Swing component hierarchy

- Graphical components in Java form an inheritance hierarchy:
  ```java
  java.lang.Object
    +- java.awt.Component
      +- java.awt.Container
        +- javax.swing.JComponent
          +- javax.swing.JLabel
          +- javax.swing.JMenuBar
          +- javax.swing.JOptionPane
          +- javax.swing.JTextField
            +- java.awt.Window
              +- java.awt.Frame
              +- javax.swing.JFrame
  ```

- When doing GUI programming, always import these packages:
  ```java
  import java.awt.*;
  import javax.swing.*;
  ```

Java GUI: AWT and Swing

- Sun’s initial idea: create a set of classes/methods that can be used to write a multi-platform GUI (Abstract Windowing Toolkit, or AWT)
  - problem: not powerful enough; limited; a bit clunky to use

- Second edition (JDK v1.2): Swing
  - a newer library written from the ground up that allows much more powerful graphics and GUI construction

An option pane is a simple dialog box for graphical input/output

- advantages:
  - simple
  - flexible (in some ways)
  - looks better than the black box of death

- disadvantages:
  - created with static methods
  - not very object-oriented
  - not very powerful (just simple dialog boxes)
Types of JOptionPane

- showMessageDialog(<parent>, <message>)
  Displays a message on a dialog with an OK button.

- showConfirmDialog(<parent>, <message>)
  Displays a message and list of choices Yes, No, Cancel: returns user’s choice as an int with one of the following values:
  - JOptionPane.YES_OPTION
  - JOptionPane.NO_OPTION
  - JOptionPane.CANCEL_OPTION

- showInputDialog(<parent>, <message>)
  Displays a message and text field for input; returns the user’s value entered as a String.
  - can pass null for the parent to all methods

JOptionPane examples 1

- showMessageDialog analogous to System.out.println to display a message

```java
import javax.swing.*;
public class MessageDialogExample {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "How's the weather?");
        JOptionPane.showMessageDialog(null, "Second message");
    }
}
```

JOptionPane examples 2

- showConfirmDialog analogous to a System.out.print that prints a question, then reading an input value from the user (can only be one of the provided choices)

```java
import javax.swing.*;
public class ConfirmDialogExample {
    public static void main(String[] args) {
        int choice = JOptionPane.showConfirmDialog(null, "Erase your hard disk?");
        if (choice == JOptionPane.YES_OPTION) {
            JOptionPane.showMessageDialog(null, "Disk erased!");
        } else {
            JOptionPane.showMessageDialog(null, "Cancelled.");
        }
    }
}
```

JOptionPane examples 3

- showInputDialog analogous to a System.out.print that prints a question, then reading an input value from the user (can be any value)

```java
import javax.swing.*;
public class InputDialogExample {
    public static void main(String[] args) {
        String name = JOptionPane.showInputDialog(null, "What's yer name, pardner?");
        JOptionPane.showMessageDialog(null, "Yeehaw, " + name);
    }
}
```
Onscreen GUI elements

- Most GUIs are not composed of option panes: they are too limited. Instead, complex GUIs contain the following elements:
  - frame: A graphical window on the screen.
  - components: GUI widgets such as buttons or text fields.
  - containers: Logical groups of components.

```java
import javax.swing.*;
public class SimpleFrame {
public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.setVisible(true);
}
}
```

Graphical output:

JFrame example 1

- A simple program that creates and shows a JFrame:
  ```java
  import javax.swing.*;
public class SimpleFrame {
p    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setVisible(true);
    }
}
  ```

Graphical output:

JFrame example 2

```java
import java.awt.*;
import javax.swing.*;
public class SimpleFrame2 {
p    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setForeground(Color.WHITE);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLocation(new Point(10, 50));
        frame.setSize(new Dimension(300, 120));
        frame.setTitle("A frame");
        frame.setVisible(true);
    }
}
```

Graphical output:

