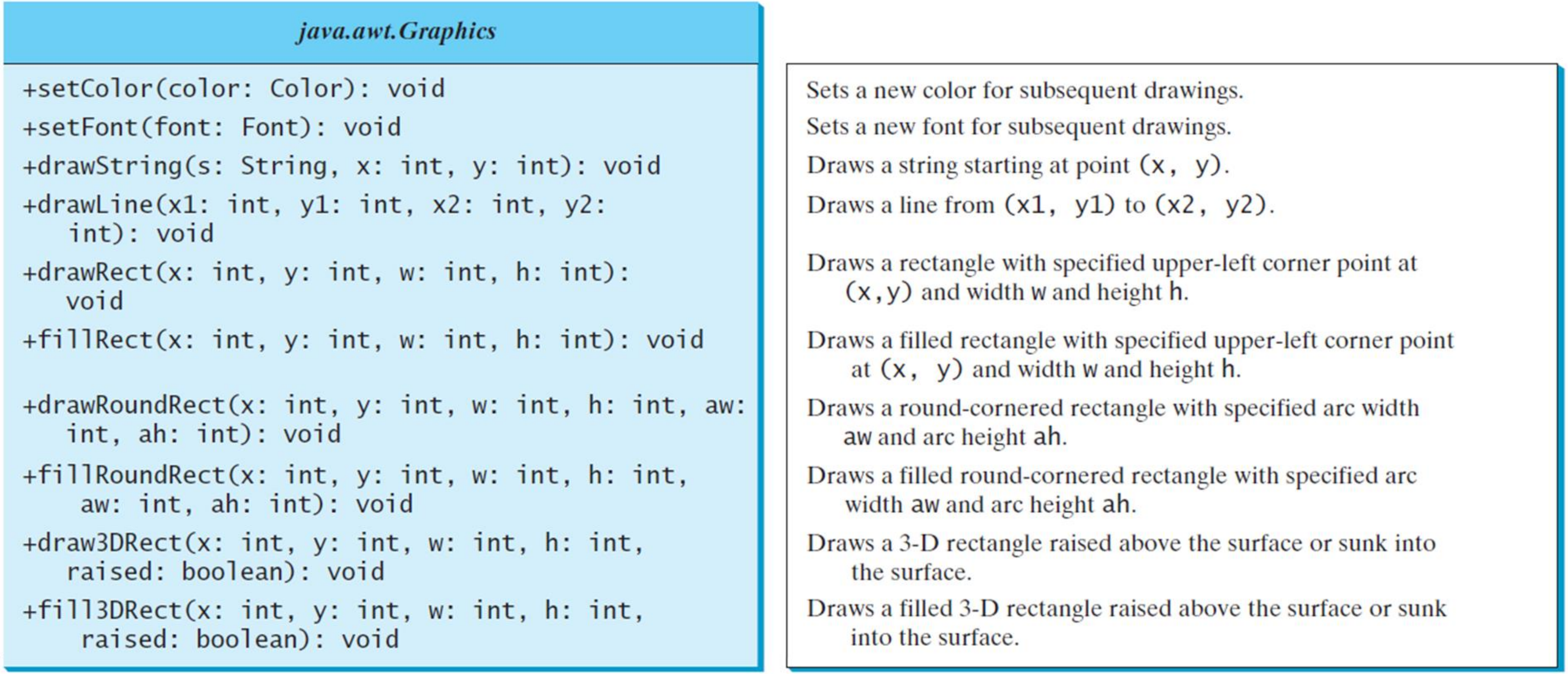
Graphics – Introduction

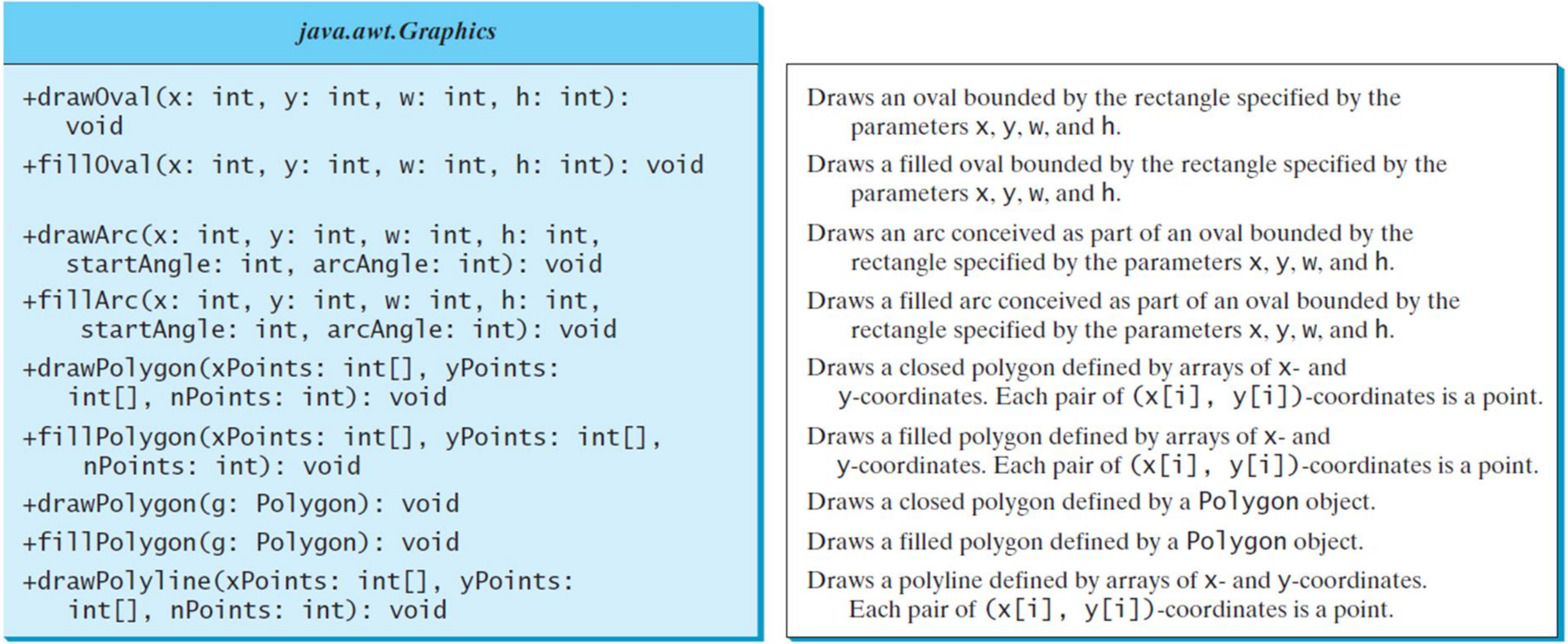
- You can draw custom shapes on a GUI component.
- Suppose you want to draw shapes such as a bar chart, a clock, or a stop sign. How do you do so?
- You can draw shapes using the drawing methods in the **Graphics** class.

The **Graphics** Class

- Each GUI component has a graphics context, which is an object of the **Graphics** class and the **Graphics** class contains the methods for drawing various shapes.
- The **Graphics** class provides the methods for drawing strings, lines, rectangles, ovals, arcs, polygons, and polylines.
- Think of a GUI component as a piece of paper and the **Graphics** object as a pencil or paintbrush. You can apply the methods in the **Graphics** class to draw graphics on a GUI component.



java.awt.Graphics	
`+setColor(color: Color): void`	Sets a new color for subsequent drawings.
`+setFont(font: Font): void`	Sets a new font for subsequent drawings.
`+drawString(s: String, x: int, y: int): void`	Draws a string starting at point `(x, y)`.
`+drawLine(x1: int, y1: int, x2: int, y2: int): void`	Draws a line from `(x1, y1)` to `(x2, y2)`.
`+drawRect(x: int, y: int, w: int, h: int): void`	Draws a rectangle with specified upper-left corner point at `(x,y)` and width `w` and height `h`.
`+fillRect(x: int, y: int, w: int, h: int): void`	Draws a filled rectangle with specified upper-left corner point at `(x, y)` and width `w` and height `h`.
`+drawRoundRect(x: int, y: int, w: int, h: int, aw: int, ah: int): void`	Draws a round-cornered rectangle with specified arc width `aw` and arc height `ah`.
`+fillRoundRect(x: int, y: int, w: int, h: int, aw: int, ah: int): void`	Draws a filled round-cornered rectangle with specified arc width `aw` and arc height `ah`.
`+draw3DRect(x: int, y: int, w: int, h: int, raised: boolean): void`	Draws a 3-D rectangle raised above the surface or sunk into the surface.
`+fill3DRect(x: int, y: int, w: int, h: int, raised: boolean): void`	Draws a filled 3-D rectangle raised above the surface or sunk into the surface.



java.awt.Graphics

+drawOval(x: int, y: int, w: int, h: int): void	Draws an oval bounded by the rectangle specified by the parameters x, y, w, and h.
+fillOval(x: int, y: int, w: int, h: int): void	Draws a filled oval bounded by the rectangle specified by the parameters x, y, w, and h.
+drawArc(x: int, y: int, w: int, h: int, startAngle: int, arcAngle: int): void	Draws an arc conceived as part of an oval bounded by the rectangle specified by the parameters x, y, w, and h.
+fillArc(x: int, y: int, w: int, h: int, startAngle: int, arcAngle: int): void	Draws a filled arc conceived as part of an oval bounded by the rectangle specified by the parameters x, y, w, and h.
+drawPolygon(xPoints: int[], yPoints: int[], nPoints: int): void	Draws a closed polygon defined by arrays of x- and y-coordinates. Each pair of (x[i], y[i])-coordinates is a point.
+fillPolygon(xPoints: int[], yPoints: int[], nPoints: int): void	Draws a filled polygon defined by arrays of x- and y-coordinates. Each pair of (x[i], y[i])-coordinates is a point.
+drawPolygon(g: Polygon): void	Draws a closed polygon defined by a Polygon object.
+fillPolygon(g: Polygon): void	Draws a filled polygon defined by a Polygon object.
+drawPolyline(xPoints: int[], yPoints: int[], nPoints: int): void	Draws a polyline defined by arrays of x- and y-coordinates. Each pair of (x[i], y[i])-coordinates is a point.

