GUIs recap:

// 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 Frameworks Recap.

- 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*
  - prone to platform-specific bugs because its peer-based approach relies heavily on the underlying platform
  - referred to as *heavyweight components*

- **Swing components**
  - more robust, versatile, and flexible
  - painted directly on canvases using Java code
    - exceptions: some Swing 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 that don’t rely on native GUI are referred to as *lightweight components*
GUI Class Hierarchy

- Object
  - Component
    - Graphics
    - Font
    - FontMetrics
    - Dimension
  - Container
    - LayoutManager
  - Color

- Classes in the java.awt package
  - Applet
    - JApplet
  - Frame
    - JFrame
  - Dialog
    - JDialog
  - Panel

- Swing Components in the javax.swing package
  - JComponent
  - Window

(c) Paul Fodor (CS Stony Brook) & Pearson
Container classes can contain other GUI components.
GUI Helper Classes

The helper classes are used to describe the properties of GUI components: graphics context, colors, fonts, and dimension.
AWT is like Swing
JFrame Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+JFrame()</code></td>
<td>Creates a default frame with no title.</td>
</tr>
<tr>
<td><code>+JFrame(title: String)</code></td>
<td>Creates a frame with the specified title.</td>
</tr>
<tr>
<td><code>+setSize(width: int, height: int): void</code></td>
<td>Specifies the size of the frame.</td>
</tr>
<tr>
<td><code>+setLocation(x: int, y: int): void</code></td>
<td>Specifies the upper-left corner location of the frame.</td>
</tr>
<tr>
<td><code>+setVisible(visible: boolean): void</code></td>
<td>Sets true to display the frame.</td>
</tr>
<tr>
<td><code>+setDefaultCloseOperation(mode: int): void</code></td>
<td>Specifies the operation when the frame is closed.</td>
</tr>
<tr>
<td><code>+setLocationRelativeTo(c: Component): void</code></td>
<td>Sets the location of the frame relative to the specified component. If the component is null, the frame is centered on the screen.</td>
</tr>
<tr>
<td><code>+pack(): void</code></td>
<td>Automatically sets the frame size to hold the components in the frame.</td>
</tr>
</tbody>
</table>
## The FlowLayout Class

**API:** [http://download.oracle.com/javase/7/docs/api/java/awt/FlowLayout.html](http://download.oracle.com/javase/7/docs/api/java/awt/FlowLayout.html)

<table>
<thead>
<tr>
<th>java.awt.FlowLayout</th>
<th>The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-alignment: int</td>
<td>The alignment of this layout manager (default: CENTER).</td>
</tr>
<tr>
<td>-hgap: int</td>
<td>The horizontal gap of this layout manager (default: 5 pixels).</td>
</tr>
<tr>
<td>-vgap: int</td>
<td>The vertical gap of this layout manager (default: 5 pixels).</td>
</tr>
<tr>
<td>+FlowLayout()</td>
<td>Creates a default FlowLayout manager.</td>
</tr>
<tr>
<td>+FlowLayout(alignment: int)</td>
<td>Creates a FlowLayout manager with a specified alignment.</td>
</tr>
</tbody>
</table>
# The GridLayout Class

The GridLayout class allows for the layout of components in a grid pattern. It is defined in the `java.awt` package. Here are its constructors:

- `GridLayout()`: Creates a default `GridLayout` manager.
- `GridLayout(rows: int, columns: int)`: Creates a `GridLayout` with a specified number of rows and columns.
- `GridLayout(rows: int, columns: int, hgap: int, vgap: int)`: Creates a `GridLayout` manager with a specified number of rows and columns, horizontal gap, and vertical gap.

The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity. You can find more details in the API documentation.

**API:** [http://download.oracle.com/javase/7/docs/api/java/awt/GridLayout.html](http://download.oracle.com/javase/7/docs/api/java/awt/GridLayout.html)

---

<table>
<thead>
<tr>
<th>java.awtGridLayout</th>
</tr>
</thead>
<tbody>
<tr>
<td>rows: int</td>
</tr>
<tr>
<td>columns: int</td>
</tr>
<tr>
<td>hgap: int</td>
</tr>
<tr>
<td>vgap: int</td>
</tr>
</tbody>
</table>