JFrame properties

- JFrames have the following properties that you can get/set:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>default close operation</td>
<td>int</td>
<td>what should happen when frame is closed</td>
<td>getDefaultCloseOperation, setDefaultCloseOperation</td>
</tr>
<tr>
<td>icon image</td>
<td>Image</td>
<td>icon in the window’s title bar</td>
<td>getIconImage, setIconImage</td>
</tr>
<tr>
<td>layout</td>
<td>LayoutManager</td>
<td>how the frame should position its components</td>
<td>getLayout, setLayout</td>
</tr>
<tr>
<td>resizable</td>
<td>boolean</td>
<td>whether the window can be resized</td>
<td>isResizable, setResizable</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
<td>window’s title bar text</td>
<td>getTitle, setTitle</td>
</tr>
</tbody>
</table>
### Component properties

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>Color</td>
<td>background color</td>
<td>getBackground, setBackground</td>
</tr>
<tr>
<td>enabled</td>
<td>boolean</td>
<td>whether the component can be interacted with</td>
<td>isEnabled, setEnabled</td>
</tr>
<tr>
<td>font</td>
<td>Font</td>
<td>font used to display any text on the component</td>
<td>getFont, setFont</td>
</tr>
<tr>
<td>foreground</td>
<td>Color</td>
<td>foreground color</td>
<td>getForeground, setForeground</td>
</tr>
<tr>
<td>location</td>
<td>Point</td>
<td>(x, y) position of component on screen</td>
<td>getLocation, setLocation</td>
</tr>
<tr>
<td>size</td>
<td>Dimension</td>
<td>width, height of component</td>
<td>getSize, setSize</td>
</tr>
<tr>
<td>preferred size</td>
<td>Dimension</td>
<td>width, height that the component wants to be</td>
<td>getPreferredSize, setPreferredSize</td>
</tr>
<tr>
<td>visible</td>
<td>boolean</td>
<td>whether the component can be seen on screen</td>
<td>isVisible, setVisible</td>
</tr>
</tbody>
</table>

### JFrame

A frame is a graphical window that can be used to hold other components.

- public JFrame()
- public JFrame(String title)
  Creates a frame with an optional title.
- public void setTitle(String text)
  Puts the given text in the frame's title bar.
- public void setDefaultCloseOperation(int op)
  Makes the frame perform the given action when it closes. Common value: JFrame.EXIT_ON_CLOSE
- public void add(Component comp)
  Places the given component or container inside the frame.

**NOTE**: Call setVisible(true) to make a frame appear on screen after creating it.

### JButton, JLabel

The most common component—a button is a clickable onscreen region that the user interacts with to perform a single command.

A text label is simply a string of text displayed on screen in a graphical program. Labels often give information or describe other components.

- public JButton(String text)
- public JLabel(String text)
  Creates a new button / label with the given string as its text.
- public String getText()
  Returns the text showing on the button / label.
- public void setText(String text)
  Sets button / label's text to be the given string.

### JTextField, JTextArea

A text field is like a label, except that the text in it can be edited and modified by the user. Text fields are commonly used for user input, where the user types information in the field and the program reads it.

A text area is a multi-line text field.

- public JTextField(int columns)
- public JTextArea(int lines, int columns)
  Creates a new text field the given number of columns (letters) wide.
- public String getText()
  Returns the text currently in the field.
- public void setText(String text)
  Sets field's text to be the given string.
Action events with ActionListener

Event-driven programming

- Program's execution is indeterminate
- On-screen components cause events to occur when they are clicked / interacted with
- Events can be handled, causing the program to respond, driving the execution thru events (an "event-driven" program)

Java GUI Event Handling

- If a component wants to respond to a given type of events, the component should have a "delegate" (listener object) for that type of events
- When an event of that type happened, the Java virtual machine will invoke the method of the listener object
  - The listener provides callback

Java GUI Event Handling

- To make sure that an installed listener object can handle a type of events, it should implement the listener interface for that type of event
  - Major advantage of using interface:
    - It does not uniquely define a single class of objects to handle a type of events, any class implementing the interface works.
Java Event Hierarchy

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

Action events: `ActionEvent`

- most common / simple event type in Swing
- represent an action occurring on a GUI component

- created by:
  - button clicks
  - check box checking / unchecking
  - menu clicks
  - pressing Enter in a text field
  - etc.