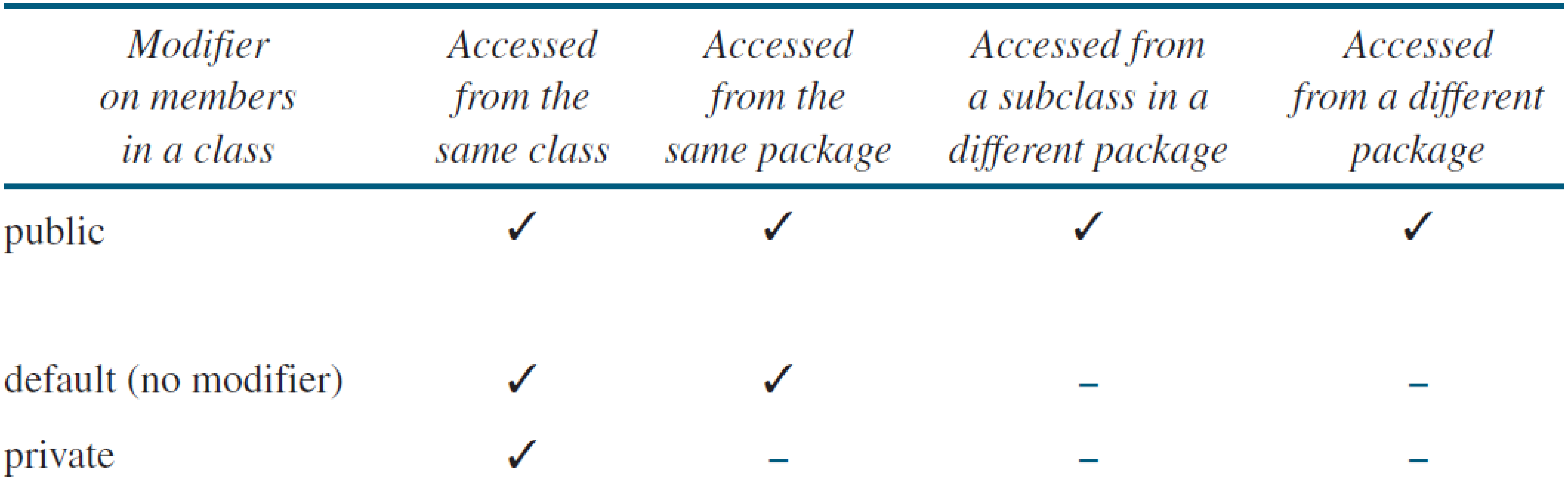
Key method() for Drawings

- Whenever a component (e.g., a button, a label, or a panel) is displayed, the JVM automatically creates a **Graphics** object for the component and passes this object to invoke the **paintComponent** method to display the drawings.

  protected void paintComponent(Graphics g)

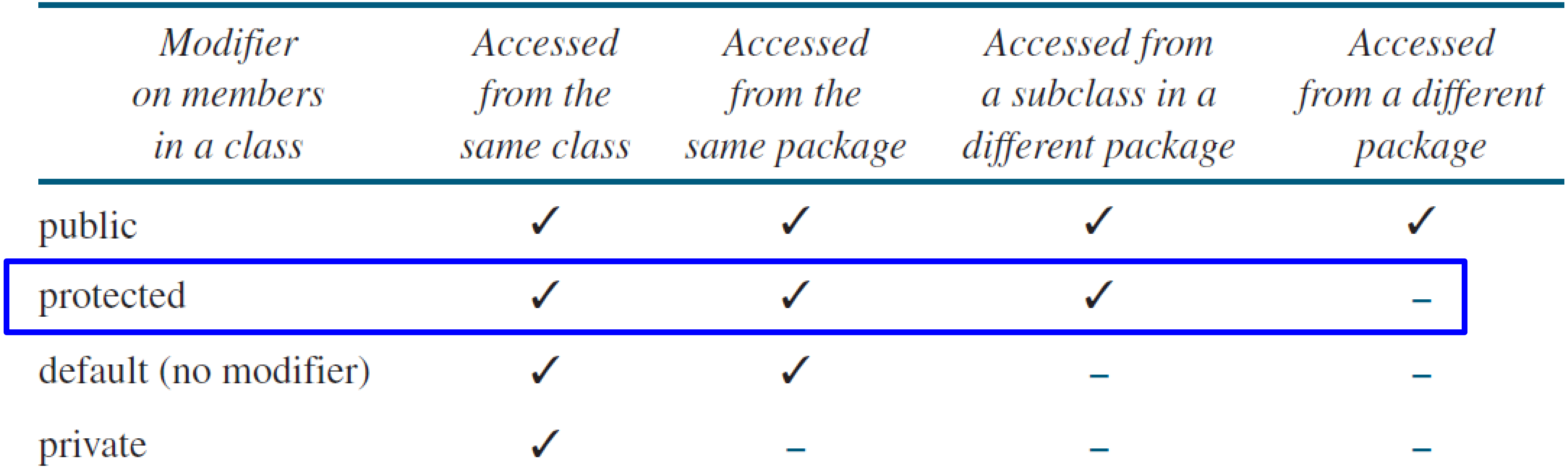
- This method, defined in the **JComponent** class, is invoked whenever a component is first displayed or redisplayed.
- To draw on a component, you need to define a class that extends **JPanel** and overrides its **paintComponent** method to specify what to draw.

Data and Methods Visibility



Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

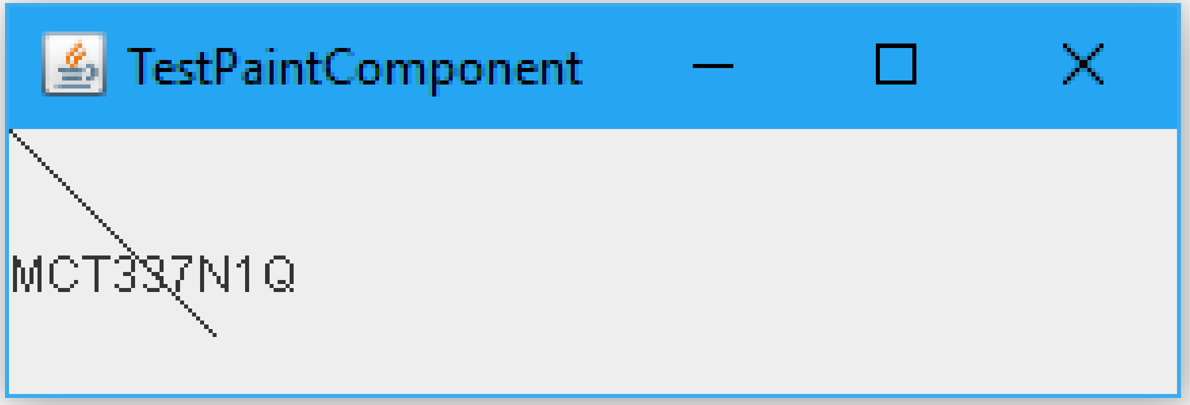
Data and Methods Visibility



Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

Graphics – Example

TestPaintComponent.java

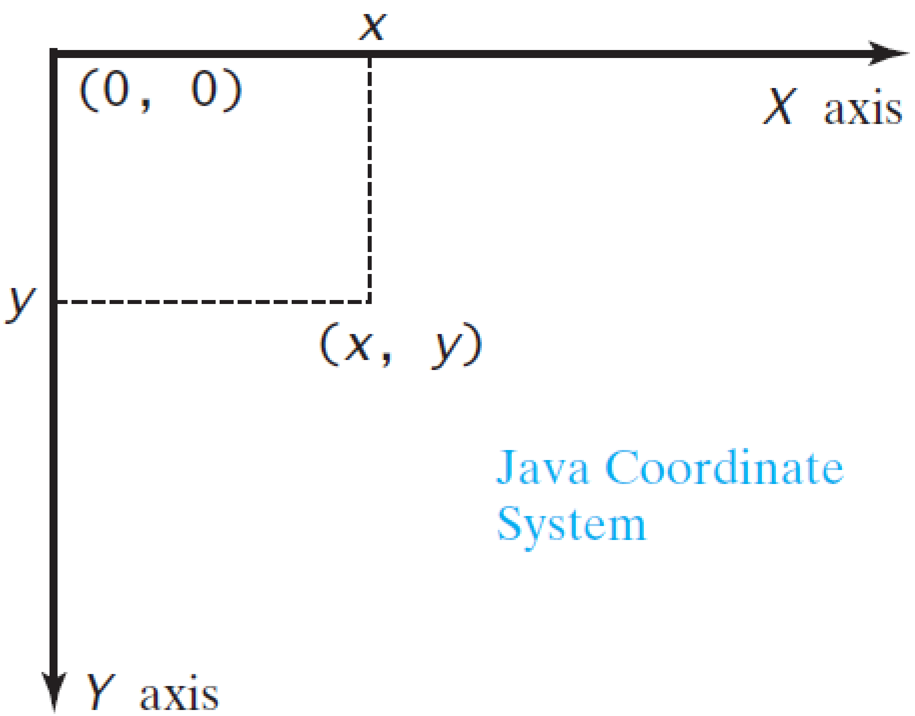


MCT337N1Q

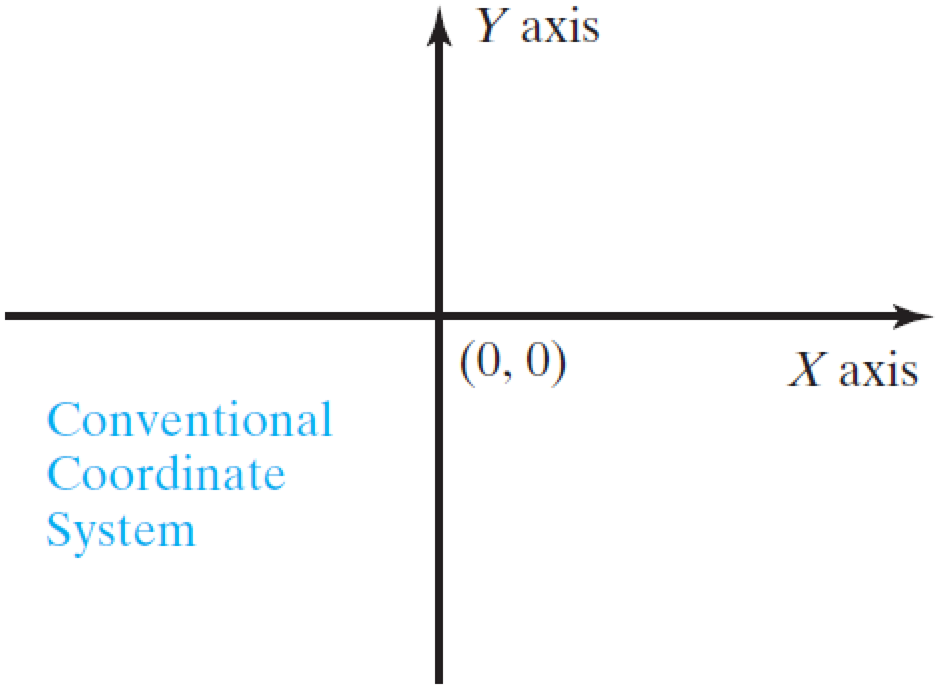
Analyzing Example

- The **paintComponent** method is automatically invoked to paint graphics when the component is first displayed or whenever the component needs to be redisplayed.
- Invoking **super.paintComponent(g)** invokes the **paintComponent** method defined in the superclass. This is necessary to ensure that the viewing area is cleared before a new drawing is displayed.
- **drawLine** method draws a line from ($\mathbf{x_0}$, $\mathbf{y_0}$) to ($\mathbf{x_1}$, $\mathbf{y_1}$) and **drawString** method draws the string at specific location.

The Java coordinate system is measured in pixels, with **(0, 0)** at its upper-left corner.



Java Coordinate System



Conventional Coordinate System

Drawing by using JPanel

- Panels are invisible and are used as small containers that group components to achieve a desired layout.
- Another important use of **JPanel** is for drawing.
- You can draw things on any Swing GUI component, but normally you should use a **JPanel** as a canvas upon which to draw things.

