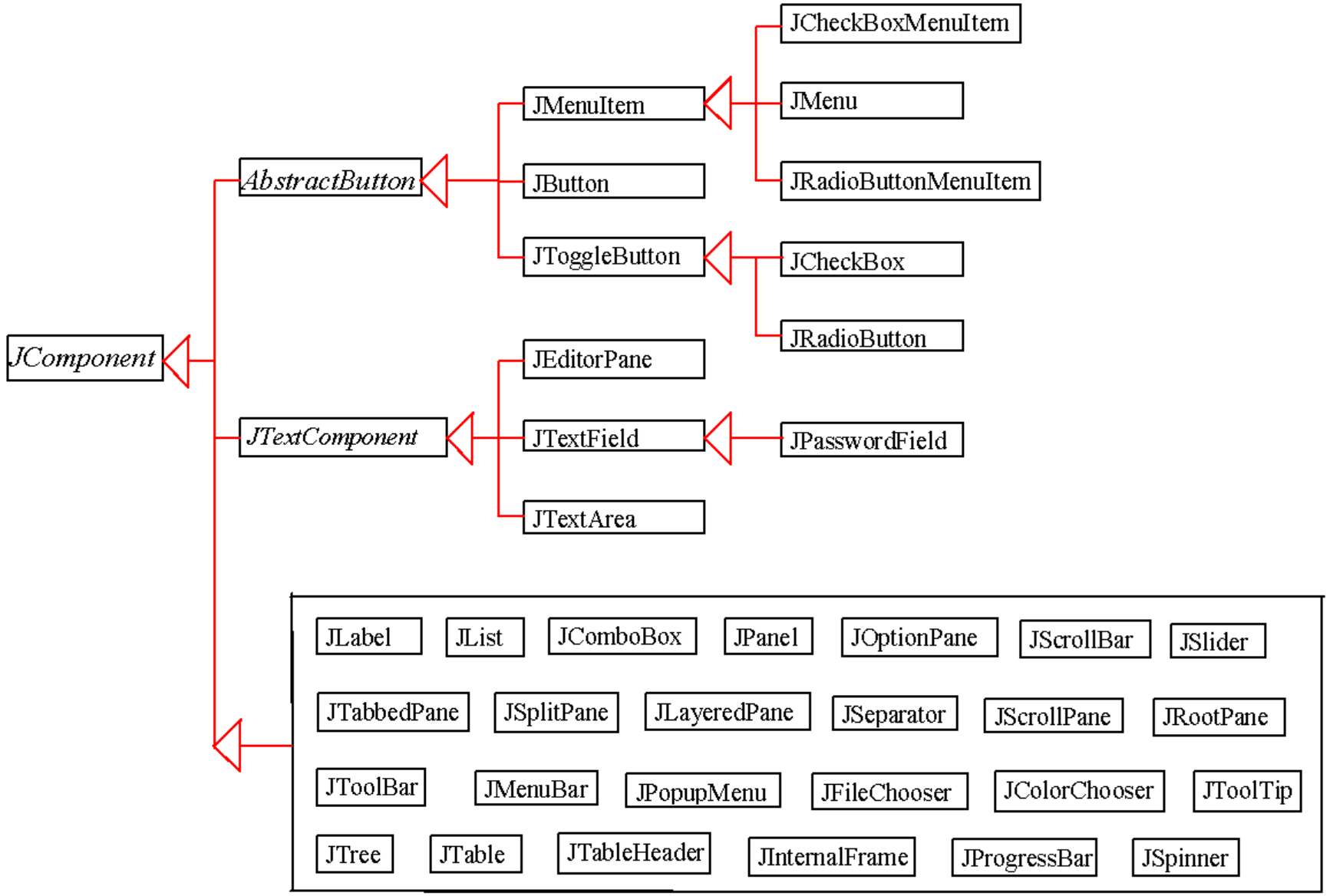
Container classes can contain other GUI components.

GUI Helper Classes



The helper classes are not subclasses of Component. They are used to describe the properties of GUI components such as graphics context, colors, fonts, and dimension.

Swing GUI Components

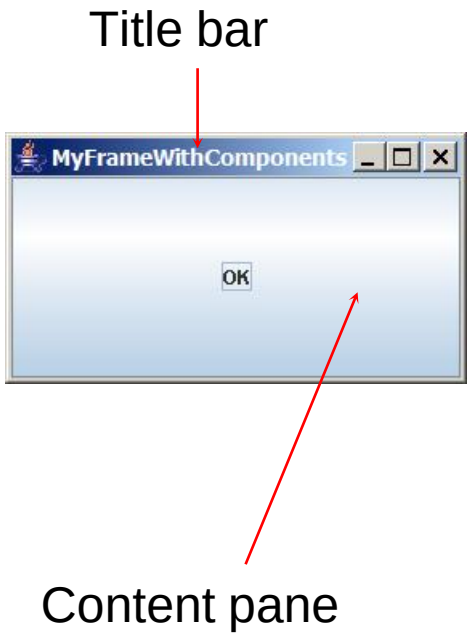


Frames

- Frame is a window that is not contained inside another window. Frame is the basis to contain other user interface components in Java GUI applications.
- The JFrame class can be used to create windows.
- For Swing GUI programs, use JFrame class to create widows.
- Creating frames

```
import javax.swing.*;
public class MyFrame {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Test Frame");
    frame.setSize(400, 300);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```


Adding Components into a Frame




```
// Add a button into the frame
frame.getContentPane().add(
   new JButton("OK"));
```

