Graphical User Interfaces, 2D Graphics & Game Programming

Goals

- Review the different types of user-interface components and how to add them to containers
- Understand how to handle mouse events and other user input
- Learn how to display graphical shapes such as lines and ellipses, and the use of colors.
- Understand how to use Java2D for game (or general graphics) development.
- Understand the different aspects of a game program including, but not limited to game logic, input, graphics and sounds.

Goals

- To become familiar with common user-interface components, such as buttons, combo boxes, text areas, and menus
- To build programs that handle events from user-interface components
- To learn how to use the Eclipse WindowBuilder plug-in.

Frame Windows

- The `JFrame` class

```
JFrame frame = new JFrame();
frame.setSize(300, 400);
frame.setTitle("An Empty Frame");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
```

- `import javax.swing.;`

A Frame Window

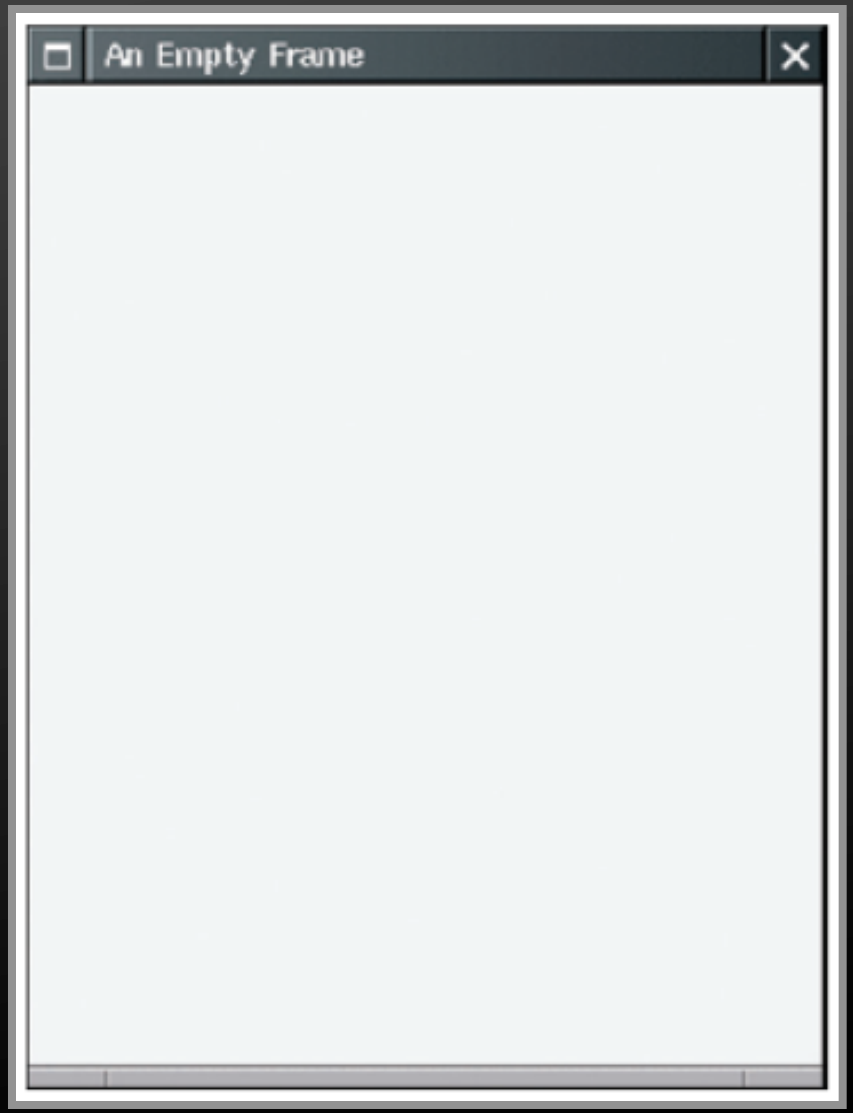


Figure 1:
A Frame Window

File EmptyFrameViewer.java

```
01: import javax.swing.*;
02:
03: public class EmptyFrameViewer
04: {
05:    public static void main(String[] args)
06:    {
07:       JFrame frame = new JFrame();
08:
09:       final int FRAME_WIDTH = 300;
10:       final int FRAME_HEIGHT = 400;
11:
12:       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
13:       frame.setTitle("An Empty Frame");
14:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:
16:       frame.setVisible(true);
17:    }
18: }
```