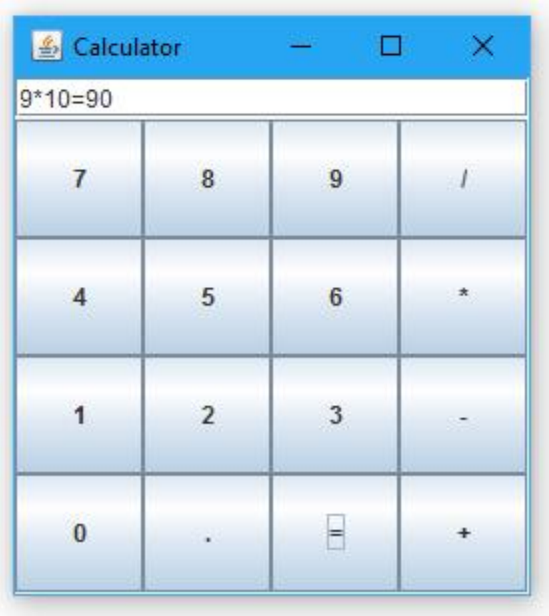
```
class DrawingPanel extends Jpanel {
}
```

repaint Method

- The **repaint** method is defined in the **Component** class.
- Invoking **repaint** causes the **paintComponent** method to be called.
- The **repaint** method is invoked to refresh the viewing area suchas:
- The size of the circle,..
- Typically, you call it if you have new things to display.

Event-Driven Programming – Introduction

- You can write code to process events such as a button click, mouse movement, and keystrokes or a timer.
- Suppose you want to write a GUI program such as calculator that lets the user enter numbers, and mathematical operators and click the equal button to obtain the result.
- How do you accomplish the task?



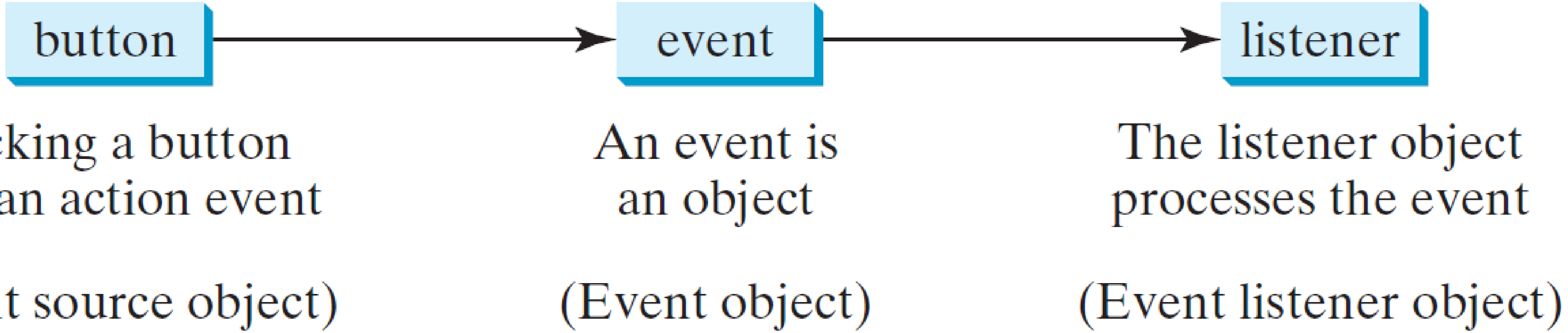
9*10=90

event-driven programming

Event-Driven Programming – Basic Example

- This example displays two buttons in a frame.
- To respond to a **button click**, you need to write the code to process the button-clicking action.
- The button is an **event source object**—where the action originates.
- You need to create an **object** capable of handling the action event on a button. This **object** is called an **event listener**.

A listener object processes the event fired from the source object.



ActionListener

- Not all objects can be listeners for an ActionEvent. To be a listeners of an action event, **two** requirements must be met:

1. The object must be an instance of the **ActionListener** interface. This interface defines the **common behavior** for all action listeners.
2. The **ActionListener** object (listener) **must be registered** with the **event source object** using the method **source.addActionListener(listener)**.

Note

- A superclass defines common behavior for related subclasses.
- An interface can be used to define common behavior for classes (including unrelated classes).
- An interface is a class-like construct that contains only constants and abstract (general) methods.

ActionListener – Example

- The **ActionListener** interface contains the **actionPerformed** method for processing the action event.
- Your listener class **must override** this method to respond to the event.
- Our example processes the **ActionEvent** on the two buttons.
- When you click the **OK** button, the message “OK button clicked” is displayed.
- When you click the **Cancel** button, the message “Cancel button clicked” is displayed.

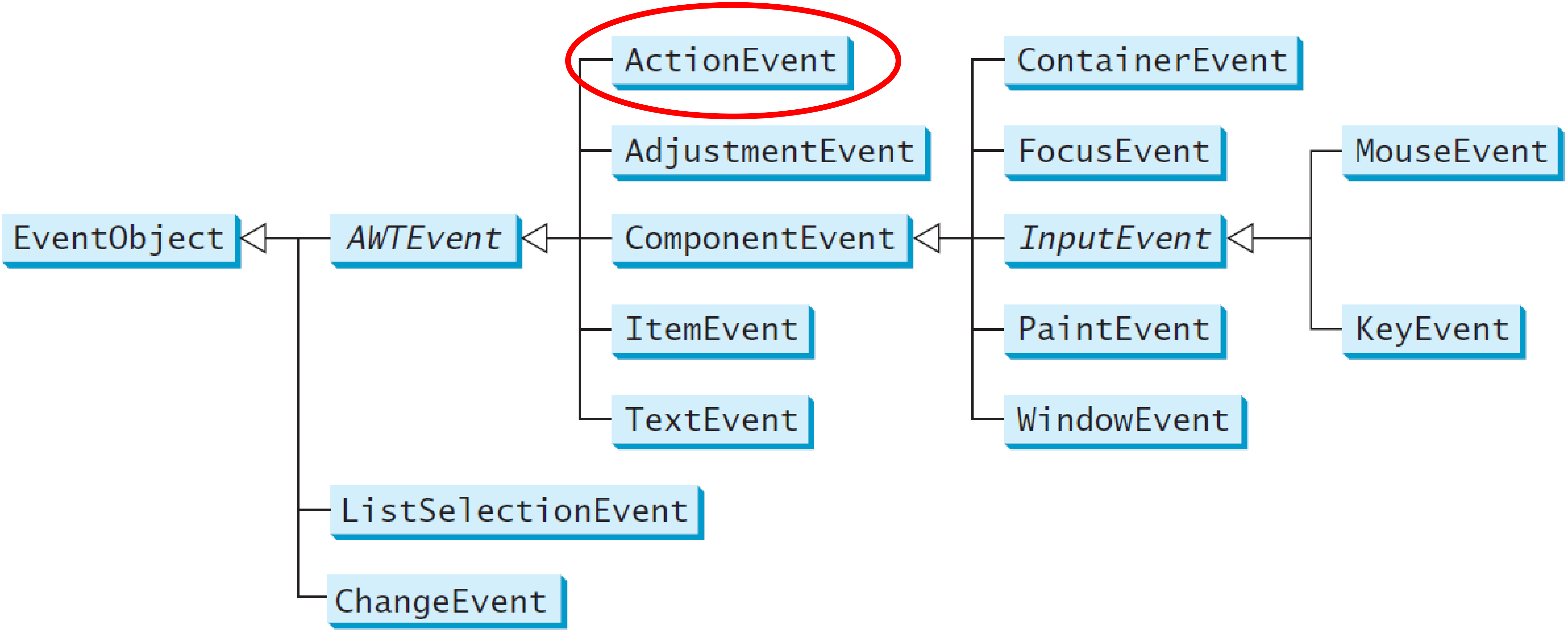
Events and Event Sources

- An event is an **object** created from an event source.
- **Firing an event** means to **create an event** and delegate the listener to **handle the event**.
- When you run a Java GUI program, the program interacts with the user, and the events drive its execution. This is called **event-driven programming**.
- An **event** can be defined as a **signal** to the program that something has happened.
- Events are triggered either by **external user actions**, such as mouse movements, button clicks, and keystrokes, or by internal program activities, such as a **timer**.

event source object

- The component that **creates** an event and **fires** it, is called the **event source object**, or simply **source object** or **source component**.
- For example, a **button** is the **source object** for a **buttonclicking action event**.
- The root class of the event classes is **java.util.EventObject**.

EventObject Hierarchy



Event objects

- An **event object** contains whatever properties are relevant to the event.
- You can identify the source object of an event using the **getSource()** instance method in the **EventObject** class.
- The subclasses of **EventObject** deal with specific types of events, such as action events, window events, component events, mouse events, and key events.
- For example, when **clicking a button**, the button creates and fires an **ActionEvent**.
- Here, the button is an **event source object** and an **ActionEvent** is the **event object** fired by the **source object**.

User Action, Source Object, Event Type, Listener Interface, and Handler

User Action	*Source Object*	*Event Type Fired*	*Listener Interface*	*Listener Interface Methods*
Click a button	`JButton`	`ActionEvent`	`ActionListener`	`actionPerformed(ActionEvent e)`
Press Enter in a text field	`JTextField`	`ActionEvent`	`ActionListener`	`actionPerformed(ActionEvent e)`

Select a new item	`JComboBox`	`ActionEvent`
		`ItemEvent`
			`ActionListener`
			`ItemListener`
				`actionPerformed(ActionEvent e)`
				`itemStateChanged(ItemEvent e)`

Check or uncheck	`JRadioButton`	`ActionEvent`
		`ItemEvent`
			`ActionListener`
			`ItemListener`
				`actionPerformed(ActionEvent e)`
				`itemStateChanged(ItemEvent e)`

Check or uncheck	`JCheckBox`	`ActionEvent`
		`ItemEvent`
			`ActionListener`
			`ItemListener`
				`actionPerformed(ActionEvent e)`
				`itemStateChanged(ItemEvent e)`

Mouse pressed	`Component`	`MouseEvent`	`MouseListener`	`mousePressed(MouseEvent e)`
Mouse released				`mouseReleased(MouseEvent e)`
Mouse clicked				`mouseClicked(MouseEvent e)`
Mouse entered				`mouseEntered(MouseEvent e)`
Mouse exited				`mouseExited(MouseEvent e)`
Mouse moved			`MouseMotionListener`	`mouseMoved(MouseEvent e)`
Mouse dragged				`mouseDragged(MouseEvent e)`
Key pressed	`Component`	`KeyEvent`	`KeyListener`	`keyPressed(KeyEvent e)`
Key released				`keyReleased(KeyEvent e)`
Key typed				`keyTyped(KeyEvent e)`

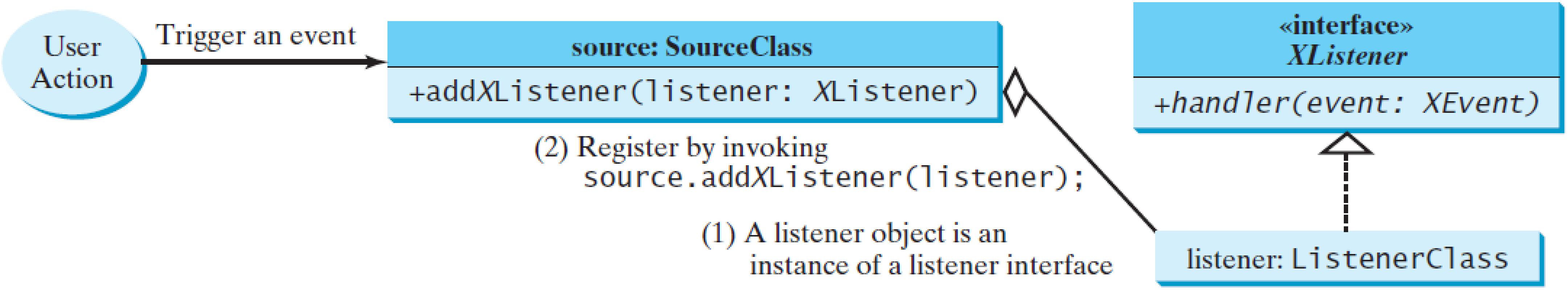
EventObject – Notes

- If a component can fire an event, any **subclass** of the component can fire the same type of event.
- For example, every GUI component can fire **MouseEvent** and **KeyEvent**, since **Component** is the superclass of all GUI components.