- The number of rows in this layout manager (default: 1).
- The number of columns in this layout manager (default: 1).
- The horizontal gap of this layout manager (default: 0).
- The vertical gap of this layout manager (default: 0).
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JFrame;
import java.awt.GridLayout;

public class ShowGridLayout extends JFrame {
    public ShowGridLayout() {
        // Set GridLayout, 3 rows, 2 columns, and gaps 5 between components horizontally and vertically
        setLayout(new GridLayout(3, 2, 5, 5));
        // Add labels and text fields to the frame
        add(new JLabel("First Name"));
        add(new JTextField(8));
        add(new JLabel("MI"));
        add(new JTextField(1));
        add(new JLabel("Last Name"));
        add(new JTextField(8));
    }

    public static void main(String[] args) {
        ShowGridLayout frame = new ShowGridLayout();
        frame.setTitle("ShowGridLayout");
        frame.setSize(200, 125);
        frame.setLocationRelativeTo(null); // Center the frame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
The **Color Class**

- **java.awt.Color**: 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) = **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);

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

Examples:

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);
Borders

- You can set a border on any object of the `JComponent`.
- To create a line border:
  ```java
  new LineBorder(Color color, int width)
  ```
  where `width` specifies the thickness of the line.
- To create a titled border:
  ```java
  new TitledBorder(Border border, String title)
  ```
- Example: displays a titled border on a panel
  ```java
  JPanel p0 = new JPanel();
  Border b1 = new LineBorder(Color.black, 2);
  Border b2 = new TitledBorder(b1, "Paul's Border");
  p0.setBorder(b2);
  ```
Image Icons

- Java uses the `javax.swing.ImageIcon` class to represent an icon.
  - Images are normally stored in image files
  - Use `new ImageIcon(filename)` to construct an image

```java
ImageIcon icon = new ImageIcon("image/us.gif");
frame.add(new JLabel(icon));
```
Procedural vs. Event-Driven Programming

• *Procedural programming* is executed in procedural order.

• In event-driven programming, *code is executed upon activation of events.*

  • Event sources and event classes,

  • Define listener classes (write the code to handle events),

  • Register listener objects with the source object.
Event Classes

EventObject

AWTEvent

ActionEvent

AdjustmentEvent

ComponentEvent

ItemEvent

TextEvent

ListSelectionEvent

ChangeEvent

ContainerEvent

FocusEvent

InputEvent

PaintEvent

WindowEvent

MouseEvent

KeyEvent
Event Information

• An event object contains whatever properties are pertinent to the event:
  • the source object of the event –
    • `getSource()` instance method in the `EventObject` superclass

• The subclasses of `EventObject` deal with special types of events, such as: button actions, window events, component events, mouse movements, and keystrokes
## Selected User Actions

<table>
<thead>
<tr>
<th>User Action</th>
<th>Source Object</th>
<th>Event Type Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click a button</td>
<td>JButton</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Click a check box</td>
<td>JCheckBox</td>
<td>ItemEvent, ActionEvent</td>
</tr>
<tr>
<td>Click a radio button</td>
<td>JRadioButton</td>
<td>ItemEvent, ActionEvent</td>
</tr>
<tr>
<td>Press return on a text field</td>
<td>JTextField</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Select a new item</td>
<td>JComboBox</td>
<td>ItemEvent, ActionEvent</td>
</tr>
<tr>
<td>Window opened, closed, etc.</td>
<td>Window</td>
<td>WindowEvent</td>
</tr>
<tr>
<td>Mouse pressed, released, etc.</td>
<td>Component</td>
<td>MouseEvent</td>
</tr>
<tr>
<td>Key released, pressed, etc.</td>
<td>Component</td>
<td>KeyEvent</td>
</tr>
</tbody>
</table>
The Delegation Model

When a user action occurs, an event is passed to every ActionListener object that registered to receive such events using the component's addActionListener method.

Trigger an event

(a) A generic source component with a generic listener

Example:

(b) A JButton source component with an ActionListener

Register by invoking source.addXListener(listener);
Internal Function of a Source Component

(a) Internal function of a generic source object

- source: SourceClass
  +addXListener(XListener listener)
  
  An event is triggered
  
  event: XEvent
  
  Invoke
  listener1.handler(event)
  listener2.handler(event)
  ...
  listennhandler(event)

  listener1
  listenn

(b) Internal function of a JButton object

- source: JButton
  +addActionListener(OnClickListener listener)
  
  An event is triggered
  
  event: ActionEvent
  
  Invoke
  listener1.actionPerformed(event)
  listener2.actionPerformed(event)
  ...
  listenn.actionPerformed(event)

  listener1
  listenn
### Selected Event Handlers

When a user action occurs, an event is passed to every ActionListener object that registered to receive such events using the component's addActionListener method. The handler method is executed.