Basic GUI Construction

- Construct a frame
- Construct an object of your component class:

```
RectangleComponent component = new RectangleComponent();
```

- Add the component(s) to the frame

```
frame.add(component);
```

However, if you use an older version of Java (before Version 5), you must make a slightly more complicated call:

```
frame.getContentPane().add(component);
```

- Make the frame visible

Using Inheritance to Customize Frames

- Use inheritance for complex frames to make programs easier to understand
- Design a subclass of `JFrame`
- Store the components as instance fields
- Initialize them in the constructor of your subclass
- If initialization code gets complex, simply add some helper methods

Layout Management

- **Each container has a *layout manager* that directs the arrangement of its components**
- **Three useful layout managers:**
  - border layout
  - flow layout
  - grid layout

Layout Management

- By default, `JPanel` places components from left to right and starts a new row when needed
- Panel layout carried out by `FlowLayout` layout manager
- Can set other layout managers

```
panel.setLayout(new BorderLayout());
```

Border Layout

- Border layout groups container into five areas: center, north, west, south and east

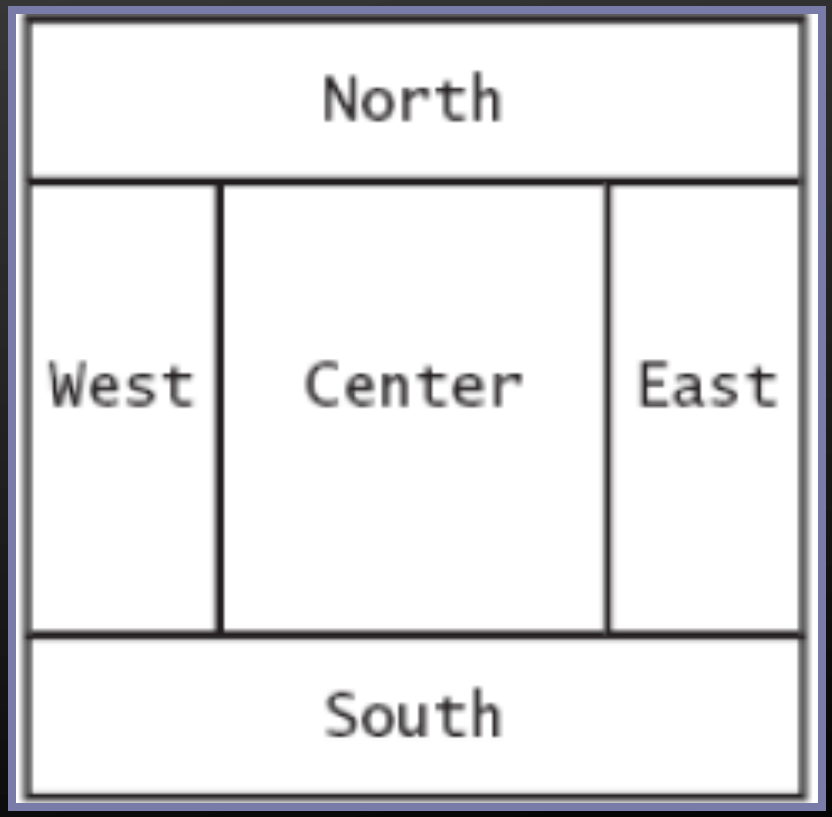


Figure 1:
Components Expand to Fill Space in the Border Layout

Continued…

Border Layout

- Default layout manager for a frame (technically, the frame's content pane)
- When adding a component, specify the position like this:

```
panel.add(component, BorderLayout.NORTH);
```

- Expands each component to fill the entire allotted area
  If that is not desirable, place each component inside a panel

Grid Layout

- Arranges components in a grid with a fixed number of rows and columns
- Resizes each component so that they all have same size
- Expands each component to fill the entire allotted area

Grid Layout

- Add the components, row by row, left to right:

```
JPanel numberPanel = new JPanel();
numberPanel.setLayout(new GridLayout(4, 3));
numberPanel.add(button7);
numberPanel.add(button8);
numberPanel.add(button9);
numberPanel.add(button4);
. . .
```

Grid Layout

Figure 2:
The Grid Layout

Grid Bag Layout

- **You can create acceptable-looking layouts by nesting panels**
  - Give each panel an appropriate layout manager
  - Panels without visible borders
  - Use as many panels as needed to organize components
- **Grid Bag provides a tabular arrangement of components**
  - Columns can have different sizes
  - Components can span multiple columns
- **More complicated to use**