Listening for events

- attach a `listener` to the component
- listener’s appropriate method will be called when event occurs (e.g. when the button is clicked)

- for Action events, use `ActionListener`

```java
// part of Java; you don't write this
public interface ActionListener {
  public void actionPerformed(ActionEvent event);
}

// Prints a message when the button is clicked.
public class MyActionListener implements ActionListener {
  public void actionPerformed(ActionEvent event)
  { ...
    
  }
}
```
**Attaching an ActionListener**

```java
JButton button = new JButton("button 1");
ActionListener listener = new MyActionListener();
button.addActionListener(listener);
```

- now `button` will print "Event occurred!" when clicked
- `addActionListener` method exists in many Swing components

---

**MouseListener Interface**

- Methods in the MouseListener interface:
  - `void mousePressed (MouseEvent event);`
    - called when the mouse button is pressed down.
  - `void mouseReleased (MouseEvent event);`
    - called when the mouse button is released.
  - `void mouseClicked (MouseEvent event);`
    - called if the mouse button is pressed & released at the same location.
  - `void mouseEntered (MouseEvent event);`
    - called when the mouse pointer passes into a component
  - `void mouseExited (MouseEvent event);`
    - called when the mouse pointer passes out of a component

---

**MouseMotionListener**

- Methods in the MouseMotionListener interface:
  - `void mouseDragged (MouseEvent event);`
    - called when the mouse button is pressed on a component and then dragged
  - `void mouseMoved (MouseEvent event);`
    - called when the mouse button has been moved on a component (with no buttons down)

---

**Handling Events: Example**

- For example, after a user clicks on top of a button component,
  - if the button component has a Mouse listener object, its `mousePressed` method will be automatically called
  - the mouse event is also translated to an `ActionEvent` object
  - if the button has a listener object for `ActionEvent`, the method `actionPerformed` of the listener object is automatically invoked
Example: InvestmentViewer

- Add a button so that each time the user clicks on the button, $10 is added to the bank account

See BankAccount.java; InvestmentFrame.java; InvestmentMain.java

Problem: position, resize

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

- Absolute positioning (C++, C#, others):
  Specify exact pixel coordinates for every component

- Layout managers (Java):
  Have special objects that decide where to position each component based on some criteria
  - What are benefits or drawbacks to each approach?

Layout managers

- Here are several common Java layout managers:
Containers

- **Container**: An object that holds components; it also governs their positions, sizes, and resizing behavior.

- Containers have the following public methods:
  - `public void add(Component comp)`
  - `public void add(Component comp, Object info)`
    Adds a component to the container, possibly giving extra information about where to place it.
  - `public void remove(Component comp)`
    Removes the given component from the container.
  - `public void setLayout(LayoutManager mgr)`
    Uses the given layout manager to position the components in the container.
  - `public void validate()`
    You should call this if you change the contents of a container that is already on the screen, to make it re-do its layout.

JPanel

- A panel is our container of choice; it provides the methods from the previous slide and defines these additional methods (among others):
  - `public JPanel()`
    Constructs a panel with a default flow layout.
  - `public JPanel(LayoutManager mgr)`
    Constructs a panel that uses the given layout manager.