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Listener Interface</th>
<th>Listener Methods (Handlers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>ActionListener</td>
<td>actionPerformed(ActionEvent)</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>ItemListener</td>
<td>itemStateChanged(ItemEvent)</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>WindowListener</td>
<td>windowClosing(WindowEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowOpened(WindowEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowIconified(WindowEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowDeiconified(WindowEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowClosed(WindowEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowActivated(WindowEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowDeactivated(WindowEvent)</td>
</tr>
<tr>
<td>ContainerEvent</td>
<td>ContainerListener</td>
<td>componentAdded(ContainerEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>componentRemoved(ContainerEvent)</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>MouseListener</td>
<td>mousePressed(MouseEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseReleased(MouseEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseClicked(MouseEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseExited(MouseEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseEntered(MouseEvent)</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>KeyListener</td>
<td>keyPressed(KeyEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>keyReleased(KeyEvent)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>keyTyped(KeyEvent)</td>
</tr>
</tbody>
</table>

(c) Paul Fodor (CS Stony Brook) & Pearson
**java.awt.event.ActionEvent**

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.

---

### java.util.EventObject

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+getSource(): Object</td>
<td>Returns the object on which the event initially occurred.</td>
</tr>
</tbody>
</table>

### java.awt.event.AWTEvent

### java.awt.event.ActionEvent

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+getActionCommand(): String</td>
<td>Returns the command string associated with this action. For a button, its text is the command string.</td>
</tr>
<tr>
<td>+getModifiers(): int</td>
<td>Returns the modifier keys held down during this action event.</td>
</tr>
<tr>
<td>+getWhen(): long</td>
<td>Returns the timestamp when this event occurred. The time is the number of milliseconds since January 1, 1970, 00:00:00 GMT.</td>
</tr>
</tbody>
</table>
One standard Event Processing Model: 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.
- 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

```java
// OuterClass.java: inner class demo
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();
        }
    }
}
```
import java.awt.*; import java.awt.event.*; import javax.swing.*
import javax.swing.border.*;

public class LoanCalculator extends JFrame {
    private JTextField jtfAnnualInterestRate = new JTextField();
    private JTextField jtfNumberOfYears = new JTextField();
    private JTextField jtfLoanAmount = new JTextField();
    private JTextField jtfMonthlyPayment = new JTextField();
    private JTextField jtfTotalPayment = new JTextField();
    private JButton jbtComputeLoan = new JButton("Compute Payment");

    public LoanCalculator() {
        JPanel p1 = new JPanel(new GridLayout(5, 2));
        p1.add(new JLabel("Annual Interest Rate"));
        p1.add(jtfAnnualInterestRate);
        p1.add(new JLabel("Number of Years"));
        p1.add(jtfNumberOfYears);
        p1.add(new JLabel("Loan Amount"));
        p1.add(jtfLoanAmount);
        p1.add(new JLabel("Monthly Payment"));
        p1.add(jtfMonthlyPayment);
    }
}
p1.add(new JLabel("Total Payment"));
p1.add(jtfTotalPayment);
p1.setBorder(new
    TitledBorder("Enter loan amount, interest rate, and year");
JPanel p2 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
p2.add(jbtComputeLoan);
add(p1, BorderLayout.CENTER);
add(p2, BorderLayout.SOUTH);
    jbtComputeLoan.addActionListener(new ButtonListener());
}

private class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        double interest =
            Double.parseDouble(jtfAnnualInterestRate.getText());
        int year =
            Integer.parseInt(jtfNumberOfYears.getText());
        double loanAmount =
            Double.parseDouble(jtfLoanAmount.getText());
        Loan loan = new Loan(interest, year, loanAmount);
jtfMonthlyPayment.setText(String.format("%.2f",
    loan.getMonthlyPayment()));

jtfTotalPayment.setText(String.format("%.2f",
    loan.getTotalPayment()));

public static void main(String[] args) {
    LoanCalculator frame = new LoanCalculator();
    frame.pack();
    frame.setTitle("LoanCalculator");
    frame.setLocationRelativeTo(null); // Center the frame
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
}
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`.