Components - Choices

- Radio buttons
- Check boxes
- Combo boxes

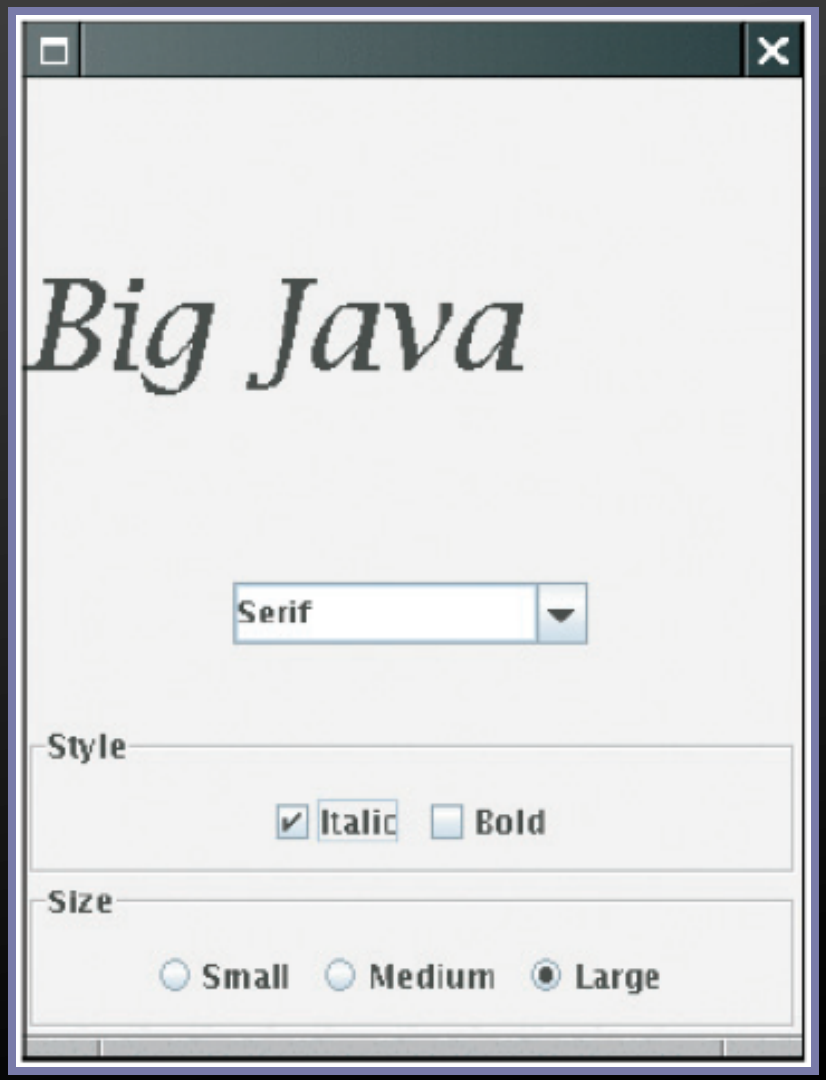


Figure 3:
A Combo Box, Check Box, and Radio Buttons

Radio Buttons

- For a small set of mutually exclusive choices, use radio buttons or a combo box
- In a radio button set, only one button can be selected at a time
- When a button is selected, previously selected button in set is automatically turned off

Radio Buttons

- In previous figure, font sizes are mutually exclusive:

```
JRadioButton smallButton = new JRadioButton("Small");
JRadioButton mediumButton = new JRadioButton("Medium");
JRadioButton largeButton = new JRadioButton("Large");


// Add radio buttons into a ButtonGroup so that
// only one button in group is on at any time
ButtonGroup group = new ButtonGroup();
group.add(smallButton);
group.add(mediumButton);
group.add(largeButton);
```

Check Boxes

- Two states: checked and unchecked
- Use a group of check boxes when one selection does not exclude another
- Construct by giving the name in the constructor:

```
JCheckBox italicCheckBox = new JCheckBox("Italic");
```

Check Boxes

- Construct by giving the name in the constructor:

```
JCheckBox italicCheckBox = new JCheckBox("Italic");
```

Combo Boxes

- **For a large set of choices, use a combo box (dropdown menu)**
  - Uses less space than radio buttons
- **"Combo": combination of a list and a text field**
  - The text field displays the name of the current selection