Preferred size

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

Buttons at preferred size:

```
  +---+---+
  |   |   |
  +---+---+
      |   |
      +---+
```

Not preferred size:

```
  +---+---+
  |   |   |
  +---+---+
      |   |
      +---+
```

FlowLayout

- `public FlowLayout()`
- treats container as a left-to-right, top-to-bottom "page" or "paragraph"
- components are given their preferred size both horizontally and vertically
- components are positioned in order added
- if too long, components wrap around to next line

```java
Container panel = new JPanel(new FlowLayout());
panel.add(new JButton("Button 1"));
```
FlowLayout example

import java.awt.*;
import javax.swing.*;
public class FlowLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(320, 75));
        frame.setTitle("Flow layout");
        frame.setLayout(new FlowLayout());
        frame.add(new JLabel("Type your ZIP Code: "));
        frame.add(new JTextField(5));
        frame.add(new JButton("Submit"));
        frame.setVisible(true);
    }
}

GridLayout example

import java.awt.*;
import javax.swing.*;
public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(300, 120));
        frame.setTitle("The grid");
        // 2 rows, 3 columns
        frame.setLayout(new GridLayout(2, 3));
        for (int i = 1; i <= 6; i++) {
            JButton button = new JButton();
            button.setText("Button " + i);
            frame.add(button);
        }
        frame.setVisible(true);
    }
}

GridLayout example

import java.awt.*;
import javax.swing.*;
public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(300, 120));
        frame.setTitle("The grid");
        // 2 rows, 3 columns
        frame.setLayout(new GridLayout(2, 3));
        for (int i = 1; i <= 6; i++) {
            JButton button = new JButton();
            button.setText("Button " + i);
            frame.add(button);
        }
        frame.setVisible(true);
    }
}

GridLayout

public GridLayout(int rows, int columns)
- treats container as a grid of equally-sized rows and columns
- components are given equal horizontal / vertical size, disregarding preferred size
- can specify 0 rows or columns to indicate expansion in that direction as needed

BorderLayout

public BorderLayout()
- divides container into five regions: NORTH, SOUTH, WEST, EAST, CENTER
  - NORTH and SOUTH regions expand to fill region horizontally, and use preferred size vertically
  - WEST and EAST regions expand to fill region vertically, and use preferred size horizontally
  - CENTER uses all space not occupied by others

Container panel = new JPanel(new BorderLayout());
panel.add(new JButton("Button 1 (NORTH)", BorderLayout.NORTH);
BorderLayout example

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

public class BorderLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(210, 200));
        frame.setTitle("Run for the border");
        frame.setLayout(new BorderLayout());
        frame.add(new JButton("north"), BorderLayout.NORTH);
        frame.add(new JButton("south"), BorderLayout.SOUTH);
        frame.add(new JButton("west"), BorderLayout.WEST);
        frame.add(new JButton("east"), BorderLayout.EAST);
        frame.add(new JButton("center"), BorderLayout.CENTER);
        frame.setVisible(true);
    }
}
```

BoxLayout

- `Box.createHorizontalBox()`
- `Box.createVerticalBox()`

- aligns components in container in a single row or column
- components use preferred sizes and align based on their preferred alignment
- preferred way to construct a container with box layout:
  - `Box.createHorizontalBox()`
  - `Box.createVerticalBox()`

Complex layouts

- How would you create a complex window like this, using the layout managers shown?

Solution: composite layout

- create panels within panels
- each panel has a different layout, and by combining the layouts, more complex / powerful layout can be achieved
- example:
  - how many panels?
  - what layout in each?
Composite layout example