Listeners, Registrations, and Handling Events

- A listener is an object that **must be registered** with an event source object, and it must be an **instance** of an **appropriate** event-handling interface.
- Java uses a delegation-based model for event handling: a source object **fires** an event, and an **object** interested in the event **handles** it.
- This **object** is called an **event listener** or simply **listener**. For an object to be a listener for an event on a source object, two things are needed.

A listener must be an instance of a listener interface and must be registered with a source object.



User Action

Trigger an event

source: SourceClass

+addXListener(listener: XListener)

«interface»
XListener

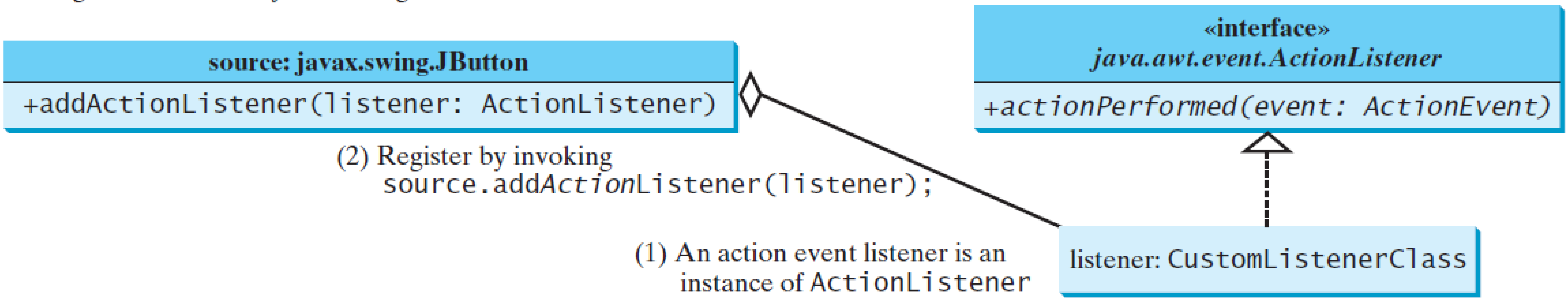
+*handler(event: XEvent)*

(2) Register by invoking
source.add*X*Listener(listener);

(1) A listener object is an
instance of a listener interface

listener: ListenerClass

A generic source object with a generic listener



source: javax.swing.JButton

+addActionListener(listener: ActionListener)

«interface»
java.awt.event.ActionListener

+*actionPerformed(event: ActionEvent)*

(2) Register by invoking
source.add*Action*Listener(listener);

(1) An action event listener is an
instance of ActionListener

listener: CustomListenerClass

A JButton source object with an ActionListener

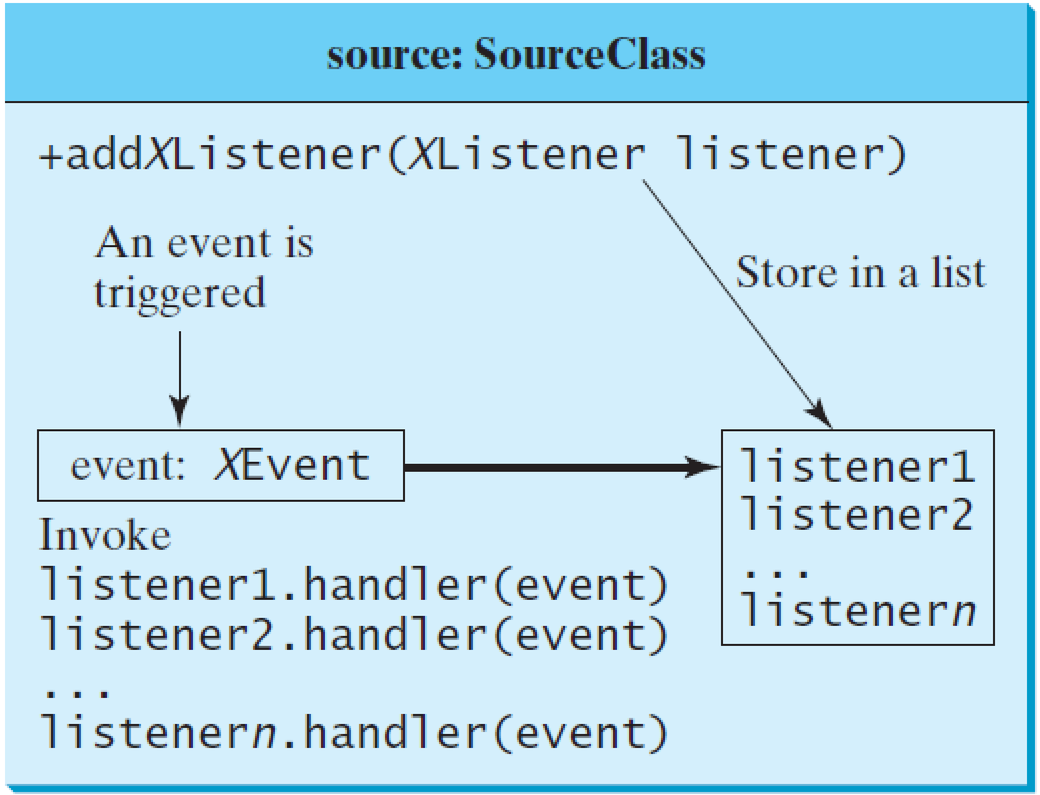
1st

- The listener object **must be** an instance of the corresponding event-listener interface to ensure that the listener has the **correct method** for processing the event.
- Java provides a listener interface for every type of event.
- The listener interface is usually named **XListener** for **XEvent**, with the exception of **MouseMotionListener**.
- The listener interface contains the **method(s)**, known as the *event handler(s)*, for processing the event.
- For example, the corresponding listener interface for **ActionEvent** is **ActionListener**; each listener for **ActionEvent** should **implement** the **ActionListener** interface; the **ActionListener** interface contains the **handler** **actionPerformed(ActionEvent)** for processing an **ActionEvent**.

2nd

- The listener object **must be registered** by the source object.
- Registration methods depend on the **event type**. For **ActionEvent**, the method is **addActionListener**. In general, the method is named **addXListener** for **XEvent**.
- A source object may fire **several** types of events, and for each event the source object maintains a **list of registered listeners** and notifies them by invoking the *handler* of the listener object to respond to the event. (See Next Slide)

The source object notifies the listeners of the event by invoking the listener object’s handler.



source: SourceClass

+add*X*Listener(*X*Listener listener)

An event is triggered

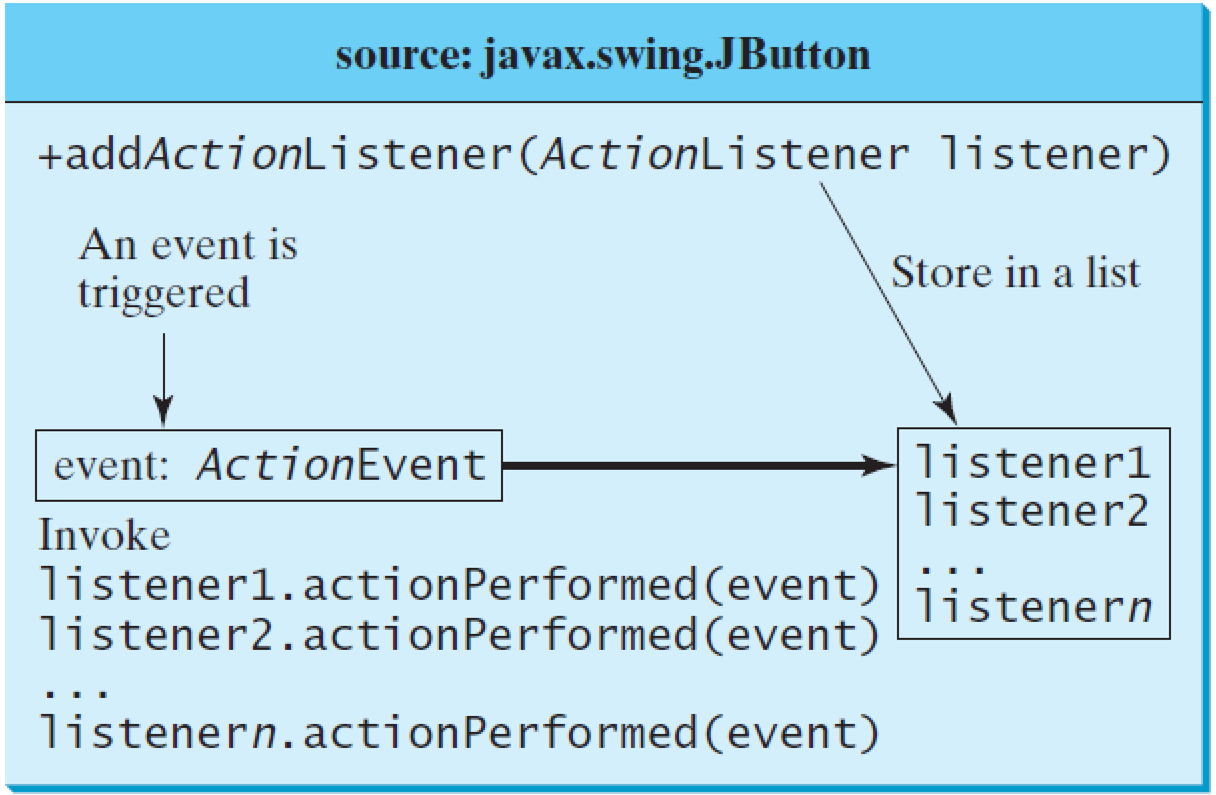
Store in a list

event: *X*Event

listener1
listener2
...
listener*n*

Invoke
listener1.handler(event)
listener2.handler(event)
...
listener*n*.handler(event)