Combo Boxes

- Get user selection with `getSelectedItem` (return type is `Object`)

```
String selectedString =
    (String) facenameCombo.getSelectedItem();
```

- Select an item with `setSelectedItem`

Borders

- Can add a border to any component, but most commonly to panels:

```
Jpanel panel = new JPanel ();
panel.setBOrder(new EtchedBorder ());
```

- `Line Border`: simple line
- `EtchedBorder`: three-dimensional etched effect
- `TitledBorder`: a border with a title

Radio Buttons, Check Boxes, and Combo Boxes

- They generate an `ActionEvent` whenever the user selects an item

Continued…

Menus

- A frame contains a menu bar
- The menu bar contains menus
- A menu contains submenus and menu items

Menus

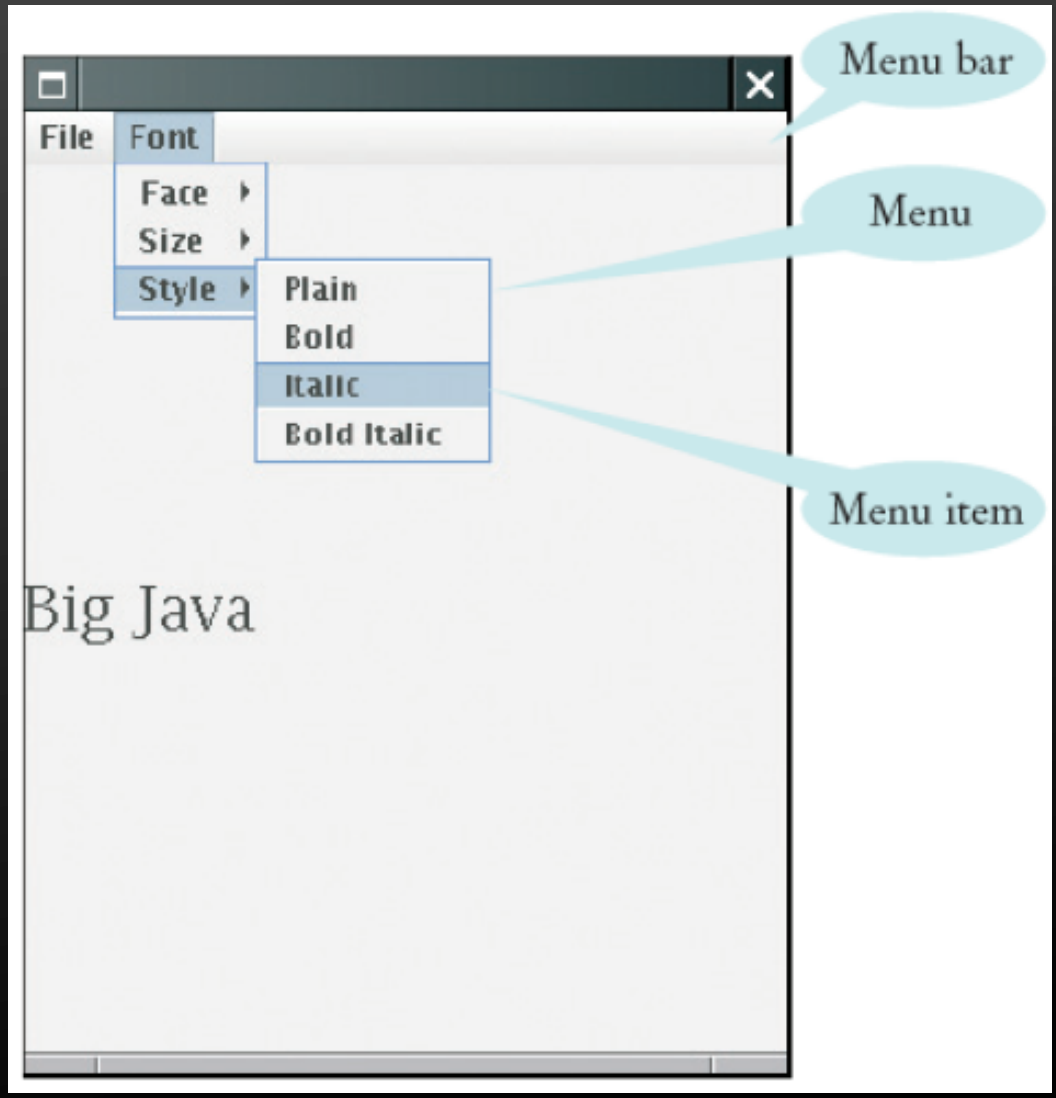


Figure 7:
Pull-Down Menus

Menu Items

- Add menu items and submenus with the `add` method:

```
JMenuItem fileExitItem = new JMenuItem("Exit");
fileMenu.add(fileExitItem);
```