- 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

```java
public class AnonymousListenerDemo extends JFrame {
    public AnonymousListenerDemo() {
        JButton jbtNew = new JButton("New");
        JPanel panel = new JPanel();
        panel.add(jbtNew);
        add(panel);
        // Create and register anonymous inner class listener
        jbtNew.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.out.println("Process New");
                }
            }
        );
    }
    public static void main(String[] args) {
        JFrame frame = new AnonymousListenerDemo();
        frame.setVisible(true);
    }
}
```
Single Listener Class for multiple buttons

```java
jbtNew.addActionListener(listener);
jbtOpen.addActionListener(listener);
jbtSave.addActionListener(listener);
jbtPrint.addActionListener(listener);
...

class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == jbtNew)
            System.out.println("Process New");
        else if (e.getSource() == jbtOpen)
            System.out.println("Process Open");
        else if (e.getSource() == jbtSave)
            System.out.println("Process Save");
        else if (e.getSource() == jbtPrint)
            System.out.println("Process Print");
    }
}
```
Frame as Listener

```java
public class FrameAsListenerDemo extends JFrame implements ActionListener {
    private JButton jbtNew = new JButton("New");
    private JButton jbtOpen = new JButton("Open");
    private JButton jbtSave = new JButton("Save");
    private JButton jbtPrint = new JButton("Print");
    public FrameAsListenerDemo() {
        JPanel panel = new JPanel();
        panel.add(jbtNew); panel.add(jbtOpen); panel.add(jbtSave); panel.add(jbtPrint);
        add(panel);
        jbtNew.addActionListener(this);
        jbtOpen.addActionListener(this); jbtSave.addActionListener(this);
        jbtPrint.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == jbtNew) System.out.println("Process New");
        else if (e.getSource() == jbtOpen) System.out.println("Process Open");
        else if (e.getSource() == jbtSave) System.out.println("Process Save");
        ...
    }
}
```
# MouseEvent

## java.awt.event.InputEvent

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+getWhen(): long</td>
<td>Returns the timestamp when this event occurred.</td>
</tr>
<tr>
<td>+isAltDown(): boolean</td>
<td>Returns whether or not the Alt modifier is down on this event.</td>
</tr>
<tr>
<td>+isControlDown(): boolean</td>
<td>Returns whether or not the Control modifier is down on this event.</td>
</tr>
<tr>
<td>+isMetaDown(): boolean</td>
<td>Returns whether or not the Meta modifier is down on this event.</td>
</tr>
<tr>
<td>+isShiftDown(): boolean</td>
<td>Returns whether or not the Shift modifier is down on this event.</td>
</tr>
</tbody>
</table>

## java.awt.event.MouseEvent

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+getButton(): int</td>
<td>Indicates which mouse button has been clicked.</td>
</tr>
<tr>
<td>+getClickCount(): int</td>
<td>Returns the number of mouse clicks associated with this event.</td>
</tr>
<tr>
<td>+getPoint(): java.awt.Point</td>
<td>Returns a <code>Point</code> object containing the x and y coordinates.</td>
</tr>
<tr>
<td>+getX(): int</td>
<td>Returns the x-coordinate of the mouse point.</td>
</tr>
<tr>
<td>+getY(): int</td>
<td>Returns the y-coordinate of the mouse point.</td>
</tr>
</tbody>
</table>
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
java.awt.event.MouseListener

+mousePressed(e: MouseEvent): void
+mouseReleased(e: MouseEvent): void
+mouseClicked(e: MouseEvent): void
+mouseEntered(e: MouseEvent): void
+mouseExited(e: MouseEvent): void
```

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

```java
java.awt.event.MouseMotionListener

+mouseDragged(e: MouseEvent): void
+mouseMoved(e: MouseEvent): void
```

Invoked when a mouse button is moved with a button pressed.
Invoked when a mouse button is moved without a button pressed.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MouseGUI_01 extends JFrame {

    MouseGUI_01()
    {
        MovableMessagePanel p =
            new MovableMessagePanel("Mouse GUI in Java");
        setLayout(new BorderLayout());
        add(p);
    }

    public static void main(String[] args) {
        MouseGUI_01 frame = new MouseGUI_01();
        frame.setTitle("MoveMessageDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 100);
        frame.setVisible(true);
    }
}

// (to continue next slide with the inner class MovableMessagePanel)
// Inner class: MovableMessagePanel draws a message
static class MovableMessagePanel extends JPanel {
    private String message = "Mouse GUI in Java";
    private int x = 20;
    private int y = 20;
    public MovableMessagePanel(String s) {
        message = s;
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                // Get the new location and repaints screen
                x = e.getX(); y = e.getY();
                repaint();
            }
        });
    }
    /** Paint the component */
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawString(message, x, y);
    }
}
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:**
  
  ```java
  getKeyChar() method
  getKeyCode() method
  ```

- **Key constants:**