Internal function of a generic source object



source: javax.swing.JButton

+add*Action*Listener(*Action*Listener listener)

An event is triggered

Store in a list

event: *Action*Event

listener1
listener2
...
listener*n*

Invoke
listener1.actionPerformed(event)
listener2.actionPerformed(event)
...
listener*n*.actionPerformed(event)

Internal function of a `JButton` object

Example

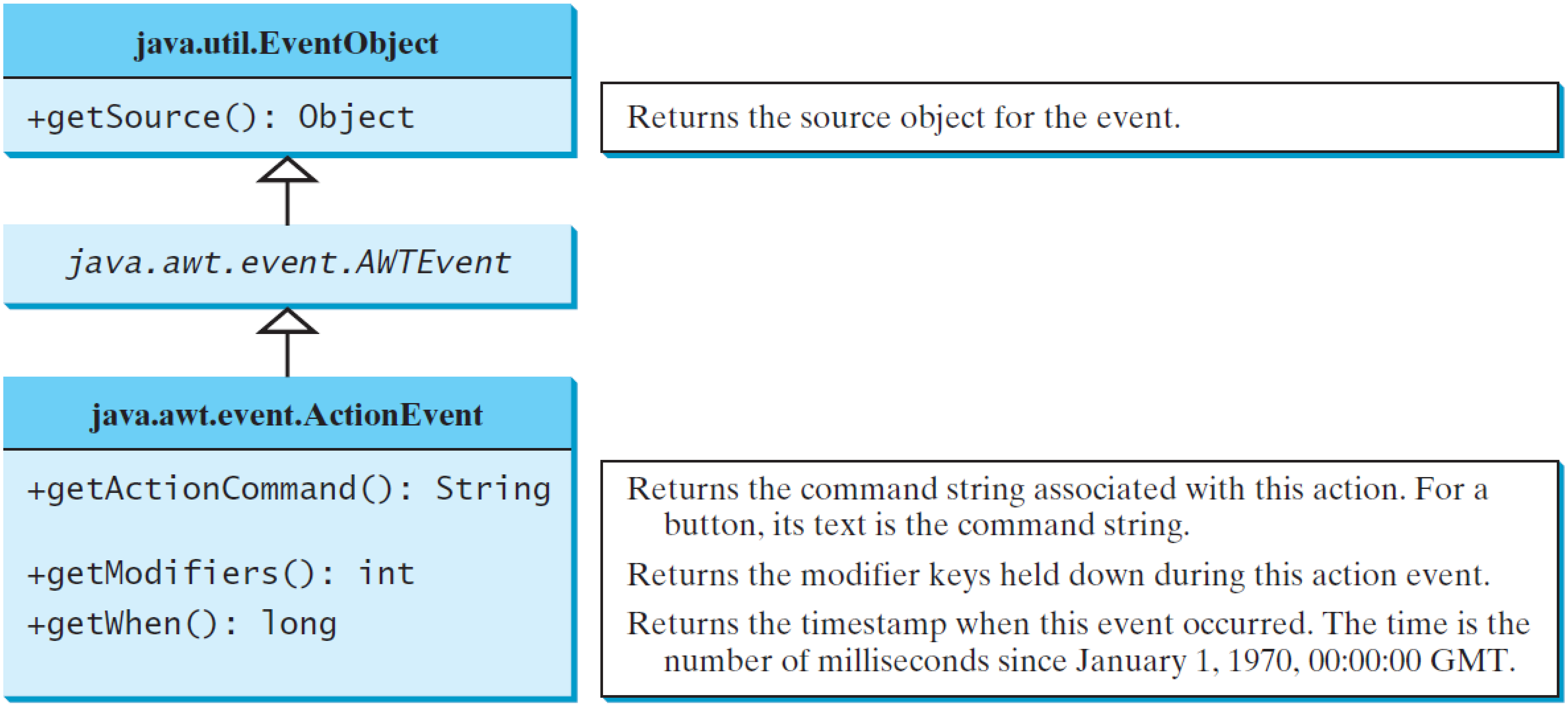
- Let’s revisit the first example, HandleEvent.java.
- Since a **JButton** object fires **ActionEvent**, a listener object for **ActionEvent** must be an instance of **ActionListener**, so the listener class **implements** **ActionListener**.
- The source object invokes **addActionListener(listener)** to **register** a listener, as follows:

```
JButton jbtOK = new JButton("OK");
OKListenerClass listener1  = new OKListenerClass();
jbtOK.addActionListener(listener1);
```

Events' Methods

- When you click the button, the **JButton** object fires an **ActionEvent** and passes it to invoke the listener's **actionPerformed** method to handle the event.
- The event object contains **information** relevant to the event, which can be obtained using the methods.
- For example, you can use **e.getSource()** to obtain the source object that fired the event.
- For an **action event**, you can use **e.getWhen()** to obtain the time when the event occurred.

You can obtain useful information from an event object.



java.util.EventObject

`+getSource(): Object`

Returns the source object for the event.

`java.awt.event.AWTEvent`

java.awt.event.ActionEvent

`+getActionCommand(): String`

`+getModifiers(): int`

`+getWhen(): long`

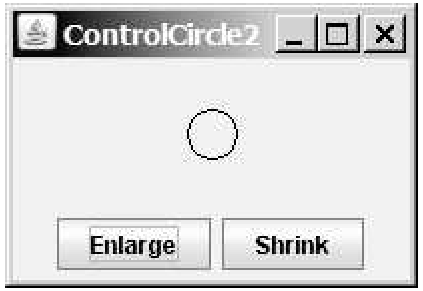
Returns the command string associated with this action. For a button, its text is the command string.

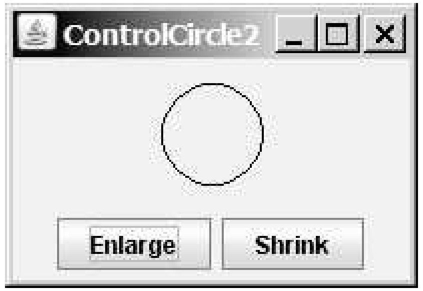
Returns the modifier keys held down during this action event.

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

Example

- We now write a program that uses two buttons to control the size of a circle.
- We will develop this program incrementally. First we will write the program that displays the user interface with a circle in the center and two buttons on the bottom.





We will develop this program incrementally. First we will write the program in that displays the user interface with a circle in the center and two buttons on the bottom

How do you use the buttons to enlarge or shrink the circle? When the Enlarge button is clicked,
you want the circle to be repainted with a larger radius. How can you accomplish this? You can expand the program with the following features:

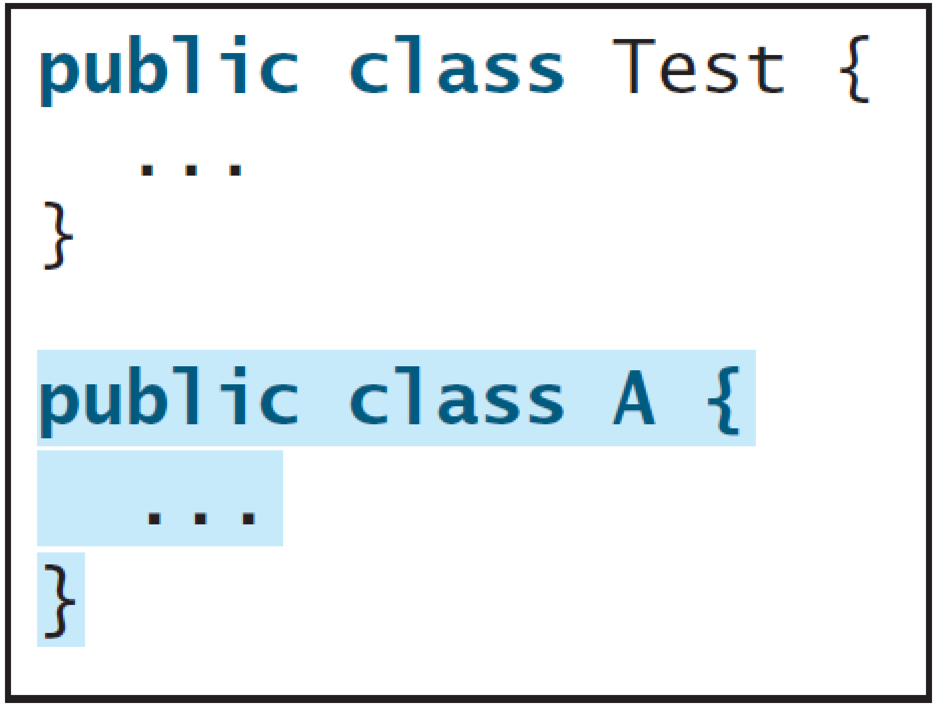
1. Define a listener class named **EnlargeListener** that implements **ActionListener**
2. Create a listener and register it with **jbtEnlarge**
3. Add a method named **enlarge()** in **CirclePanel** to increase the radius, then repaint the panel
4. Implement the **actionPerformed** method in **EnlargeListener** to invoke **canvas.enlarge()**
5. To make the reference variable **canvas** accessible from the **actionPerformed** method, define **EnlargeListener** as an inner class of the **ControlCircle** class

(*Inner classes* are defined inside another class. We will introduce inner classes in the next section.)

Inner Classes

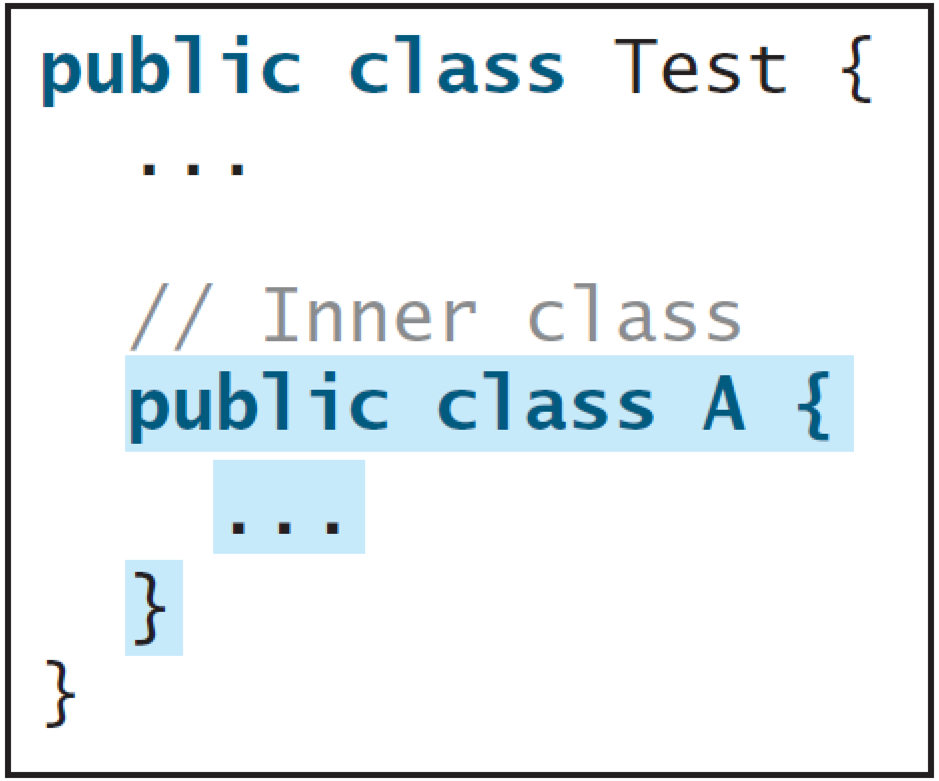
- An inner class, or nested class, is a class defined **within** the scope of another class.
- An inner class may be used just like a regular class.
- Normally, you define a class as an inner class if it is used **only** by its outer class.
- Inner classes are useful for defining listener classes.

