Building Java Programs
Chapter 14

Graphical User Interfaces

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.
**JFrame example 1**

• A simple program that creates and shows a JFrame:

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

• Graphical output: ![Graphical output](image)

**JFrame example 2**

```java
import java.awt.*;
import javax.swing.*;
public class SimpleFrame2 {
    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: ![Graphical output](image)

**JFrame properties**

<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>
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.JComboBox
  │    │    └── javax.swing.JOptionPane
  │    │         └── javax.swing.JTextPane
  │    │             └── javax.swing.JTextComponent
  │    │                 └── javax.swing.JTextComponent
  │    │                     └── javax.swing.JTextComponent
  │    │                             └── javax.swing.JTextComponent
  │    │                                 └── javax.swing.JTextComponent
  │    └── 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

- Drawback: Both exist in Java now; easy to get them mixed up; still have to use both sometimes!

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.
      - How would we add more than one component to 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.
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 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.

Components example

- This program attempts to show 2 buttons:
  import java.awt.*;
  import javax.swing.*;
  public class ComponentsExample1 {
    public static void main(String[] args) {
      JFrame frame = new JFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(new Dimension(300, 100));
      frame.setTitle("A frame");
      JButton button1 = new JButton();
      button1.setText("I'm a button.");
      button1.setBackground(Color.BLUE);
      frame.add(button1);
      JButton button2 = new JButton();
      button2.setText("Click me!");
      button2.setBackground(Color.RED);
      frame.add(button2);
      frame.setVisible(true);
    }
  }

Layout problems

- The preceding program added two buttons to the frame, but only one appeared on the screen.

  layout manager: An object contained inside frames and other graphical containers that decides the position and size of the components inside the container.

  The default layout manager sizes each component added to occupy the full window space.

Changing layouts

- We can correct the program's appearance by changing the frame's layout manager.

  Change the layout by calling the setLayout method on the frame and passing a layout manager object.
  - We will see several layout managers later.
  - We'll use one called a FlowLayout, which sizes each component to its preferred size and positions them in left-to-right rows.

  If the following line is added to the preceding program just before calling setVisible(true), its appearance will be:
  frame.setLayout(new FlowLayout());
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 Event Hierarchy
```java
java.lang.Object
  +--java.util.EventObject
      +--java.awt.AWTEvent
          +--java.awt.event.ActionEvent
          +--java.awt.event.TextEvent
          +--java.awt.event.ComponentEvent
              +--java.awt.event.FocusEvent
              +--java.awt.event.WindowEvent
              +--java.awt.event.InputEvent
                  +--java.awt.event.KeyEvent
                  +--java.awt.event.MouseEvent
```

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`

Writing an `ActionListener`

// 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) {
        JOptionPane.showMessageDialog(null, "An event occurred!");
    }
}

Attaching an `ActionListener`

```
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

Laying out components
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?

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:

![Preferred Size Buttons](image)

Not preferred size:

![Not Preferred Size Buttons](image)

FlowLayout

```java
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"));
```

GridLayout

```java
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

```java
public GridLayout(int rows, int columns) {
    // implementation
}
```
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);
    }
}

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);
    }
}

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

public class BoxLayout {
    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);
        panel.add(new JButton("Button 2 (SOUTH)", BorderLayout.SOUTH);
        panel.add(new JButton("Button 3 (EAST)", BorderLayout.EAST);
        panel.add(new JButton("Button 4 (CENTRE)", BorderLayout.CENTER);
        panel.setVisible(true);
    }
}

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(); or Box.createVerticalBox();
Other layouts

- **CardLayout**
  layers of "cards" stacked on top of each other; one visible at a time

- **GridBagLayout**
  very complicated; recommendation: avoid

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: ", new Font("SansSerif", Font.PLAIN, 12)));
        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)**
  - public JCheckBox(String text, boolean isChecked)
  - 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 `RadioButton` 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:

  ```java
  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)

### Mnemonics

- **mnemonic**: menu hotkey assigned to a button or component – usually visible as an underlined key, activated by pressing:
  - Ctrl+key (buttons) or Alt+key (menus)
  - only work when input focus is on the component
  - (affects menus)