```
// Add a button into the frame
frame.add(
   new JButton("OK"));
```

Sampai dengan JDK 1.5, penambahan component dilakukan terhadap content pane pada container.

Setelah JDK 1.5, penambahan component dapat dilakukan langsung terhadap containernya.

JFrame Class

javax.swing.JFrame	
+JFrame()	Creates a default frame with no title.
+JFrame(title: String)	Creates a frame with the specified title.
+setSize(width: int, height: int): void	Specifies the size of the frame.
+setLocation(x: int, y: int): void	Specifies the upper-left corner location of the frame.
+setVisible(visible: boolean): void	Sets true to display the frame.
+setDefaultCloseOperation(mode: int): void	Specifies the operation when the frame is closed.
+setLocationRelativeTo(c: Component): void	Sets the location of the frame relative to the specified component. If the component is null, the frame is centered on the screen.
+pack(): void	Automatically sets the frame size to hold the components in the frame.

Layout Managers

- Java’s layout managers provide a level of abstraction to automatically map your user interface on all window systems.
- The UI components are placed in containers. Each container has a layout manager to arrange the UI components within the container.
- Layout managers are set in containers using the setLayout(LayoutManager) method in a container.

Kinds of Layout Managers

- FlowLayout
- GridLayout
- BorderLayout
- Null layout
- BoxLayout
- CardLayout
- OverlayLayout
- etc.

FlowLayout

The components are arranged in the container from left to right in the order in which they were added.

java.awt.FlowLayout
-alignment: int -hgap: int -vgap: int
+FlowLayout() +FlowLayout(alignment: int) +FlowLayout(alignment: int, hgap: int, vgap: int)

The `get` and `set` methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The alignment of this layout manager (default: `CENTER`).

The horizontal gap between the components (default: 5 pixels).

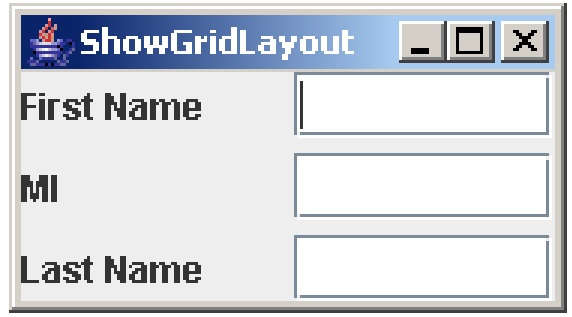
The vertical gap between the components (default: 5 pixels).

Creates a default `FlowLayout` manager.

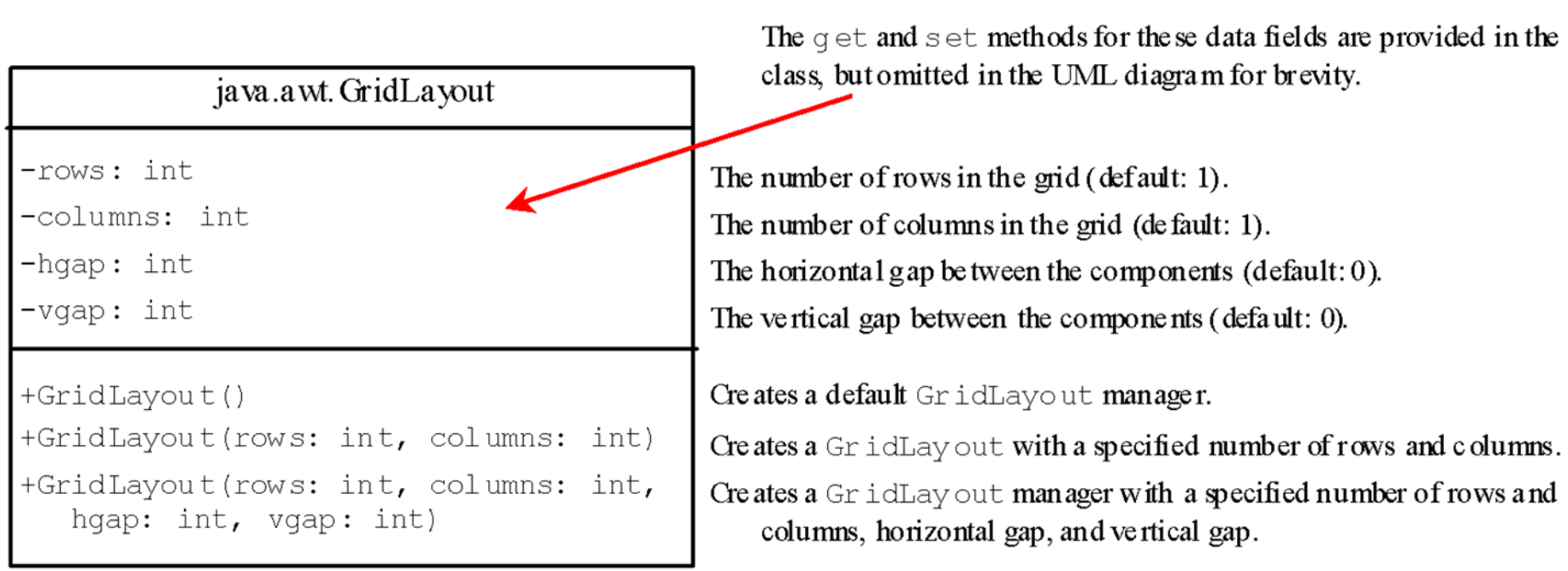
Creates a `FlowLayout` manager with a specified alignment.

Creates a `FlowLayout` manager with a specified alignment, horizontal gap, and vertical gap.

GridLayout



The components are arranged in a grid (matrix) formation from left to right, starting with the first row, then the second, and so on, in order in which they were added.



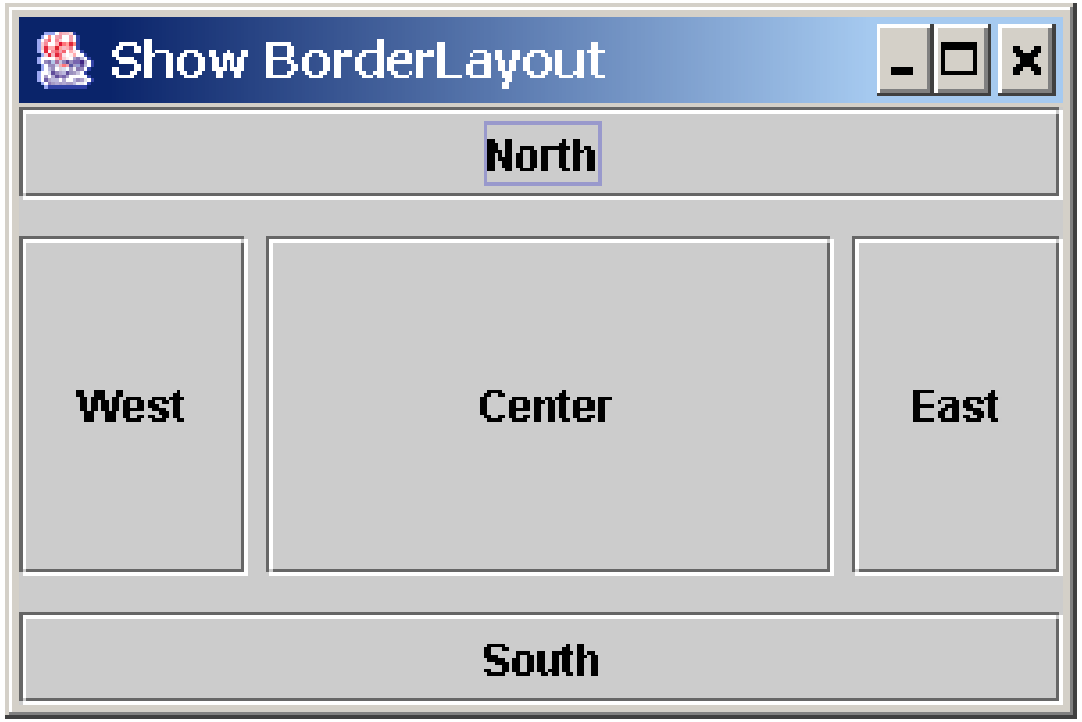
Nilai rows/columns boleh 0 (salah satu). Nilai yang tidak 0 akan fix.

Jika rows/columns keduanya tidak 0, maka nilai rows akan fix.

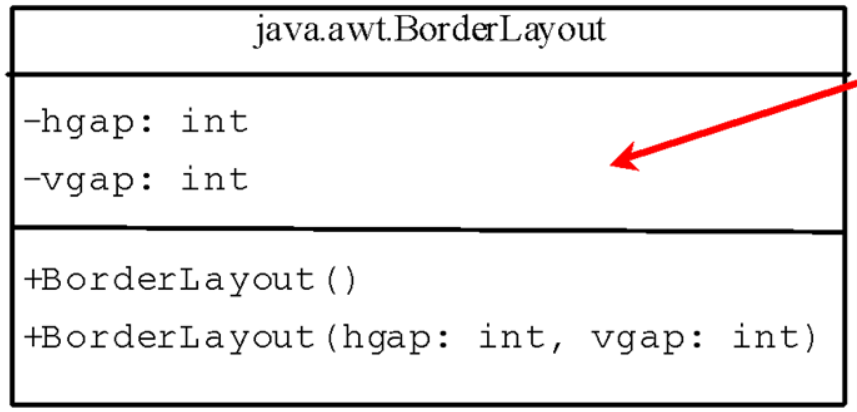
BorderLayout

The `BorderLayout` manager divides the container into five areas: East, South, West, North, and Center. Components are added to a `BorderLayout` by using the add method.

`add(Component, constraint)`, where `constraint` is `BorderLayout.EAST`, `BorderLayout.SOUTH`, `BorderLayout.WEST`, `BorderLayout.NORTH`, or `BorderLayout.CENTER`.



The `BorderLayout` Class



The `get` and `set` methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The horizontal gap between the components (default: 0).

The vertical gap between the components (default: 0).

Creates a default `BorderLayout` manager.

Creates a `BorderLayout` manager with a specified number for horizontal gap and vertical gap.

The `Color` Class

You can set colors for GUI components by using the java.awt.Color class. Colors are made of red, green, and blue components, each of which is represented by a byte value that describes its intensity, ranging from 0 (darkest shade) to 255 (lightest shade). This is known as the *RGB model*.

```
Color c = new Color(r, g, b);
```

`r`, `g`, and `b` specify a color by its red, green, and blue components.

Example:

```
Color c = new Color(228, 100, 255);
```


Standard Colors

Thirteen standard colors (black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow) are defined as constants in java.awt.Color.

The standard color names are constants, but they are named as variables with lowercase for the first word and uppercase for the first letters of subsequent words. Thus the color names violate the Java naming convention. Since JDK 1.4, you can also use the new constants: BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW.

Setting Colors

You can use the following methods to set the component's background and foreground colors:

```
setBackground(Color c)
setForeground(Color c)
```

Example:

```
jbt.setBackground(Color.yellow);
jbt.setForeground(Color.red);
```


The `Font` Class

Font Names

Standard font names that are supported in all platforms are: SansSerif, Serif, Monospaced, Dialog, or DialogInput.

Font Style

Font.PLAIN (0), Font.BOLD (1), Font.ITALIC (2), and Font.BOLD + Font.ITALIC (3)

```
Font myFont = new Font(name, style, size);
```

Example:

```
Font myFont = new Font("SansSerif ", Font.BOLD, 16);

Font myFont = new Font("Serif", Font.BOLD+Font.ITALIC, 12);

JButton jbtOK = new JButton("OK");