The code in the left figure defines two separate classes, **Test** and **A**. The code in the right figure defines **A** as an inner class in **Test**.



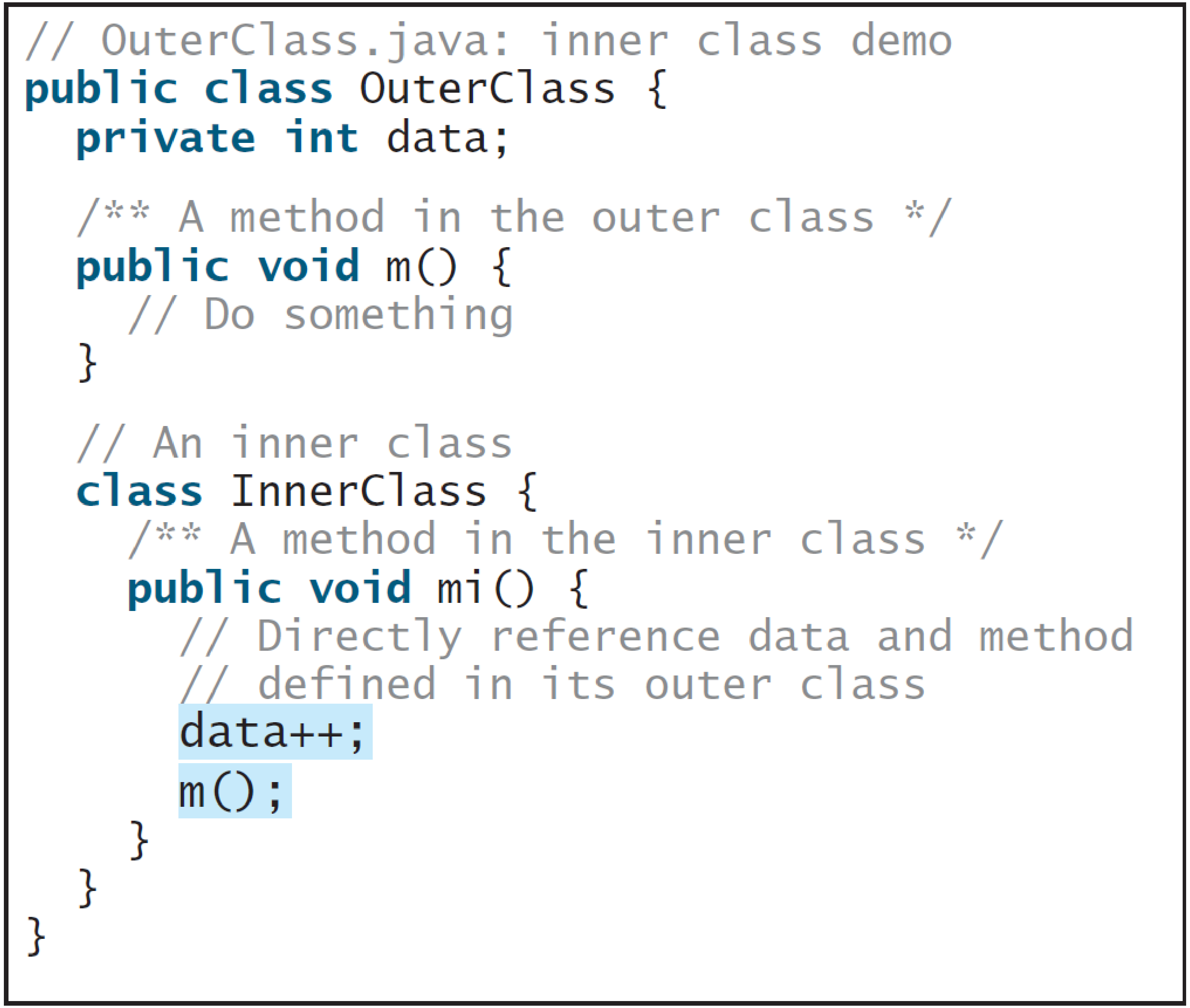
```
public class Test {
   ...
}

public class A {
   ...
}
```



```
public class Test {
   ...

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



```
// 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();
    }
  }
}
```

Inner Classes – features

- An inner class is compiled into a class named **OuterClassName$InnerClassName.class**.
- For example, the inner class **A** in **Test** is compiled into **Test$A.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.
- For example, **canvas** is defined in **ControlCircle**. It can be referenced in the inner class **EnlargeListener**.

Inner Classes – features (Cont.)

- A simple use of inner classes is to combine dependent classes into a primary class. This reduces the number of source files. It also makes class files easy to organize, since they are all named with the primary class as the prefix.
- For example, rather than creating the two source files **Test.java** and **A.java**, you can merge class **A** into class **Test** and create just one source file, **Test.java**. The resulting class files are **Test.class** and **Test$A.class**.

Inner Classes – features (Cont.)

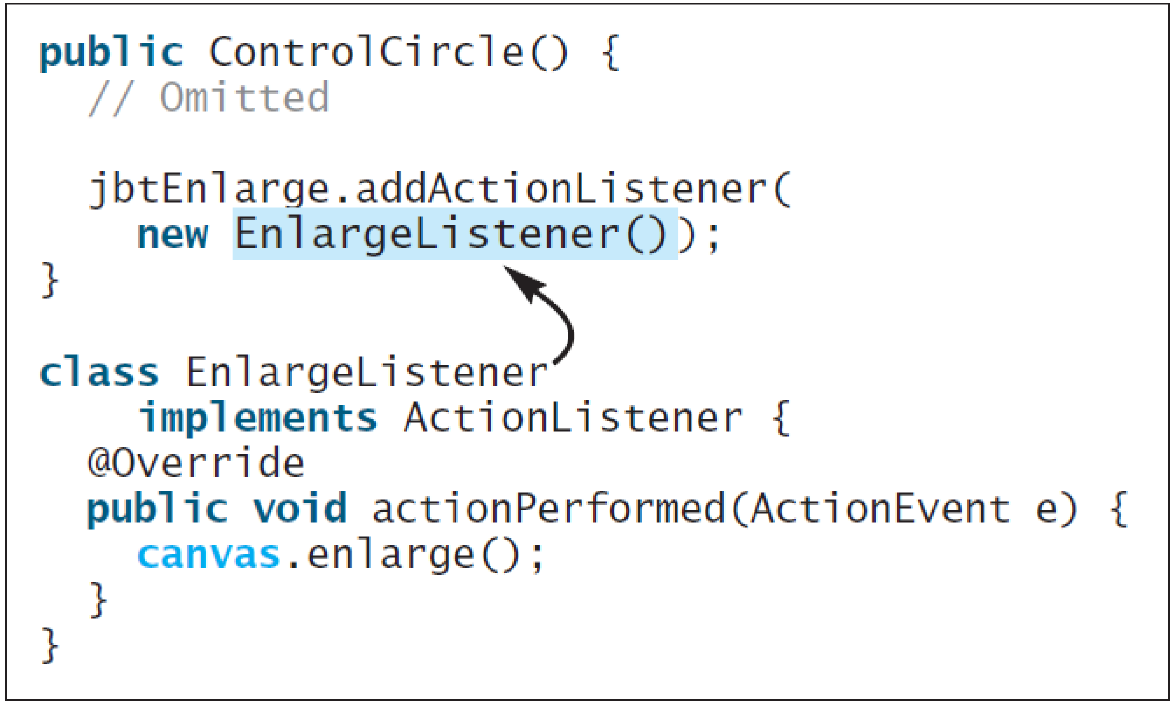
- Another practical use of inner classes is to avoid class-naming conflicts.
- Example: Two versions of **CirclePanel** are defined.
- A listener class is designed specifically to create a listener object for a GUI component (e.g., a button).
- The listener class will not be shared by other applications and therefore is appropriate to be defined inside the frame class as an inner class.

Anonymous Class Listeners

- An anonymous inner class is an inner class **without** a name. It combines defining an inner class and creating an instance of the class into one step.
- Inner-class listeners can be shortened using **anonymous inner classes**.
- Using an inner class or an anonymous inner class is preferred for defining listener classes.
- The syntax for an anonymous inner class is:

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

Anonymous Class Listeners – Example

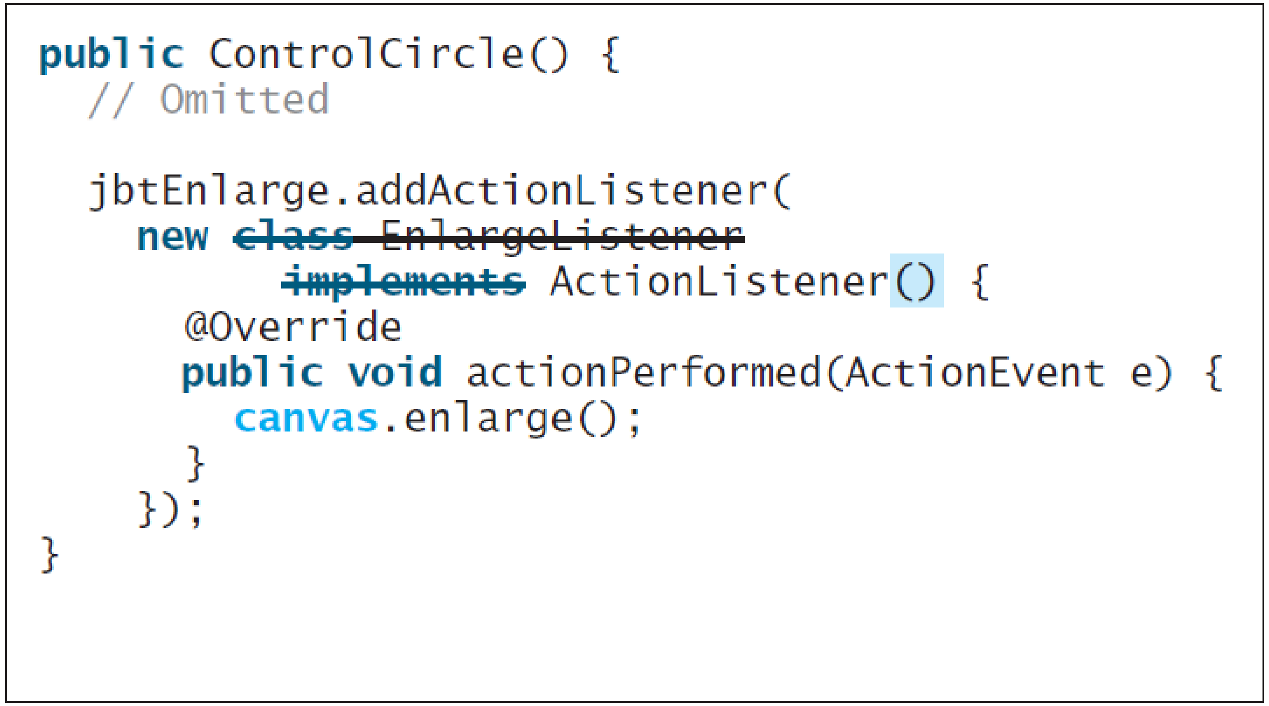