- A menu item has no further submenus
- Menu items generate action events

Continued…

Menu Items

- Add a listener to each menu item:

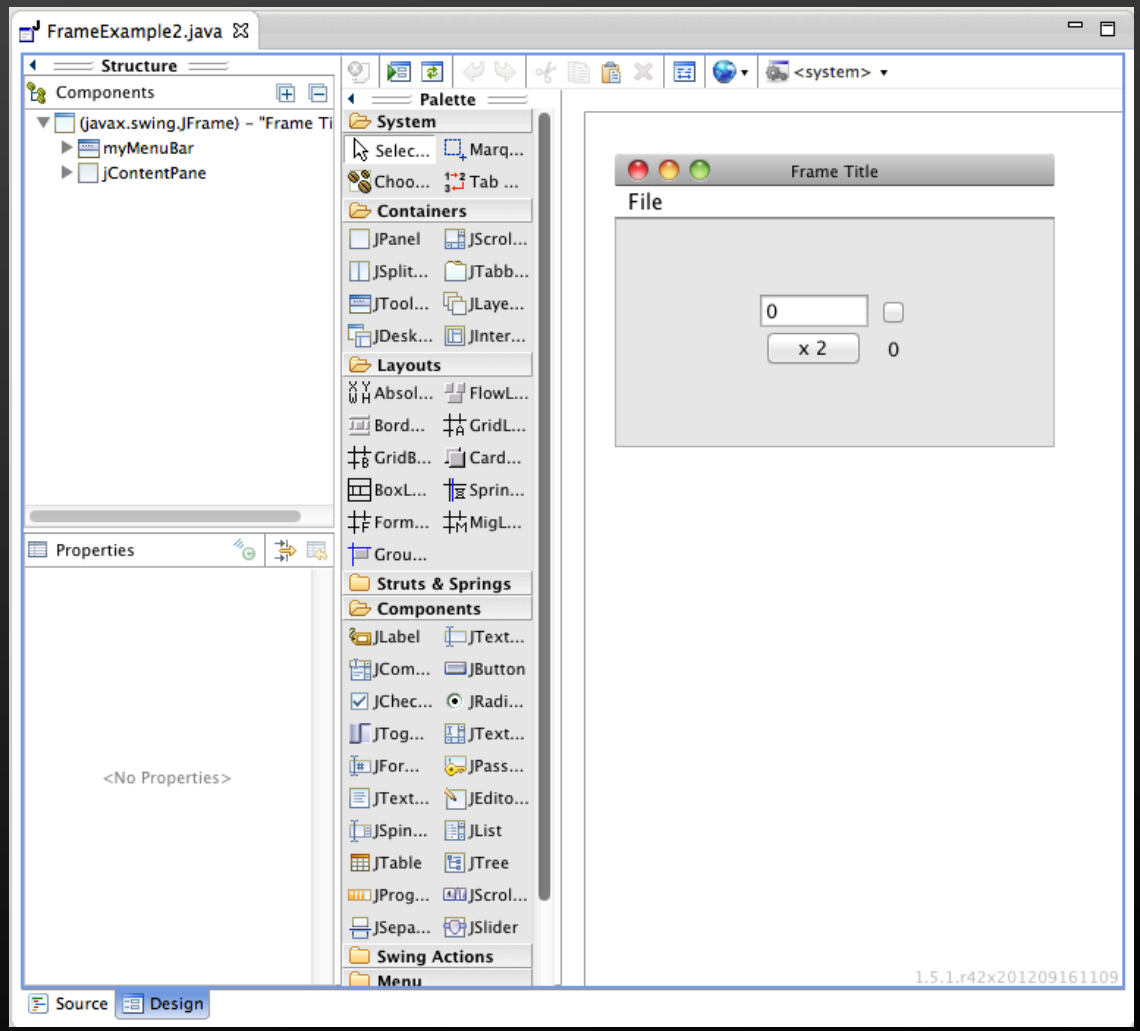
```
fileExitItem.addActionListener(listener);
```