- **usage**: call setMnemonic(char) method – menu items also have constructor that takes mnemonic

myQuitButton.setMnemonic('Q');
myNewItem = new JMenuItem("New", 'N');
myNewItem.setMnemonic('N');  // or this
Accelerators

- **accelerator**: global hotkey combination that performs an action (ex: Alt-X to exit program) even on components that aren't in focus / visible
  - can be run at any time in the application
  - can optionally include modifiers like Shift, Alt
  - created by calling the getKeyStroke method of the KeyStroke class, and passing this to setAccelerator method of various components (buttons, menus)

```java
menuItem.setAccelerator(KeyStroke.getKeyStroke('T', KeyEvent.ALT_MASK));
```

JComboBox

- public JComboBox()
- public JComboBox(Object[] items)
- public JComboBox(Vector items)
- public JComboBox(ComboBoxModel model)

Constructs a combo box. Can optionally pass a vector or model of items. (See DefaultComboBoxModel for a model implementation.)

- public void addActionListener(ActionListener al)

Causes an action event to be sent to listener al when the user selects or types a new item in the combo box.

JComboBox: Managing Items

- public void addItem(Object item)
- public Object getItemAt(int index)
- public void removeAllItems()
- public void removeItem(Object item)
- public void removeItemAt(int index)

JComboBox: Selected Item

- public int getSelectedIndex()
- public Object getSelectedItem()
- public void setSelectedItem(Object item)
- public void setSelectedIndex(int index)
- public void setEnabled(boolean enabled)
- public void setEditable(boolean editable)

If editable, the user can type new arbitrary values into the combo box.
JTabbedPane

A container that can hold many "tab" subcontainers, each with components in it

• public JTabbedPane()
• public JTabbedPane(int tabAlignment)
  Constructs a new tabbed pane. Defaults to having the tabs on top; can be set to JTabbedPane.BOTTOM, LEFT, RIGHT, etc.

• public void addTab(String title, Component comp)
• public void addTab(String title, Icon icon, Component comp)
• public void addTab(String title, Icon icon, Component comp, String tooltip)
  Adds the given component as a tab in this tabbed pane. Can optionally use an icon and/or tool tip.

JTabbedPane methods

• public void insertTab(String title, Icon icon, Component comp, String tooltip, int index)
• public void remove(Component comp)
• public void remove(int index)
• public void removeAll()
• public void setSelectedComponent(Component c)
• public void setSelectedIndex(int index)

JSplitPane

A container that can hold two components, divided by a movable separator

• public JSplitPane()
• public JSplitPane(int orientation)
  Constructs a new tabbed pane. Defaults to having a horizontal split; can be set to JSplitPane.HORIZONTAL_SPLIT, VERTICAL_SPLIT, etc.

• public void setBottomComponent(Component comp)
• public void setLeftComponent(Component comp)
• public void setRightComponent(Component comp)
• public void setTopComponent(Component comp)
  Sets the given component to occupy the desired region of the split pane.

JToolBar

A movable container to hold common buttons, commands, etc

• public JToolBar()
• public JToolBar(int orientation)
• public JToolBar(String title)
• public JToolBar(String title, int orientation)
  Constructs a new tool bar, optionally with a title and orientation; can be JToolBar.HORIZONTAL or VERTICAL, defaults to horizontal

• public void add(Component comp)
  Adds the given component to this tool bar's flowing layout.
JToolBar: Usage

- construct toolbar
- add items (usually buttons) to toolbar
- add toolbar to edge of BorderLayout of content pane (usually NORTH)
  - don't put anything in other edges (N/S/E/W)!

JList: more methods

- public void clearSelection()
- public int getSelectedIndex()
- public int[] getSelectedIndices()
- public Object getSelectedValue()
- public Object[] getSelectedValues()
- public void setSelectedIndex(int index)
- public void setSelectedIndices(int[] indices)

Methods for getting / setting the selected item and index in the list.