jbtOK.setFont(myFont);
```

Finding All Available Font Names

```
GraphicsEnvironment e =
  GraphicsEnvironment.getLocalGraphicsEnvironment();
String[] fontnames =   e.getAvailableFontFamilyNames();
for (int i = 0; i < fontnames.length; i++)
  System.out.println(fontnames[i]);
```

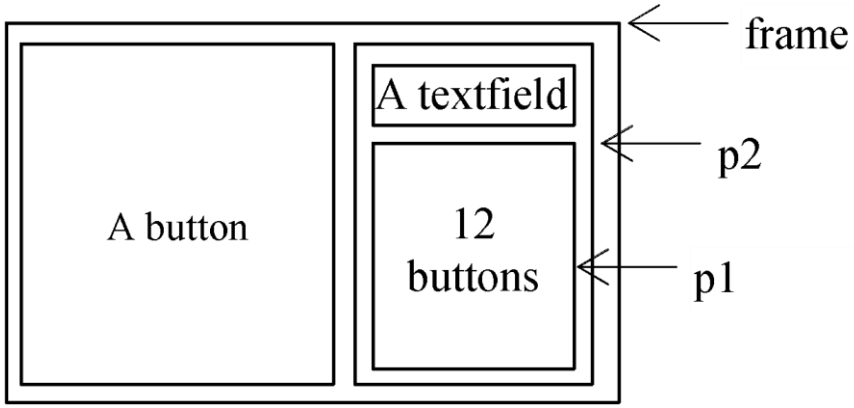

Using Panels as Sub-Containers

- Panels act as sub-containers for grouping user interface components.
- It is recommended that you place the user interface components in panels and place the panels in a frame. You can also place panels in a panel.
- To add a component to JFrame, you actually add it to the content pane of JFrame. To add a component to a panel, you add it directly to the panel using the add method.

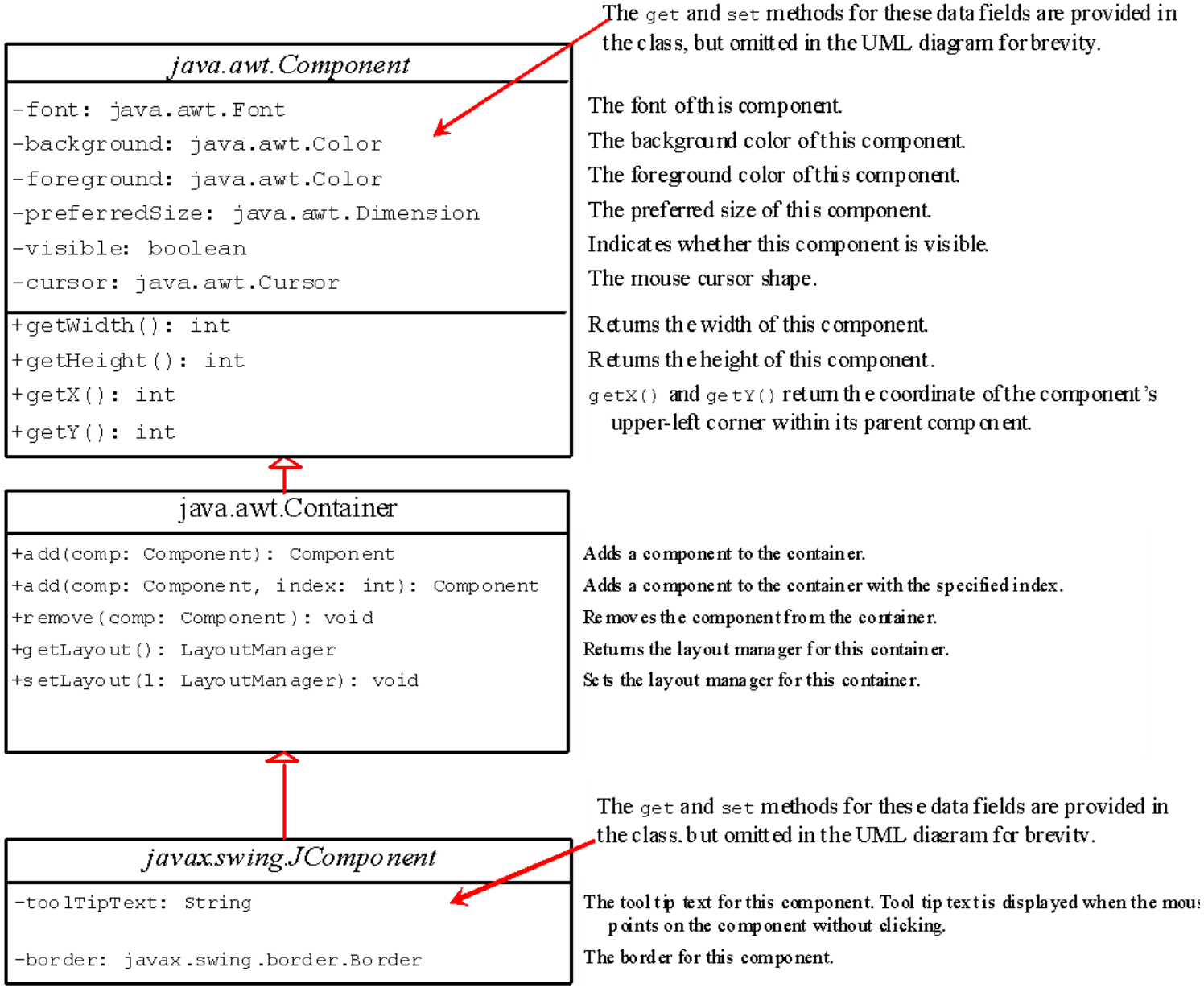
Creating a JPanel

You can use new JPanel() to create a panel with a default FlowLayout manager or new JPanel(LayoutManager) to create a panel with the specified layout manager. Use the add(Component) method to add a component to the panel. For example,

```
JPanel p = new JPanel();
p.add(new JButton("OK"));
```



Common Features of Swing Components



The `get` and `set` methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The font of this component.

The background color of this component.

The foreground color of this component.

The preferred size of this component.

Indicates whether this component is visible.

The mouse cursor shape.

Returns the width of this component.

Returns the height of this component.

`getX()` and `getY()` return the coordinate of the component's upper-left corner within its parent component.

Adds a component to the container.

Adds a component to the container with the specified index.

Removes the component from the container.

Returns the layout manager for this container.

Sets the layout manager for this container.

The `get` and `set` methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The tool tip text for this component. Tool tip text is displayed when the mouse points on the component without clicking.

The border for this component.

Borders

You can set a border on any object of the JComponent class. Swing has several types of borders. To create a titled border, use

new TitledBorder(String title).

To create a line border, use

new LineBorder(Color color, int width),

where width specifies the thickness of the line. For example, the following code displays a titled border on a panel:

```
JPanel panel = new JPanel();

panel.setBorder(new TitleBorder("My Panel"));
```


Image Icons

Java uses the javax.swing.ImageIcon class to represent an icon. An icon is a fixed-size picture; typically it is small and used to decorate components. Images are normally stored in image files. You can use new ImageIcon(filename) to construct an image icon. For example, the following statement creates an icon from an image file us.gif in the image directory under the current class path:

```
ImageIcon icon = new ImageIcon("image/us.gif");
```

Splash Screen

A *splash screen* is an image that is displayed while the application is starting up. If your program takes a long time to load, you may display a splash screen to alert the user. For example, the following command:

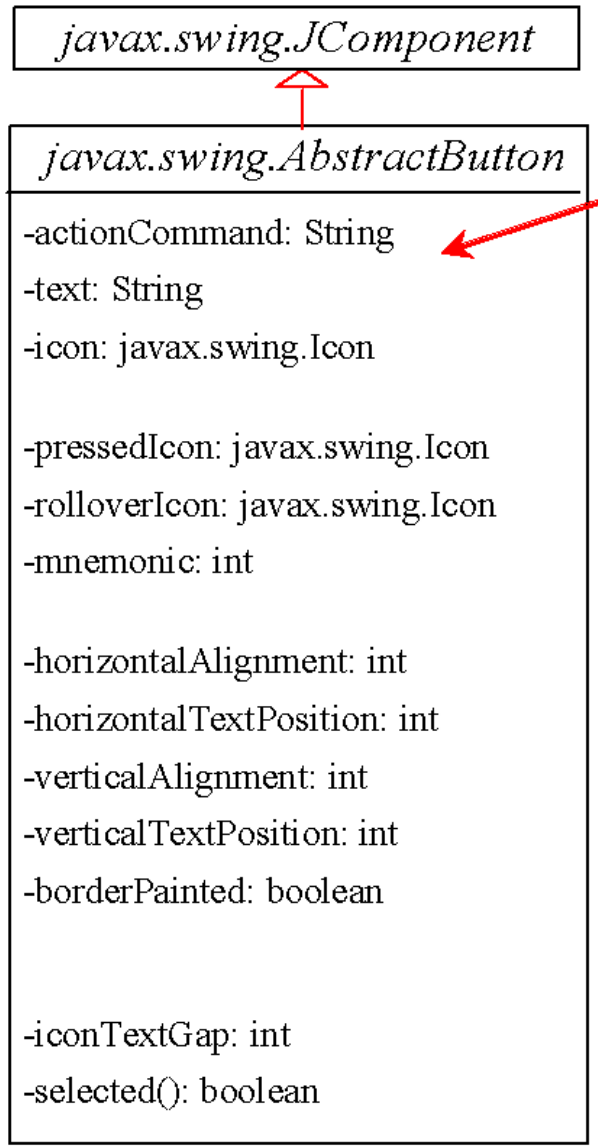
```
java –splash:image/us.gf TestImageIcon
```

displays an image while the program TestImageIcon is being loaded.

Buttons

A *button* is a component that triggers an action event when clicked. Swing provides regular buttons, toggle buttons, check box buttons, and radio buttons. The common features of these buttons are generalized in javax.swing.AbstractButton.

AbstractButton



The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The action command of this button.

The button's text (i.e., the text label on the button).

The button's default icon. This icon is also used as the "pressed" and "disabled" icon if there is no explicitly set pressed icon.

The pressed icon (displayed when the button is pressed).

The rollover icon (displayed when the mouse is over the button).

The mnemonic key value of this button. You can select the button by pressing the ALT key and the mnemonic key at the same time.

The horizontal alignment of the icon and text (default: CENTER).

The horizontal text position relative to the icon (default: RIGHT).

The vertical alignment of the icon and text (default: CENTER).

The vertical text position relative to the icon (default: CENTER).

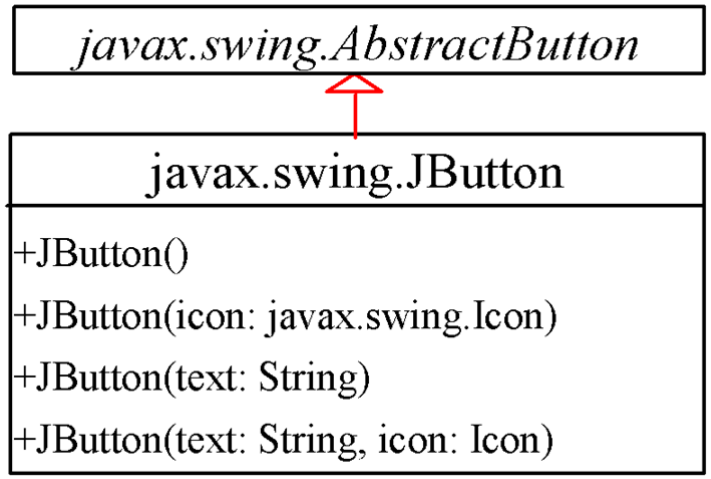
Indicates whether the border of the button is painted. By default, a regular button's border is painted, but the borders for a check box and a radio button is not painted.

The gap between the text and the icon on the button (JDK 1.4).

The state of the button. True if the check box or radio button is selected, false if it's not.

JButton

JButton inherits AbstractButton and provides several constructors to create buttons.



Creates a default button with no text and icon.
Creates a button with an icon.
Creates a button with text.
Creates a button with text and an icon.

`Jbutton` properties:

- `text`
- `icon`
- `mnemonic`
- `horizontalAlignment`
- `verticalAlignment`
- `horizontalTextPosition`
- `verticalTextPosition`
- `iconTextGap`

Default Icons, Pressed Icon, and Rollover Icon

A regular button has a default icon, pressed icon, and rollover icon. Normally, you use the default icon. All other icons are for special effects. A pressed icon is displayed when a button is pressed and a rollover icon is displayed when the mouse is over the button but not pressed.

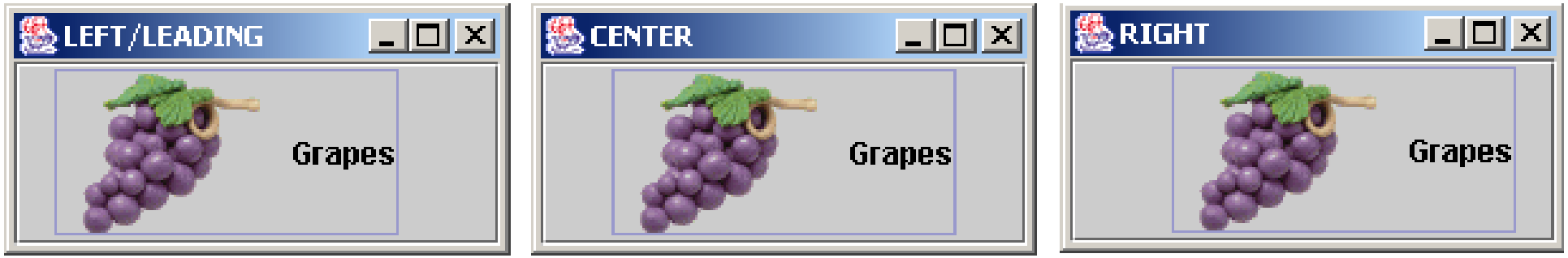
(A) Default icon

(B) Pressed icon

(C) Rollover icon

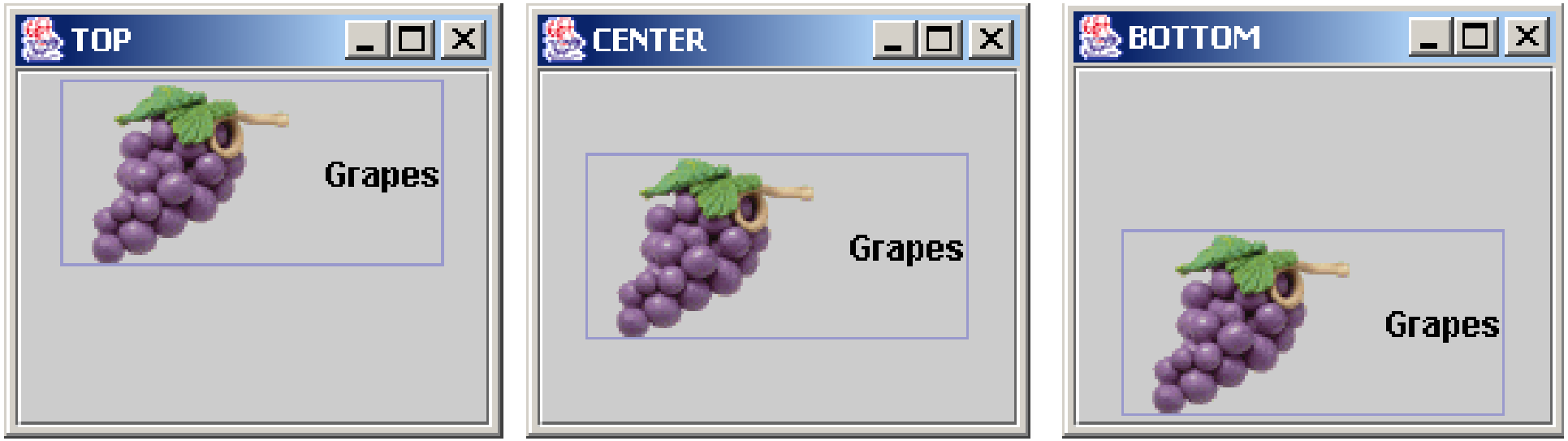
Horizontal Alignments

Horizontal alignment specifies how the icon and text are placed horizontally on a button. You can set the horizontal alignment using one of the five constants: LEADING, LEFT, CENTER, RIGHT, TRAILING. At present, LEADING and LEFT are the same and TRAILING and RIGHT are the same. Future implementation may distinguish them. The default horizontal alignment is SwingConstants.TRAILING.



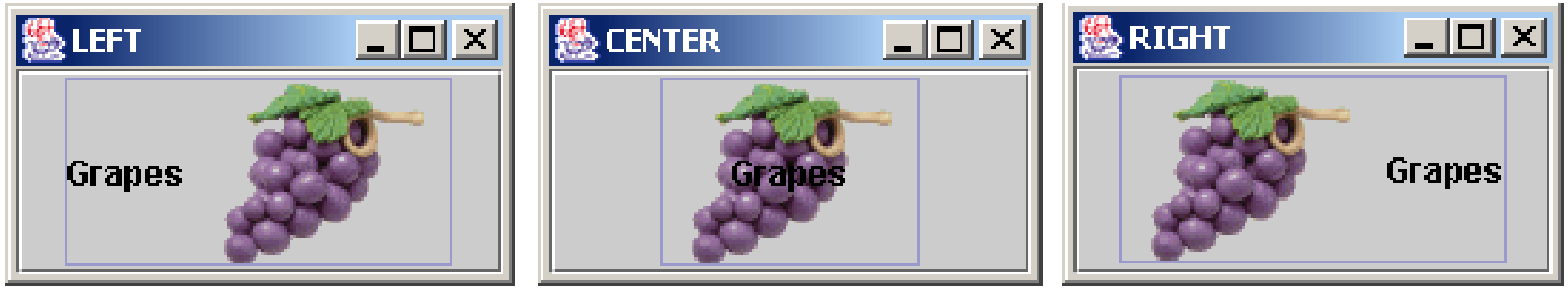
Vertical Alignments

Vertical alignment specifies how the icon and text are placed vertically on a button. You can set the vertical alignment using one of the three constants: TOP, CENTER, BOTTOM. The default vertical alignment is SwingConstants.CENTER.



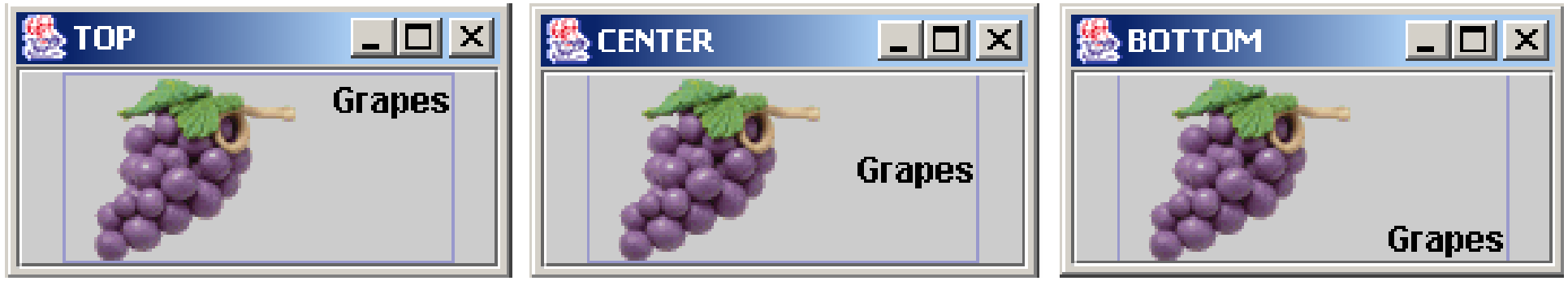
Horizontal Text Positions

Horizontal text position specifies the horizontal position of the text relative to the icon. You can set the horizontal text position using one of the five constants: LEADING, LEFT, CENTER, RIGHT, TRAILING. The default horizontal text position is SwingConstants.RIGHT.



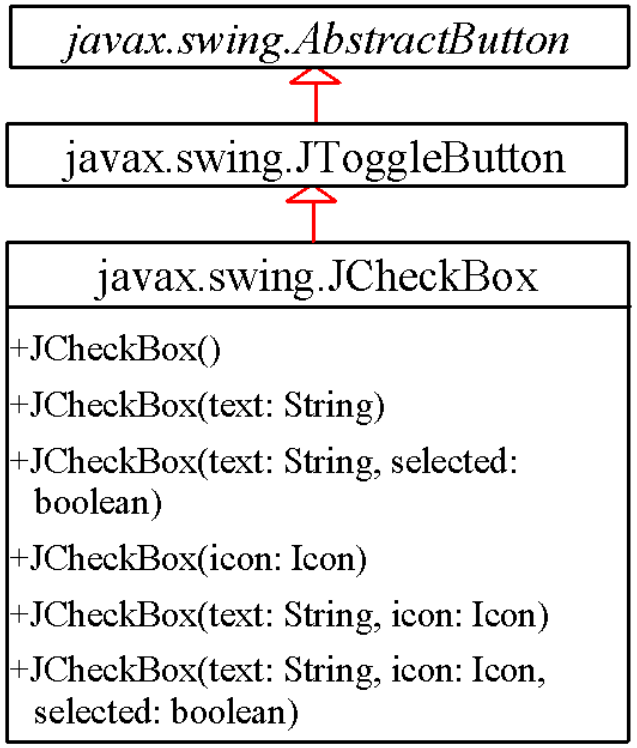
Vertical Text Positions

Vertical text position specifies the vertical position of the text relative to the icon. You can set the vertical text position using one of the three constants: TOP, CENTER. The default vertical text position is SwingConstants.CENTER.



JCheckBox

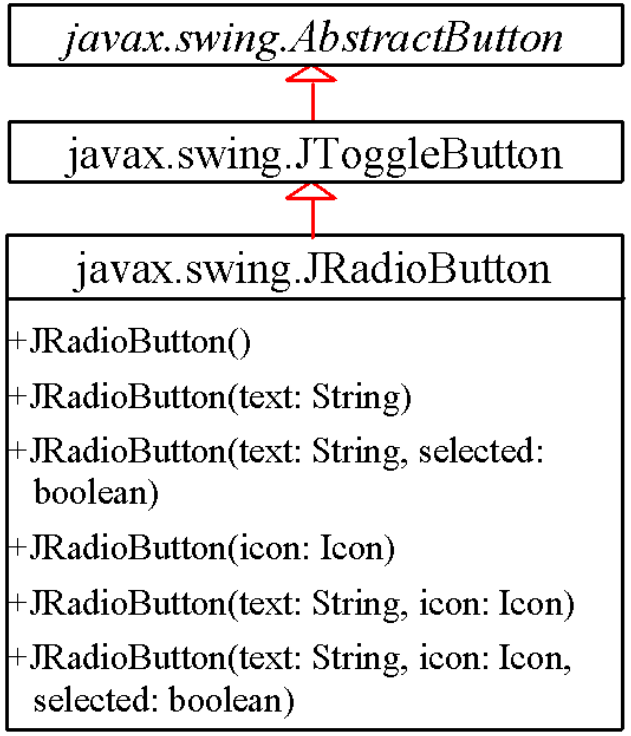
JCheckBox inherits all the properties such as text, icon, mnemonic, verticalAlignment, horizontalAlignment, horizontalTextPosition, verticalTextPosition, and selected from AbstractButton, and provides several constructors to create check boxes.



+JCheckBox()	Creates a default check box button with no text and icon.
+JCheckBox(text: String)	Creates a check box with text.
+JCheckBox(text: String, selected: boolean)	Creates a check box with text and specifies whether the check box is initially selected.
+JCheckBox(icon: Icon)	Creates a checkbox with an icon.
+JCheckBox(text: String, icon: Icon)	Creates a checkbox with text and an icon.
+JCheckBox(text: String, icon: Icon, selected: boolean)	Creates a check box with text and an icon, and specifies whether the check box is initially selected.

JRadioButton

Radio buttons are variations of check boxes. They are often used in the group, where only one button is checked at a time.



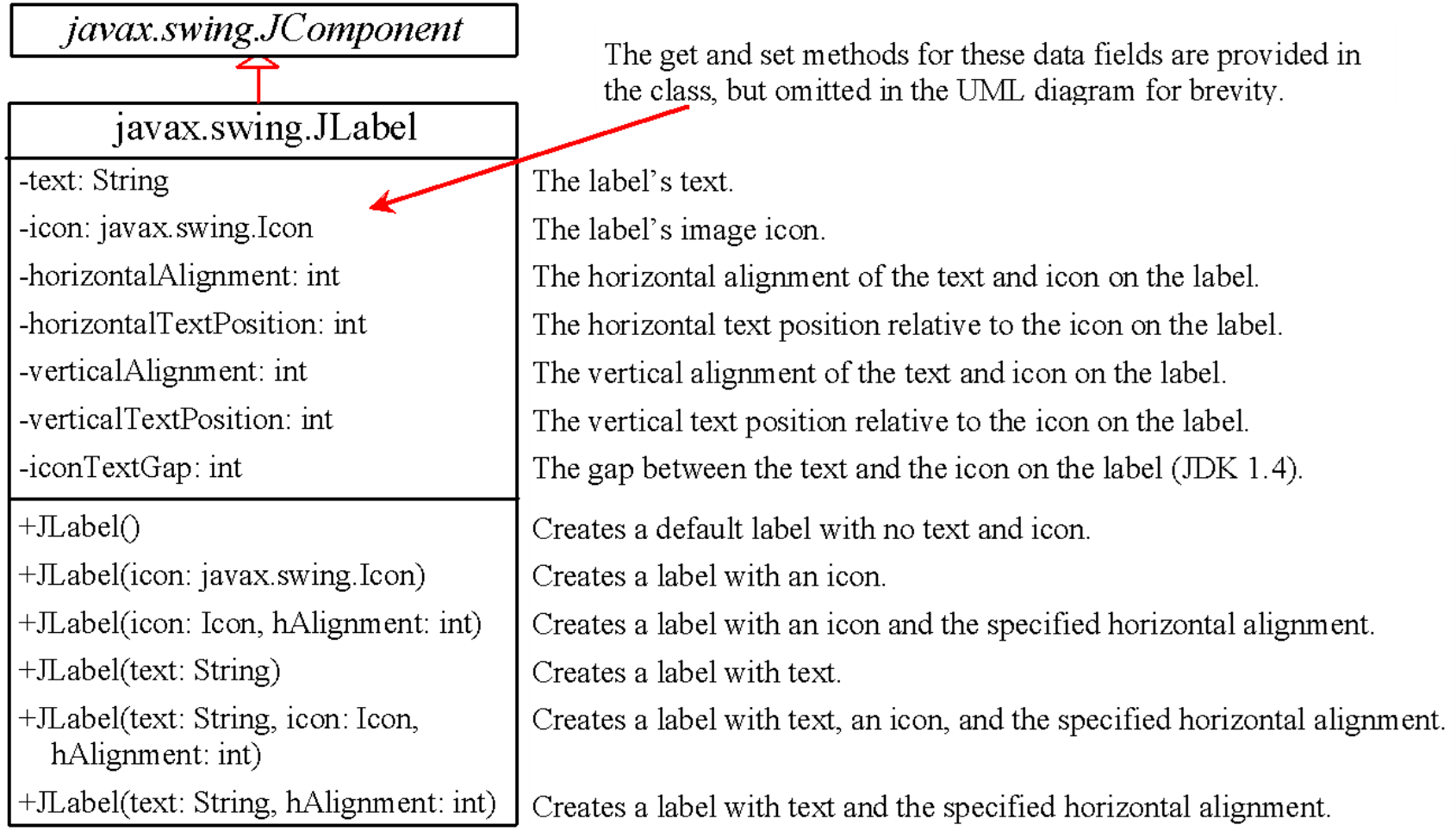
+JRadioButton()	Creates a default radio button with no text and icon.
+JRadioButton(text: String)	Creates a radio button with text.
+JRadioButton(text: String, selected: boolean)	Creates a radio button with text and specifies whether the radio button is initially selected.
+JRadioButton(icon: Icon)	Creates a radio button with an icon.
+JRadioButton(text: String, icon: Icon)	Creates a radio button with text and an icon.
+JRadioButton(text: String, icon: Icon, selected: boolean)	Creates a radio button with text and an icon, and specifies whether the radio button is initially selected.

Grouping Radio Buttons