```java
import java.awt.*;
import javax.swing.*;
public class Telephone {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(250, 200));
        frame.setTitle("Telephone");
        frame.setLayout(new BorderLayout());
        JPanel centerPanel = new JPanel(new GridLayout(4, 3));
        for (int i = 1; i <= 9; i++) {
            centerPanel.add(new JButton("" + i));
        }
        centerPanel.add(new JButton("*"));
        centerPanel.add(new JButton("0"));
        centerPanel.add(new JButton("#"));
        frame.add(centerPanel, BorderLayout.CENTER);
        JPanel southPanel = new JPanel(new FlowLayout());
        southPanel.add(new JLabel("Number to dial: "));
        southPanel.add(new JTextField(10));
        frame.add(southPanel, BorderLayout.SOUTH);
        frame.setVisible(true);
    }
}
```

Additional components

A check box is a toggleable button with two states: checked and unchecked

A radio button is a button that can be selected; usually part of a group of mutually-exclusive radio buttons (1 selectable at a time)

- `public JCheckBox / JRadioButton(String text)`
  - Creates checked/unchecked check box with given text.
- `public boolean isSelected()`
  - Returns true if check box is checked.
- `public void setSelected(boolean selected)`
  - Sets box to be checked/unchecked.

ButtonGroup

A logical group of radio buttons that ensures that only one is selected at a time

- `public ButtonGroup()`
- `public void add(JRadioButton button)`

- The ButtonGroup is not a graphical component, just a logical group; the JButtons themselves are added to the container, not the ButtonGroup.
**ImageIcon**

Allows you to put a picture on a button, label or other component

- public class ImageIcon implements Icon
  - public ImageIcon(String filename)
  - public ImageIcon(URL address)

- in JButton, JRadioButton, JCheckBox, JLabel, etc...
  - constructor that takes an Icon
  - public void setIcon(Icon)
  - public void setSelectedIcon(Icon)
  - public void setRolloverIcon(Icon)

**JScrollPane**

A special container that holds a component, using scrollbars to allow that component to be seen

- public JScrollPane(Component comp)
  - Wraps the given component with scrollbars.

- After constructing the scroll pane, add the scroll pane to the container, not the original component.

```java
frame.add(new JScrollPane(area), BorderLayout.CENTER);
```

**JFileChooser**

A special dialog box that allows the user to select one or more files/folders

- public JFileChooser()
- public JFileChooser(String currentDir)
- public int showOpenDialog(Component parent)
- public int showSaveDialog(Component parent)
- public File getSelectedFile()
- public static int APPROVE_OPTION, CANCEL_OPTION
  - Possible result values from showXxxDialog(..).

```java
JFileChooser chooser = new JFileChooser();
int result = chooser.showSaveDialog(this);
if (result == JFileChooser.APPROVE_OPTION)
    this.saveData(chooser.getSelectedFile().getName());
```

**JColorChooser**

Another special dialog that lets the user pick from a palette of colors

- public JColorChooser()
- public JColorChooser(Color initial)
- public Color showDialog(Component parent, String title, Color initialColor)
  - returns null if user chose Cancel option
### JMenuBar

**The top-level container that holds menus; can be attached to a frame**

- public JMenuBar()
- public void add(JMenu menu)

**Usage:** In JFrame, the following method exists:
- public void setJMenuBar(JMenuBar bar)

### JMenu

**A menu to hold menu items; menus can contain other menus**

- public JMenu(String text)
- public void add(JMenuItem item)
- public void addSeparator()
- public void setMnemonic(int mnemonic)

### JMenuItem

**An entry in a frame’s Menu bar, which can be clicked to perform commands**

- public JMenuItem(String text)
- public JMenuItem(String text, Icon icon)
- public JMenuItem(String text, int mnemonic)
- public void addActionListener(ActionListener al)
- public void setAccelerator(KeyStroke ks)
- public void setEnabled(boolean b)
- public void setMnemonic(int mnemonic)

**Mouse and keyboard events**
Mouse events

Uses of mouse events:

- listen to clicks and movement of mouse within a GUI component (usually a panel)
- respond to mouse activity with appropriate actions
- create interactive programs that are driven by mouse activity

MouseListener example

```java
public class MyMouseListener implements MouseListener {
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
    public void mousePressed(MouseEvent event) {
        System.out.println("User pressed mouse button!");
    }
    public void mouseReleased(MouseEvent event) {}
}
```

MouseListener usage

```java
// assumes some custom panel class named MyPanel
MyPanel panel = new MyPanel();
panel.addMouseListener(new MyMouseListener());
```

Problem: Tedious to implement entire interface when only partial behavior is wanted/needed
MouseAdapter

- an abstract class with empty implementations of all MouseListener methods
- usage: extend MouseAdapter and override the methods you want to do something
- removes need for you to type in empty methods for all the ones you don’t want
- an example of the Adapter design pattern

MouseAdapter usage