  ```
  Home       VK_HOME
  End        VK_END
  Page Up    VK_PGUP
  Page Down  VK_PGDN
  etc...
  ```
The KeyEvent Class, cont.

java.awt.event.KeyEvent

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

Returns the character associated with the key in this event.
Returns the integer keyCode associated with the key in this event.
The **KeyEvent** Class, cont.

// KeyEventDemo example

![KeyEventDemo example](image-url)
// KeyEventDemo example

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class KeyEventDemo extends JFrame {
    private KeyboardPanel keyboardPanel = new KeyboardPanel();

    /** Initialize UI */
    public KeyEventDemo() {
        // Add the keyboard panel to accept and display user input
        add(keyboardPanel);

        // Set focus
        keyboardPanel.setFocusable(true);
    }

    /** Main method */
    public static void main(String[] args) {
        KeyEventDemo frame = new KeyEventDemo();
        frame.setTitle("KeyEventDemo");
        frame.setLocationRelativeTo(null); // Center the frame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
}
// Inner class: KeyboardPanel for receiving key input
static class KeyboardPanel extends JPanel {
    private int x = 100;
    private int y = 100;
    private char keyChar = 'A'; // Default key

    public KeyboardPanel() {
        addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                switch (e.getKeyCode()) {
                case KeyEvent.VK_DOWN: y += 10; break;
                case KeyEvent.VK_UP: y -= 10; break;
                case KeyEvent.VK_LEFT: x -= 10; break;
                case KeyEvent.VK_RIGHT: x += 10; break;
                default: keyChar = e.getKeyChar();
                }
                repaint();
            }
        });
    }

    /** Draw the character */
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.setFont(new Font("TimesRoman", Font.PLAIN, 24));
        g.drawString(String.valueOf(keyChar), x, y);
    }
}

(c) Paul Fodor (CS Stony Brook) & Pearson
The Timer Class

- Some non-GUI components can fire events:
  - The `javax.swing.Timer` class is a source component that fires an `ActionEvent` at a predefined rate

```
javax.swing.Timer
+Timer(delay: int, listener: ActionListener)
+addActionListener(listener: ActionListener): void
+start(): void
+stop(): void
+setDelay(delay: int): void
```

- Creates a Timer with a specified delay in milliseconds and an ActionListener.
- Adds an ActionListener to the timer.
- Starts this timer.
- Stops this timer.
- Sets a new delay value for this timer.

- The `Timer` class can be used to control animations
The Timer Class

AnimationDemo Example:

```
message moving?

2410!
```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  

public class AnimationDemo extends JFrame {  

    public AnimationDemo() {  
        // Create a MovingMessagePanel for displaying a moving message  
        this.setLayout(new GridLayout(2, 1));  
        add(new MovingMessagePanel("message moving?", 1000));  
        add(new MovingMessagePanel("2410!", 500));  
    }  

    public static void main(String[] args) {  
        AnimationDemo frame = new AnimationDemo();  
        frame.setTitle("AnimationDemo");  
        frame.setLocationRelativeTo(null); // Center the frame  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.setSize(280, 100);  
        frame.setVisible(true);  
    }  
}
// Inner class: Displaying a moving message
static class MovingMessagePanel extends JPanel {
  private String message = "Welcome to Java";
  private int xCoordinate = 0;
  private int yCoordinate = 20;

  public MovingMessagePanel(String message, int delay) {
    this.message = message;
    // Create a timer
    Timer timer = new Timer(delay, new TimerListener());
    timer.start();
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (xCoordinate > getWidth()) {
      xCoordinate = -20;
    }
    xCoordinate += 5;
    g.drawString(message, xCoordinate, yCoordinate);
  }

  class TimerListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      repaint();
    }
  }
}