```
ButtonGroup btg = new
   ButtonGroup();
btg.add(jrb1);
btg.add(jrb2);
```

JLabel

A *label* is a display area for a short text, an image, or both.



The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The label's text.

The label's image icon.

The horizontal alignment of the text and icon on the label.

The horizontal text position relative to the icon on the label.

The vertical alignment of the text and icon on the label.

The vertical text position relative to the icon on the label.

The gap between the text and the icon on the label (JDK 1.4).

Creates a default label with no text and icon.

Creates a label with an icon.

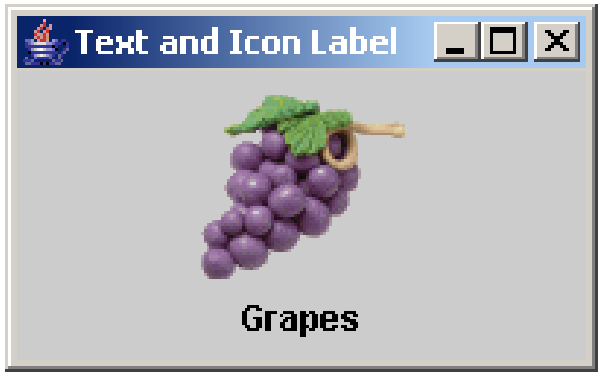
Creates a label with an icon and the specified horizontal alignment.

Creates a label with text.

Creates a label with text, an icon, and the specified horizontal alignment.

Creates a label with text and the specified horizontal alignment.

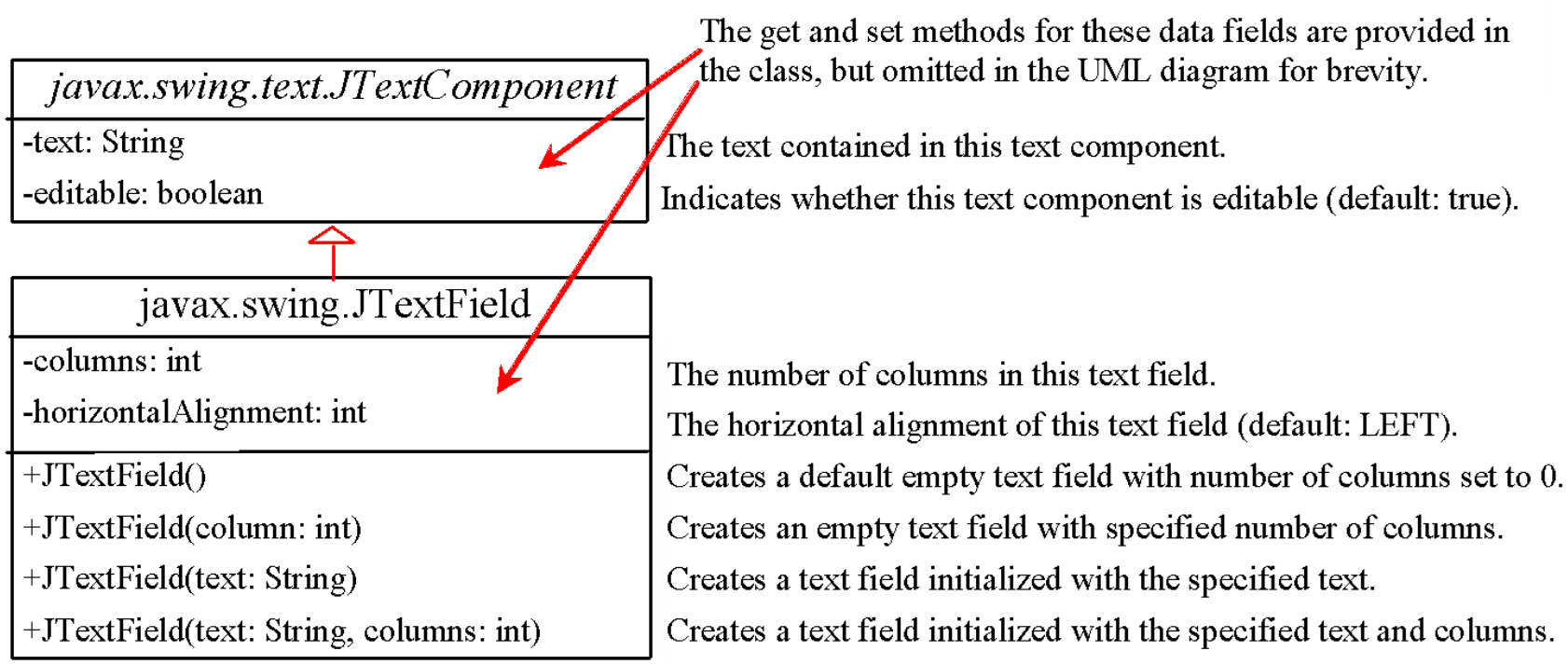
Using Labels