```java
public class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent event) {
        System.out.println("User pressed mouse button!");
    }
}
```

// using the MyMouseAdapter
MyPanel panel = new MyPanel();
panel.addMouseListener(new MyMouseAdapter());

MouseEvent objects

- InputEvent
  - public static int BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK, CTRL_MASK, ALT_MASK, SHIFT_MASK
- MouseEvent
  - public int getClickCount()
  - public Point getPoint()
  - public int getX(), getY()
  - public Object getSource()
  - public int getModifiers() *(use the above button masks with this)*

SwingUtilities
- isLeftMouseButton(MouseEvent event)
- isRightMouseButton(MouseEvent event)

MouseEvent: usage

```java
public class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent event) {
        Point p = event.getPoint();
        Object source = event.getSource();
        if (source == this.panel && p.getX() < 10) {
            JOptionPane.showMessageDialog(null, "You clicked the left side of myPanel!");
        }
    }
}
```
**Mouse movement: MouseMotionListener**

```java
package java.awt.event;

public interface MouseMotionListener {
    public void mouseDragged(MouseEvent event);
    public void mouseMoved(MouseEvent event);
}
```

- abstract MouseMotionAdapter class provides empty implementations of both methods if you just want to override one

**MouseMotionAdapter example**

```java
public class MyMouseMotionAdapter extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent event) {
        Point p = event.getPoint();
        double x = event.getX();
        double y = event.getY();
        System.out.println("Mouse is at " + p);
        System.out.println("x is " + x);
        System.out.println("y is " + y);
    }
}
```

// using the listener
myPanel.addMouseMotionListener(new MyMouseMotionAdapter());

---

**MouseInputListener**

```java
package javax.swing.event;

public interface MouseInputListener extends MouseListener, MouseMotionListener {}
```

- more importantly: MouseInputAdapter class includes empty implementations for ALL methods from both mouse input interfaces, allowing same listener to listen to mouse clicks and movement

**MouseInputAdapter: Example**

```java
public class MyMouseInputAdapter extends MouseInputAdapter {
    public void mousePressed(MouseEvent event) {
        System.out.println("Mouse was pressed");
    }
    public void mouseMoved(MouseEvent event) {
        Point p = event.getPoint();
        System.out.println("Mouse is at " + p);
        System.out.println("x is " + p);
        System.out.println("y is " + y);
    }
}
```

// using the listener
MyMouseInputAdapter adapter = new MyMouseInputAdapter();
myPanel.addMouseListener(adapter);
myPanel.addMouseMotionListener(adapter);
Keyboard Events

- Usage of keyboard events:
  - listen to keyboard activity within a GUI component (usually a panel)
  - respond to keyboard activity with appropriate actions
  - control onscreen drawn characters and simulate text input

KeyListener

```java
package java.awt.event;

public interface KeyListener {
  public void keyPressed(KeyEvent event);
  public void keyReleased(KeyEvent event);
  public void keyTyped(KeyEvent event);
}
```

abstract class KeyAdapter implements all KeyListener methods

InputEvent

- public static int CTRL_MASK, ALT_MASK, SHIFT_MASK

KeyEvent

- public static int VK_A .. VK_Z, VK_0 .. VK_9, VK_F1 .. VK_F10, VK_UP, VK_LEFT, .., VK_TAB, VK_SPACE, VK_ENTER, ...
- (one for every key)
- public char getKeyChar()
- public int getKeyCode()
- public Object getSource()
- public int getModifiers()
- (use masks with this)

KeyAdapter example

class PacManKeyListener extends KeyAdapter {
  public void keyPressed(KeyEvent event) {
    char keyChar = event.getKeyChar();
    int keyCode = event.getKeyCode();
    if (keyCode == KeyEvent.VK_RIGHT) {
      pacman.setX(pacman.getX() + 1);
      pacpanel.repaint();
    } else if (keyChar == 'Q') {
      System.exit(0);
    }
  }
}

// assumes some custom panel class named PacPanel
PacPanel panel = new PacPanel();
panel.addKeyListener(new PacKeyListener());