JList: even more methods

- public ListModel getModel()
- public void setListData(Object[] data)
- public void setListData(Vector data)
- public void setModel(ListModel model)

Causes this list to now display the given collection of data.

- public int getSelectionMode()
- public void setSelectionMode(int mode)

Get / set selection mode for the list. For example, set to ListSelectionModel.SINGLE_SELECTION to only allow one item in the list to be chosen at once.

A list of selectable pre-defined items

- public JList() Constructs an empty JList.
- public JList(ListModel model)
- public JList(Object[] data)
- public JList(Vector data)

Constructs a JList that displays the given data.

- public void addListSelectionListener(
  ListSelectionListener lsl)

Adds the given listener to be informed when the selected index / indices change for this list.
A window that is a child of the overall JFrame; used for popup windows, option/config boxes, etc.

- public JDialog(Dialog parent, boolean modal)
- public JDialog(Frame parent, String title, boolean modal)
  Constructs a new dialog with the given parent and title. If the modal flag is set, this dialog is a child of the parent and the parent will be locked until the dialog is closed.

- public void show()
  Shows this dialog on screen. If the dialog is modal, calling show() will lock the parent frame/dialog.

- JDialog has most all JFrame methods: getContentPane(), setJMenuBar(JMenuBar), setResizable(boolean), setTitle(String), ...

Some eye candy...

- making a Java Swing GUI look like the native operating system:
  ```java
  try {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
  } catch (Exception e) {};
  ```

- adding borders to components:
  ```java
  whateverComponent.setBorder(BorderFactory.createLineBorder(Color.BLUE, 3));
  ```

Mouse and keyboard 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
Listening to clicks: MouseListener

package java.awt.event;

public interface MouseListener {
    public void mouseClicked(MouseEvent event);
    public void mouseEntered(MouseEvent event);
    public void mouseExited(MouseEvent event);
    public void mousePressed(MouseEvent event);
    public void mouseReleased(MouseEvent event);
}

MouseListener example

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

// 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
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());

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

public class MyMouseAdapter usage

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!");
        }
    }
}

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)

Mouse movement:

MouseMotionListener

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
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.addMouseListener(new MyMouseMotionAdapter());

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);
    }
}

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

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

MouseMotionAdapter example

MouseInputListener

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
### Key Presses: 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

### KeyEvent objects

- 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

```java
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());

### Other kinds of events

- WindowListener/Adapter/Event:
  Responds to events in a top-level window (JFrame). Useful to catch windowClosing event and respond, e.g. prompt to save current file

- FocusListener/Event:
  Responds to when the keyboard focus changes to a different component. Useful to validate input format on a text field.

- ListSelectionListener, HyperlinkListener, ItemListener:
  Event listeners for specific Swing components (JList, JEditorPane, JComboBox respectively)
2D Graphics

Some details

- JPanel's paintComponent does important things that we don't want to lose, so call the method super.paintComponent first thing!
- paintComponent's Graphics g argument represents "graphical context" object that can be told to draw things on the panel
- Guaranteed for every JPanel that actual object passed in is a Graphics2D (can cast)
- panel has small preferred size, so it will be tiny when put into some layouts; use setPreferredSize(Dimension) to make it bigger!
- refreshing panel: can't call paintComponent, so call repaint()

Basic panel painting syntax

```java
public class MyPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // important!
        // put your painting code here
    }
}
```

A drawing panel

- extend JPanel and override paintComponent method to use a panel as a drawing surface
- coordinate system: (0, 0) at top-left, x-axis increases rightward, y-axis downward
- panel surface is transparent unless drawn on
Quick drawing example