```
// Create an image icon from image file
ImageIcon icon = new ImageIcon("image/grapes.gif");

// Create a label with text, an icon,
// with centered horizontal alignment
JLabel jlbl = new JLabel("Grapes", icon,
SwingConstants.CENTER);

// Set label's text alignment and gap between text and icon
jlbl.setHorizontalTextPosition(SwingConstants.CENTER);
jlbl.setVerticalTextPosition(SwingConstants.BOTTOM);
jlbl.setIconTextGap(5);
```

JTextField

A *text field* is an input area where the user can type in characters. Text fields are useful in that they enable the user to enter in variable data (such as a name or a description).



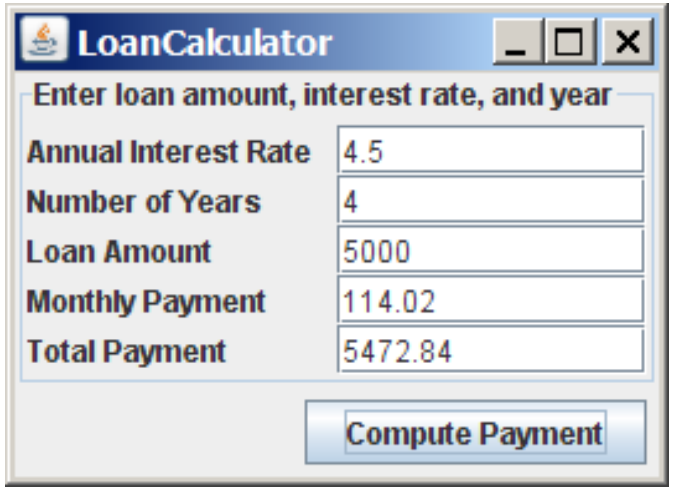
`JTextField` Methods

- `getText()`
  Returns the string from the text field.
- `setText(String text)`
  Puts the given string in the text field.
- `setEditable(boolean editable)`
  Enables or disables the text field to be edited. By default, `editable` is `true`.
- `setColumns(int)`
  Sets the number of columns in this text field.
  The length of the text field is changeable.

Event-Driven Programming

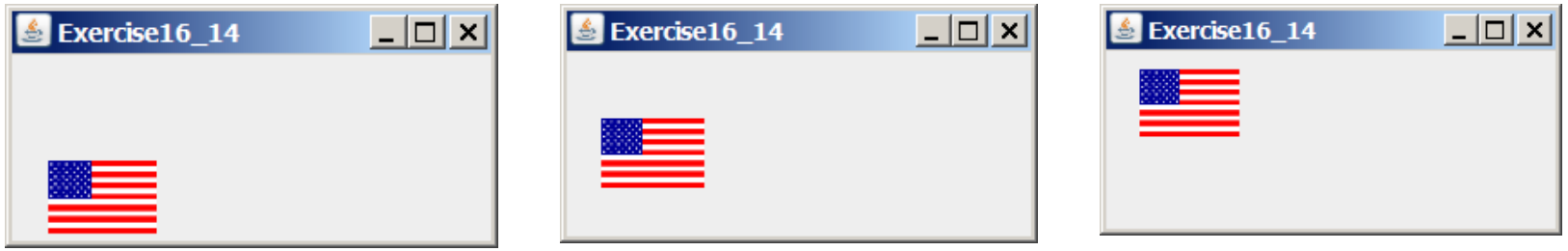
Motivations

Suppose you wish to write a GUI program that lets the user enter the loan amount, annual interest rate, and number of years, and click the *Compute Loan* button to obtain the monthly payment and total payment. How do you accomplish the task? You have to use event-driven programming to write the code to respond to the button-clicking event.



Motivations

Suppose you wish to write a program that animates a rising flag, as shown in Figure 16.1(b-d). How do you accomplish the task? There are several solutions to this problem. An effective way to solve it is to use a timer in event-driven programming, which is the subject of this chapter.



Objectives

- To get a taste of event-driven programming (§16.1).
- To describe events, event sources, and event classes (§16.2).
- To define listener classes, register listener objects with the source object, and write the code to handle events (§16.3).
- To define listener classes using inner classes (§16.4).
- To define listener classes using anonymous inner classes (§16.5).
- To explore various coding styles for creating and registering listener classes (§16.6).
- To develop a GUI application for a loan calculator (§16.7).
- To write programs to deal with **MouseEvent**s (§16.8).
- To simplify coding for listener classes using listener interface adapters (§16.9).
- To write programs to deal with **KeyEvent**s (§16.10).
- To use the **javax.swing.Timer** class to control animations (§16.11).

Procedural vs. Event-Driven Programming

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

Handling GUI Events

Source object (e.g., button)

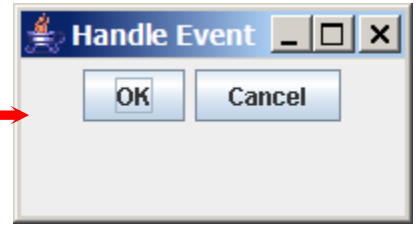
Listener object contains a method for processing the event.

animation

Trace Execution

```
public class HandleEvent extends JFrame {
  public HandleEvent() {
    …
   OKListenerClass listener1 = new OKListenerClass();
    jbtOK.addActionListener(listener1);
    …
  }

  public static void main(String[] args) {
    …
  }
}

class OKListenerClass implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

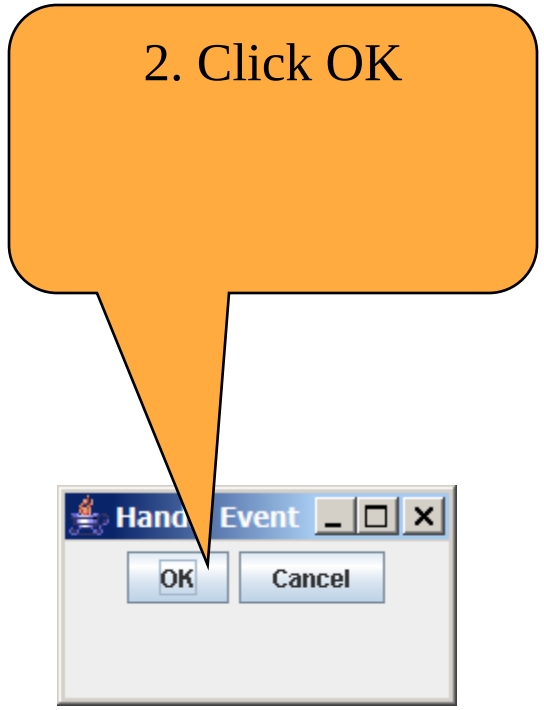
1. Start from the main method to create a window and display it



animation

Trace Execution

```
public class HandleEvent extends JFrame {
  public HandleEvent() {
    …
   OKListenerClass listener1 = new OKListenerClass();
    jbtOK.addActionListener(listener1);
    …
  }

  public static void main(String[] args) {
    …
  }
}

class OKListenerClass implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

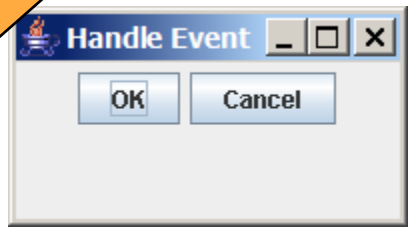


animation

Trace Execution