```
public ControlCircle() {
  // Omitted

  jbtEnlarge.addActionListener(
    new EnlargeListener());
}

class EnlargeListener
    implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    canvas.enlarge();
  }
}
```

Inner class EnlargeListener



```
public ControlCircle() {
  // Omitted

  jbtEnlarge.addActionListener(
    new ~~class EnlargeListener~~
        ~~implements~~ ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        canvas.enlarge();
      }
    });
}
```

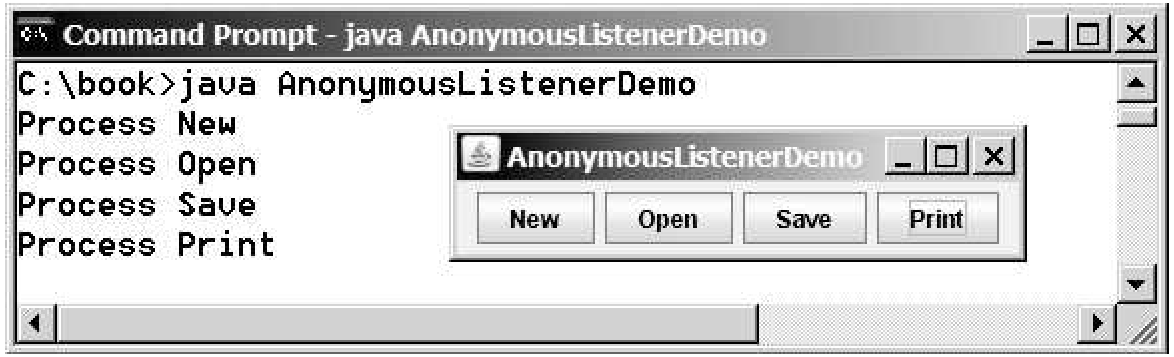
Anonymous inner class

Anonymous Inner Class Features

- 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 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, they are compiled into **Test$1.class** and **Test$2.class**.

Example

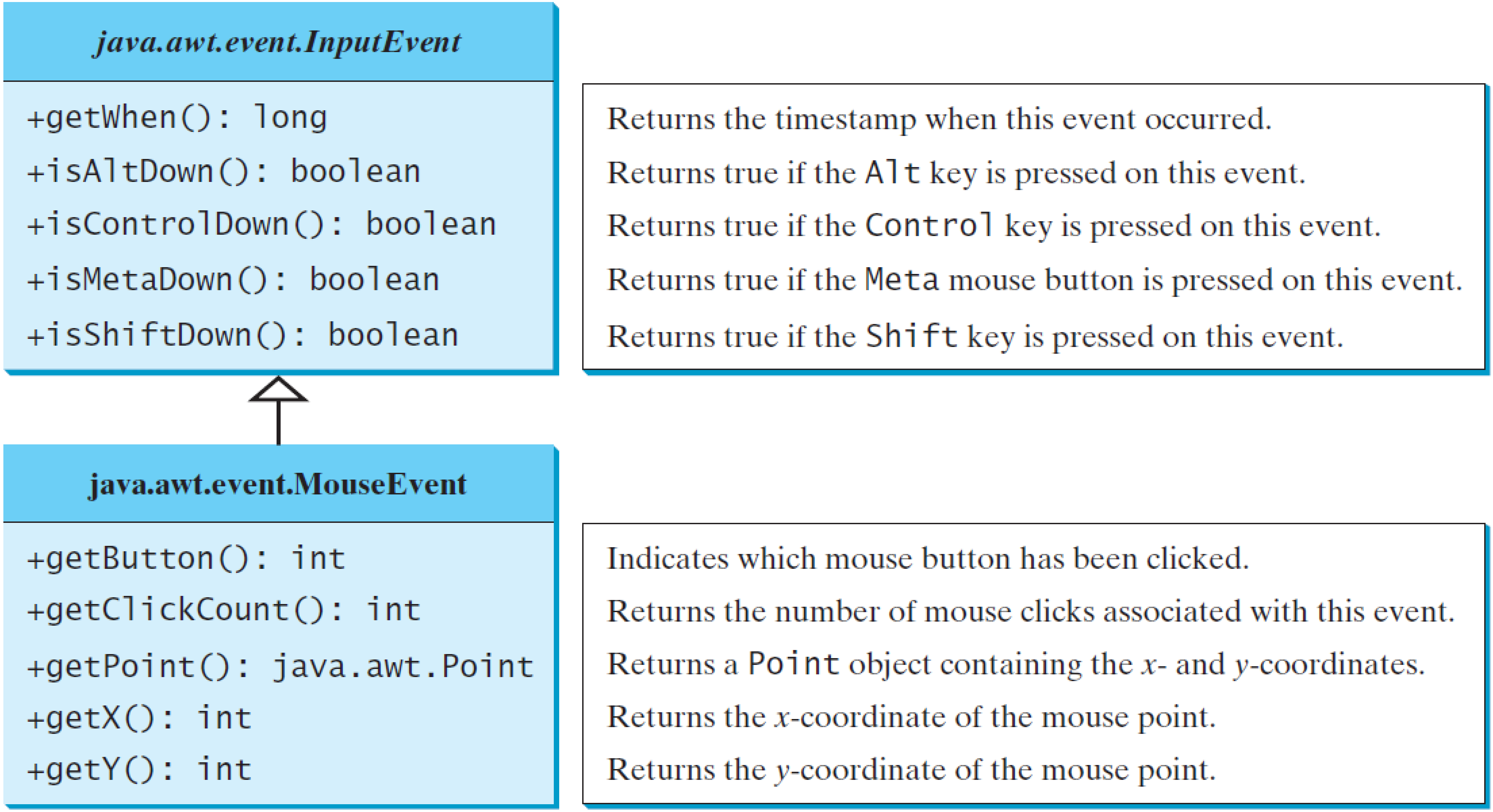
- Write a program that this program handles the events from four buttons.