```java
public class MyPanel extends JPanel {
    public MyPanel() {
        this.setBackground(Color.WHITE);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g2.setColor(Color.BLUE);
        g.fillOval(10, 10, 20, 50);
    }
}
```

Drawing text strings

- public void `drawString(String s, int x, int y)`
  - Draws a string with its first letter’s bottom-left corner at the given location.
  - The string is drawn using the Graphics’ current color and font settings.
  - Before drawing the string, you can set the font, color, and other attributes.

Fonts

- public void `setFont(Font f)`
  - Sets this Graphics context to start writing text in the given font. (Forgotten at end of paintComponent call!)

- `java.awt.Font`
  - Text styles used when drawing Strings on the panel
    - public `Font(String name, int style, int size)`
    - some predefined font names:
      - "Serif", "SansSerif", "Monospaced"
    - font styles (can be combined with + operator):
      - `Font.BOLD`
      - `Font.ITALIC`
      - `Font.PLAIN`

Colors and paints

- `java.awt.Color`
  - (a simple single-colored paint)
    - public `Color(int r, int g, int b)`
    - public `Color(int r, int g, int b, int alpha)`
      - a partially-transparent color (range 0-255, 0=transparent)
    - public Color `brighter()`, `darker()`
    - public static Color BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW
Drawing shapes

Graphics
- public void drawLine(int x1, int y1, int x2, int y2)
- public void drawOval(int x, int y, int w, int h)
- public void fillOval(int x, int y, int w, int h)
- public void drawPolygon(int[] xpoints, int[] ypoints, int len)
- public void fillPolygon(int[] xpoints, int[] ypoints, int len)
- public void drawRect(int x, int y, int w, int h)
- public void fillRect(int x, int y, int w, int h)
- public void drawArc(...)
- public void fillArc(...)
...

Drawing images

Graphics
- public void drawImage(Image i, int x, int y, int width, int height, ImageObserver io)

Draws given image with this Graphics pen with its top-left corner at given location (pass the panel as the ImageObserver).

Classes for imaging

- java.awt.Toolkit
  Gets images from disk or internet
  - public static Toolkit getDefaultToolkit()
  - public Image getImage(String filename)
  - public Image getImage(URL address)
- java.awt.MediaTracker
  Ensures that images are loaded fully
  - public MediaTracker(Component comp)
  - public void addImage(Image img, int id)
  - public void waitForAll() throws InterruptedException
- java.awt.Image
  Represents a graphic image (BMP, GIF, ...)
  - public int getWidth(ImageObserver obs)
  - public int getHeight(ImageObserver obs)
- java.awt.image.BufferedImage
  A blank graphic image surface onto which you can draw
  - public BufferedImage(int w, int h, int type)
  - public Graphics getGraphics()

Images, continued

- Code to load an image:
  Toolkit tk = Toolkit.getDefaultToolkit();
  Image img = tk.getImage("myimage.jpg");
  MediaTracker mt = new MediaTracker(this);
  mt.addImage(img, 0); // any ID will do
  try {
    mt.waitForAll();
  } catch (InterruptedException ie) {}  

- This is tedious!
  - suggest making a helper method to load and return one image
    public Image loadImage(String filename)
Animation with Timers

Timers—Why?

- execute an action multiple times at given intervals
- create animations in our GUI programs
- add delays and pauses when required / desired

- a Timer is an example of a "callback" -- your code starts the timer, then later at a specified time, the timer activates, causing an event in your system

javax.swing.Timer

- public Timer(int ms_delay, ActionListener al)
  Causes al to fire an actionPerformed every ms_delay milliseconds.

- public void start()
  Causes timer to begin ticking; the first tick happens after the initial delay, and the rest occur separated by the timer’s delay.

- public void stop()
  Stops timer from ticking.

- public void restart()
  Restarts timer, causing it to wait its initial delay and then begin firing at its delay rate.

Timer: more methods

- public void addActionListener(ActionListener a)
  Adds additional listeners to be fired as the timer ticks.

- public boolean isRunning()
  Returns true when timer is ticking (has been started).

- public void setDelay(int delay)
  Changes the delay between timer ticks.

- public void setInitialDelay(int delay)
  Sets a one-time delay to occur before ticking starts.

- public void setRepeats(boolean b)
  Set to false to make timer fire only once.
// This code could be used to move a shape to the right
// across the screen.
int DELAY = 100;

ActionListener updater = new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        x++;
        myPanel.repaint();
    }
};

Timer tim = new Timer(DELAY, updater);
tim.start();