```
public class HandleEvent extends JFrame {
  public HandleEvent() {
    …
   OKListenerClass listener1 = new OKListenerClass();
    jbtOK.addActionListener(listener1);
    …
  }

  public static void main(String[] args) {
    …
  }
}

class OKListenerClass implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

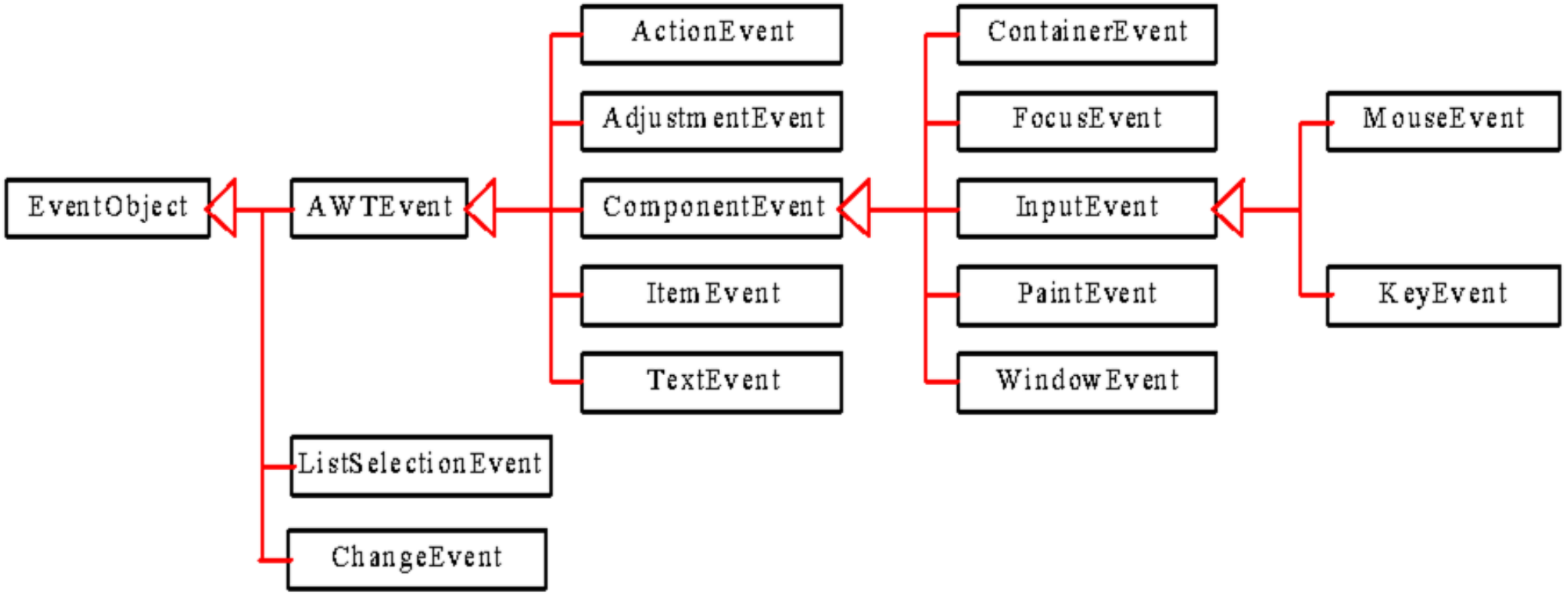
3. Click OK. The JVM invokes the listener's actionPerformed method



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

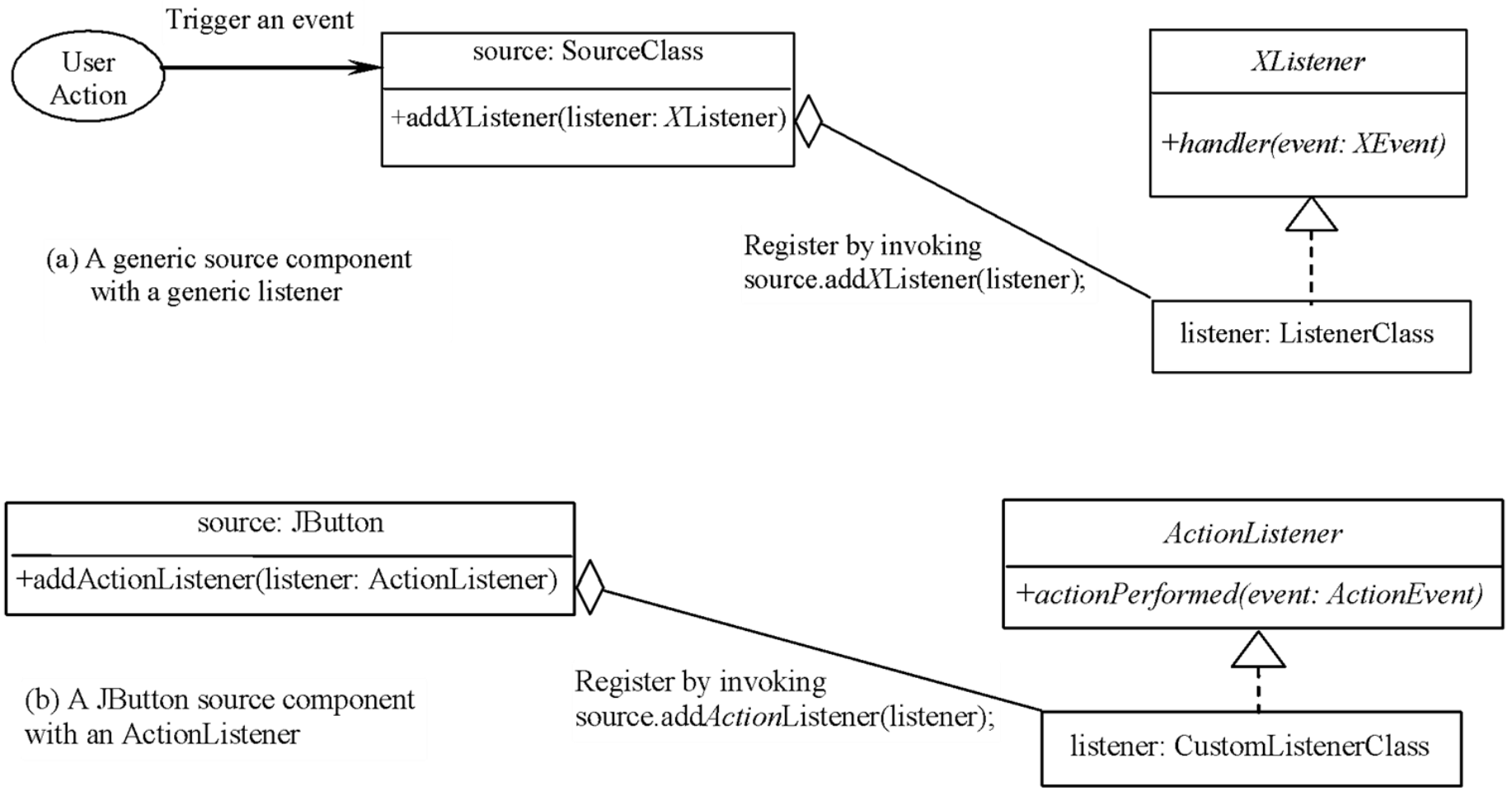


Event Information

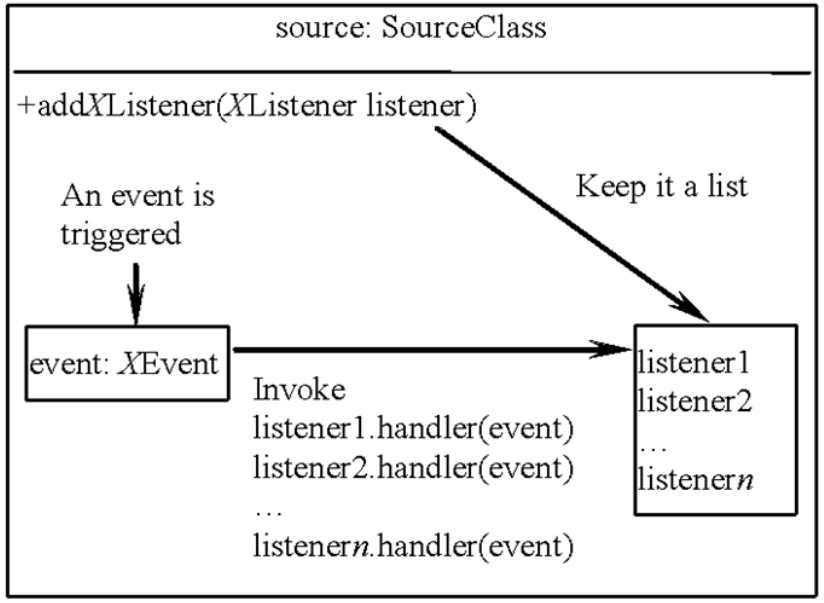
An event object contains whatever properties are pertinent to the event. You can identify the source object of the event using the getSource() instance method in the EventObject class. The subclasses of EventObject deal with special types of events, such as button actions, window events, component events,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

User Action	Source Object	Event Type Generated
Click a button	`JButton`	`ActionEvent`
Click a check box	`JCheckBox`	`ItemEvent`, `ActionEvent`
Click a radio button	`JRadioButton`	`ItemEvent`, `ActionEvent`
Press return on a text field	`JTextField`	`ActionEvent`
Select a new item	`JComboBox`	`ItemEvent`, `ActionEvent`
Window opened, closed, etc.	`Window`	`WindowEvent`
Mouse pressed, released, etc.	`Component`	`MouseEvent`
Key released, pressed, etc.	`Component`	`KeyEvent`