```
C:\book>java AnonymousListenerDemo
Process New
Process Open
Process Save
Process Print
```

Mouse Events

- A mouse event is fired whenever a **mouse button** is pressed, released, or clicked, the mouse is moved, or the mouse is dragged onto a component.
- The **MouseEvent** object captures the event, such as the number of clicks associated with it, the location (the *x*- and *y*-coordinates) of the mouse, or which button was pressed.



java.awt.event.InputEvent

`+getWhen(): long`	Returns the timestamp when this event occurred.
`+isAltDown(): boolean`	Returns true if the `Alt` key is pressed on this event.
`+isControlDown(): boolean`	Returns true if the `Control` key is pressed on this event.
`+isMetaDown(): boolean`	Returns true if the `Meta` mouse button is pressed on this event.
`+isShiftDown(): boolean`	Returns true if the `Shift` key is pressed on this event.

java.awt.event.MouseEvent

`+getButton(): int`	Indicates which mouse button has been clicked.
`+getClickCount(): int`	Returns the number of mouse clicks associated with this event.
`+getPoint(): java.awt.Point`	Returns a `Point` object containing the *x*- and *y*-coordinates.
`+getX(): int`	Returns the *x*-coordinate of the mouse point.
`+getY(): int`	Returns the *y*-coordinate of the mouse point.

MouseEvent Methods

- Three **int** constants—**BUTTON1**, **BUTTON2**, and **BUTTON3**—are defined in **MouseEvent** to indicate the left, middle, and right mouse buttons.
- You can use the **getButton()** method to detect which button is pressed.
- For example, **getButton() == MouseEvent.BUTTON3** indicates that the right button was pressed.

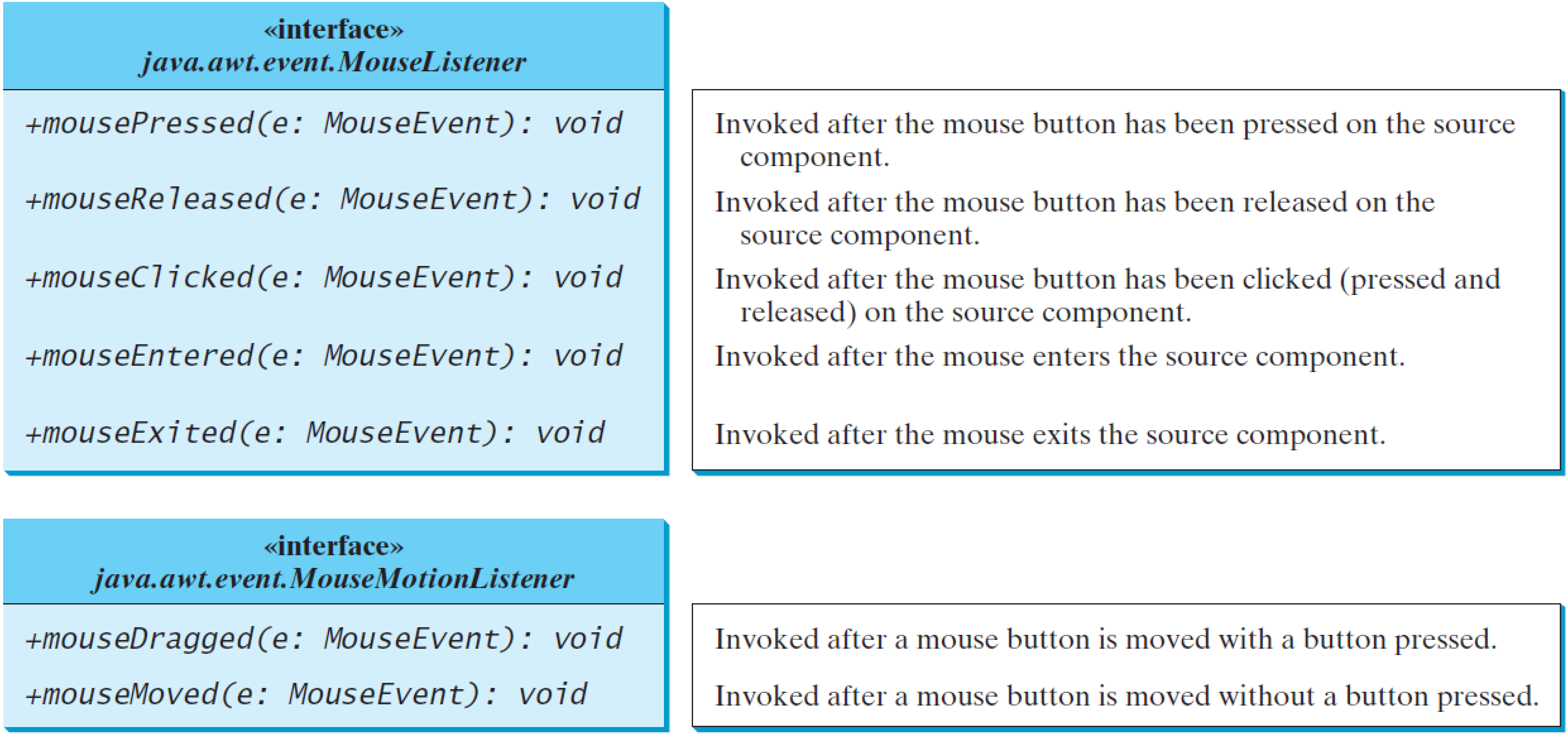
MouseEvent Methods (Cont.)

- The **java.awt.Point** class represents a point on a component.
- The class contains two public variables, **x** and **y**, for coordinates.
- To create a **Point**, use the following constructor:

  Point(**int** x, **int** y)
- This constructs a **Point** object with the specified *x*- and *y*-coordinates. Normally, the data fields in a class should be private, but this class has two public data fields.

Mouse Listener

- Java provides two listener interfaces, **MouseListener** and **MouseMotionListener**, to handle mouse events. (See Next Slide)
- Implement the **MouseListener** interface to listen for such actions as **pressing**, **releasing**, **entering**, **exiting**, or **clicking** the mouse.
- Implement the **MouseMotionListener** interface to listen for such actions as **dragging** or **moving** the mouse.



«interface»
java.awt.event.MouseListener

+mousePressed(e: MouseEvent): void — Invoked after the mouse button has been pressed on the source component.

+mouseReleased(e: MouseEvent): void — Invoked after the mouse button has been released on the source component.

+mouseClicked(e: MouseEvent): void — Invoked after the mouse button has been clicked (pressed and released) on the source component.

+mouseEntered(e: MouseEvent): void — Invoked after the mouse enters the source component.

+mouseExited(e: MouseEvent): void — Invoked after the mouse exits the source component.

«interface»
java.awt.event.MouseMotionListener

+mouseDragged(e: MouseEvent): void — Invoked after a mouse button is moved with a button pressed.

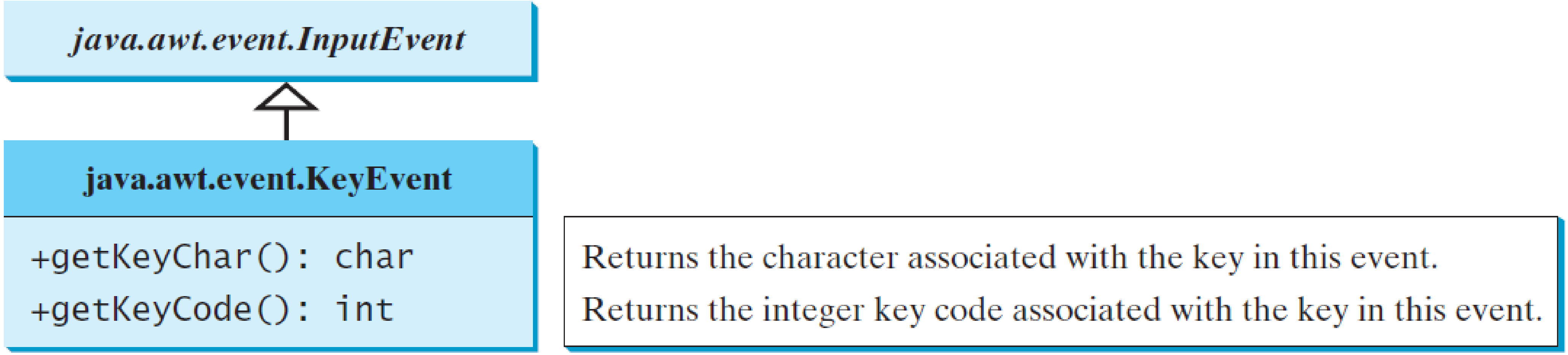
+mouseMoved(e: MouseEvent): void — Invoked after a mouse button is moved without a button pressed.

Mouse Events – Example

- To demonstrate using mouse events, let’s see an example that displays a message in a panel and enables the message to be moved using a mouse.
- The message moves as the mouse is dragged, and it is always displayed at the mouse point.

Key Events

- A key event is fired whenever a **key** is **pressed**, **released**, or **typed** *on a component*.
- *Key events* enable the use of the keys to control and perform actions or get input from the keyboard.
- The **KeyEvent** object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key.
- Java provides the **KeyListener** interface to handle key events.

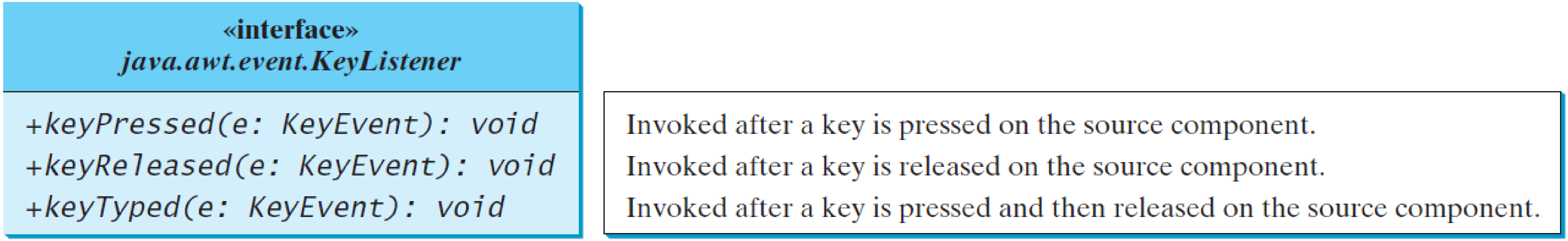


java.awt.event.InputEvent

java.awt.event.KeyEvent

+getKeyChar(): char
+getKeyCode(): int

Returns the character associated with the key in this event.
Returns the integer key code associated with the key in this event.

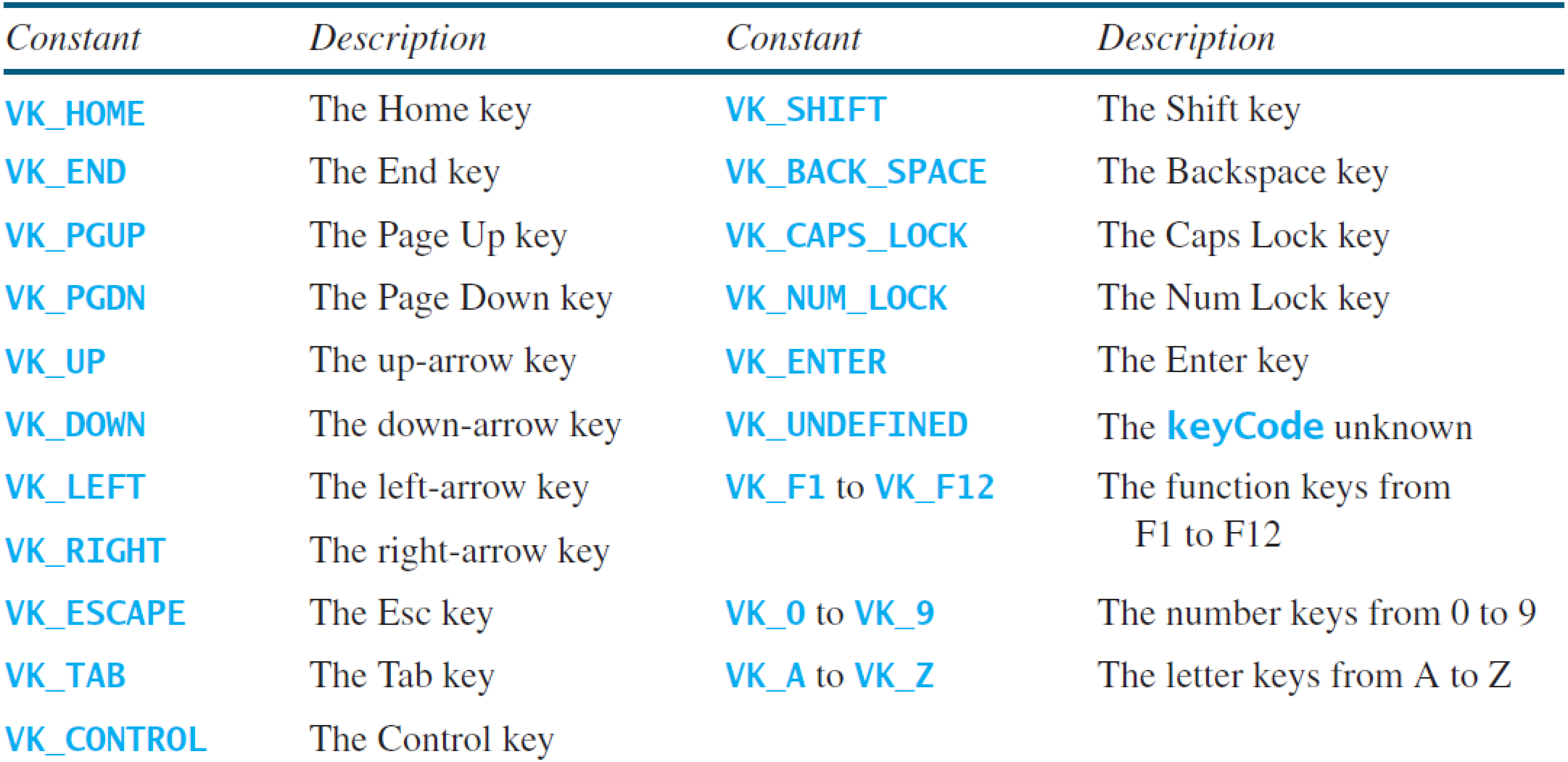


«interface»
java.awt.event.KeyListener

+keyPressed(e: KeyEvent): void
+keyReleased(e: KeyEvent): void
+keyTyped(e: KeyEvent): void

Invoked after a key is pressed on the source component.
Invoked after a key is released on the source component.
Invoked after a key is pressed and then released on the source component.

The key codes are constants defined in the KeyEvent class.



Constant	*Description*	*Constant*	*Description*
VK_HOME	The Home key	VK_SHIFT	The Shift key
VK_END	The End key	VK_BACK_SPACE	The Backspace key
VK_PGUP	The Page Up key	VK_CAPS_LOCK	The Caps Lock key
VK_PGDN	The Page Down key	VK_NUM_LOCK	The Num Lock key
VK_UP	The up-arrow key	VK_ENTER	The Enter key
VK_DOWN	The down-arrow key	VK_UNDEFINED	The keyCode unknown
VK_LEFT	The left-arrow key	VK_F1 to VK_F12	The function keys from F1 to F12
VK_RIGHT	The right-arrow key		
VK_ESCAPE	The Esc key	VK_0 to VK_9	The number keys from 0 to 9
VK_TAB	The Tab key	VK_A to VK_Z	The letter keys from A to Z
VK_CONTROL	The Control key		