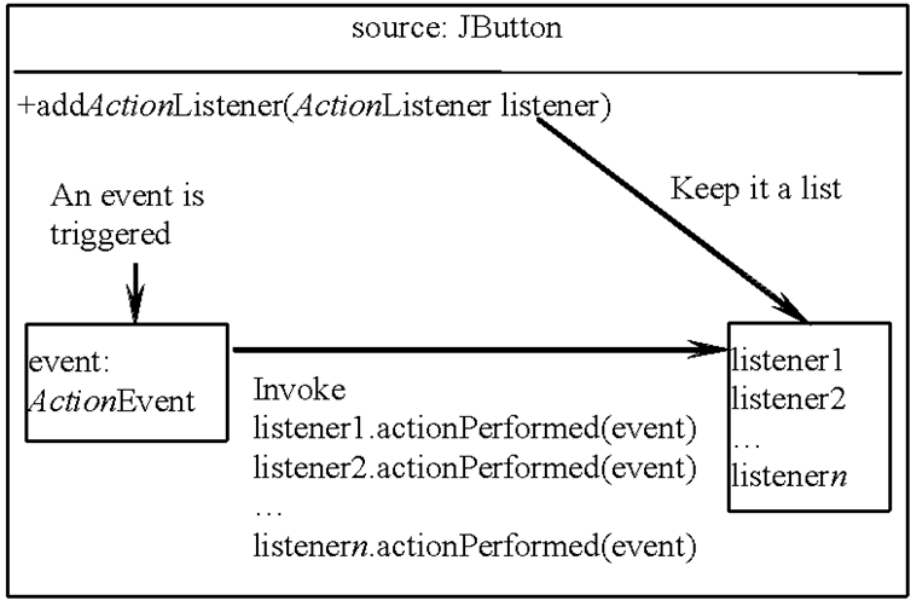
The Delegation Model



Internal Function of a Source Component



(a) Internal function of a generic source object



(b) Internal function of a JButton object

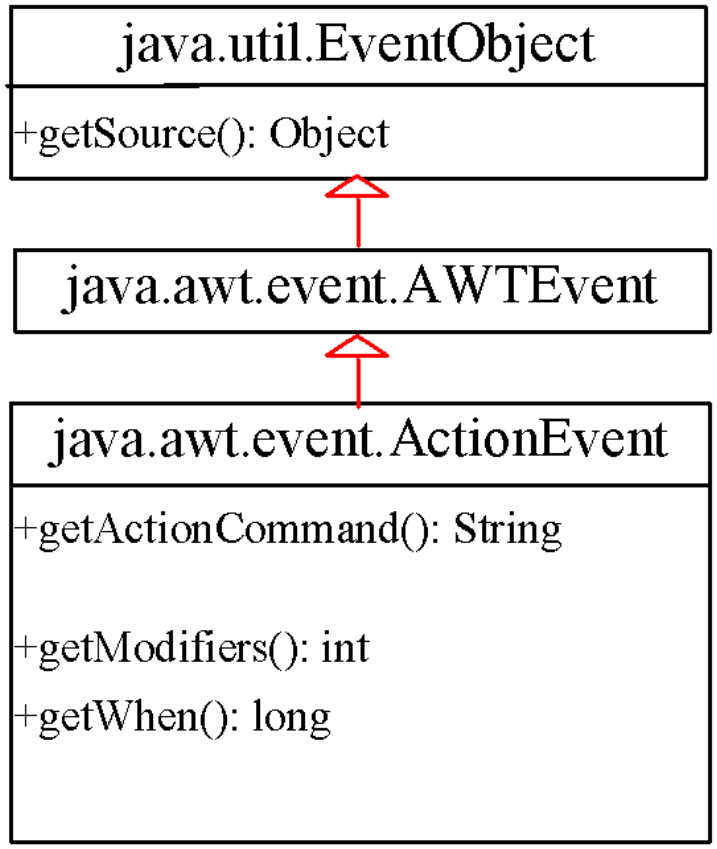
The Delegation Model: Example

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


Selected Event Handlers

Event Class	Listener Interface	Listener Methods (Handlers)
ActionEvent	ActionListener	actionPerformed(ActionEvent)
ItemEvent	ItemListener	itemStateChanged(ItemEvent)
WindowEvent	WindowListener	windowClosing(WindowEvent)
		windowOpened(WindowEvent)
		windowIconified(WindowEvent)
		windowDeiconified(WindowEvent)
		windowClosed(WindowEvent)
		windowActivated(WindowEvent)
		windowDeactivated(WindowEvent)
ContainerEvent	ContainerListener	componentAdded(ContainerEvent)
		componentRemoved(ContainerEvent)
MouseEvent	MouseListener	mousePressed(MouseEvent)
		mouseReleased(MouseEvent)
		mouseClicked(MouseEvent)
		mouseExited(MouseEvent)
		mouseEntered(MouseEvent)
KeyEvent	KeyListener	keyPressed(KeyEvent)
		keyReleased(KeyEvent)
		keyTypeed(KeyEvent)

java.awt.event.ActionEvent



Returns the object on which the event initially occurred.

Returns the command string associated with this action. For a button, its text is the command string.

Returns the modifier keys held down during this action event.

Returns the timestamp when this event occurred. The time is the number of milliseconds since January 1, 1970, 00:00:00 GMT.

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 class: A class is a member of another class.

Advantages: In some applications, you can use an inner class to make programs simple.

- 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.

```
public class Test {
   ...
}

public class A {
   ...
}
```

(a)

```
public class Test {
   ...

   // Inner class
   public class A {
      ...
   }
}
```

(b)

```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

(c)

Inner Classes

- 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 .
- An inner class can be declared public, protected, or private subject to the same visibility rules applied to a member of the class.
- An inner class can be declared static. A static inner class can be accessed using the outer class name. A static inner class cannot access nonstatic members of the outer class

Anonymous Inner Classes

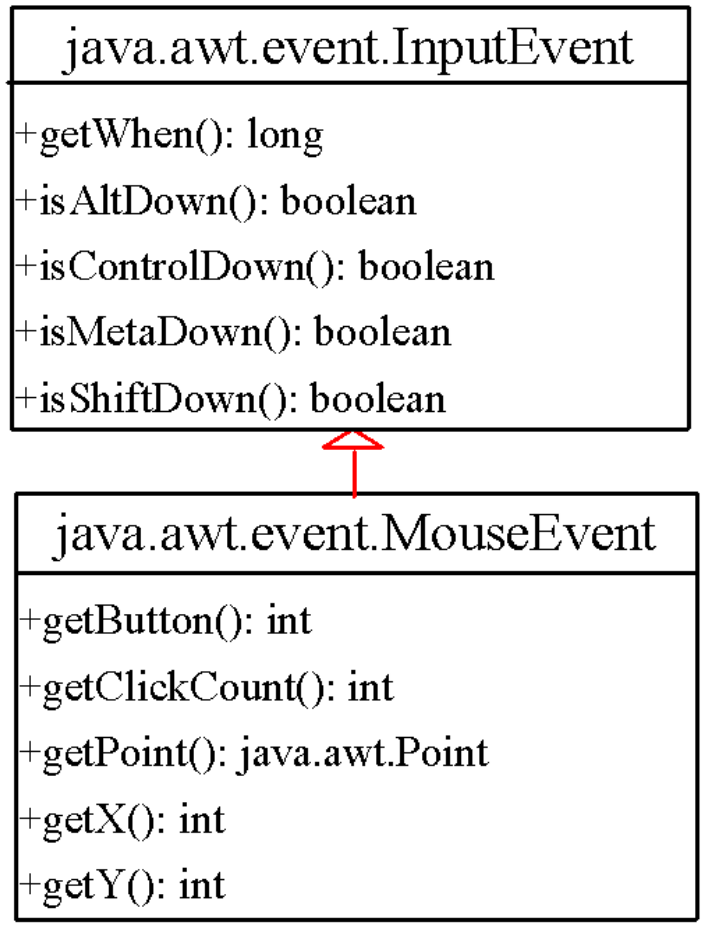
- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit extends or implements clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is Object().
- An anonymous inner class is compiled into a class named OuterClassName$*n*.class. For example, if the outer class Test has two anonymous inner classes, these two classes are compiled into Test$1.class and Test$2.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
}
```


MouseEvent



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.

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.

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.

The `KeyEvent` Class

- Methods:

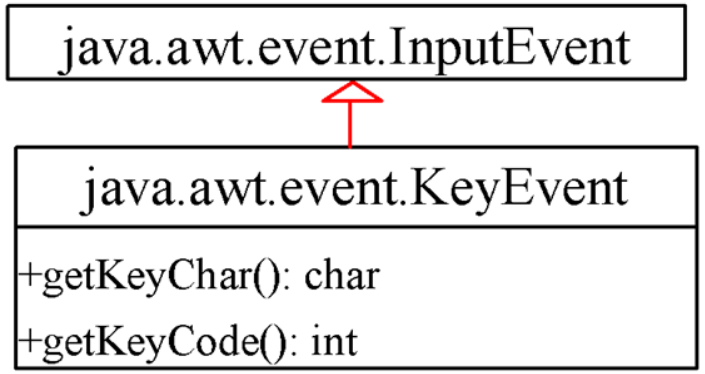
  `getKeyChar()` method

  `getKeyCode()` method

- Keys:

  Home `VK_HOME`
  End `VK_END`
  Page Up `VK_PGUP`
  Page Down `VK_PGDN`
  etc...

The `KeyEvent` Class, cont.



Returns the character associated with the key in this event.

Returns the integer keyCode associated with the key in this event.