- Add action listeners only to menu items, not to menus or the menu bar

Visual Programming

- **Allow you to have an overview of how the frame will look.**
- **Palette for selecting components.**
- **Properties and Layout management.**
- **Eclipse Plugin – Window Builder**
  - http://www.eclipse.org/windowbuilder/

Visual Programming



WindowBuilder Visual Programming Environment

Event Handling

- To understand the Java event model
- To install action and mouse event listeners
- To accept input from buttons, text fields, and the mouse

Events, Event Sources, and Event Listeners

- User interface *events* include key presses, mouse moves, button clicks, and so on
- Most programs don't want to be flooded by boring events
- A program can indicate that it only cares about certain specific events

Events and Event Listeners

- **Event listener:**
  - Notified when event happens
  - Belongs to a class that is provided by the application programmer
  - Its methods describe the actions to be taken when an event occurs

Events, Event Sources, and Event Listeners

- Example: Use `JButton` components for buttons; attach an `ActionListener` to each button
- `ActionListener` interface:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

- Need to supply a class whose `actionPerformed` method contains instructions to be executed when button is clicked

Events, Event Sources, and Event Listeners

- `event` parameter contains details about the event, such as the time at which it occurred
- Construct an object of the listener and add it to the button:

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

Processing Text Input

- Use `JTextField` components to provide space for user input

```
final int FIELD_WIDTH = 10; // In characters
final JTextField rateField = new JTextField(FIELD_WIDTH);
```

- Place a `JLabel` next to each text field

```
JLabel rateLabel = new JLabel("Interest Rate: ");
```

- Supply a button that the user can press to indicate that the input is ready for processing

Continued…

Processing Text Input

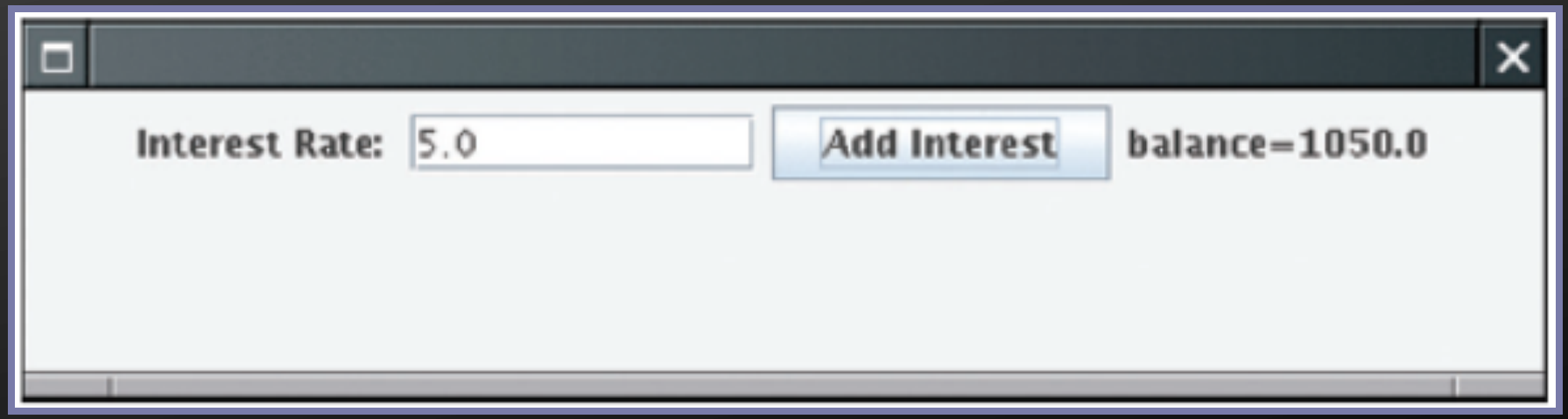


Figure 3:
An Application With a Text Field

Continued…

Processing Text Input

- The button's `actionPerformed` method reads the user input from the text fields (use `getText`)

```
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      double rate = Double.parseDouble(rateField.getText());
      . . .
   }
}
```

Mouse Events

- Use a mouse listener to capture mouse events
- Implement the `MouseListener` interface:

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   // Called when a mouse button has been pressed on a component
   void mouseReleased(MouseEvent event);
   // Called when a mouse button has been released on a component
   void mouseClicked(MouseEvent event);
   // Called when the mouse has been clicked on a component
   void mouseEntered(MouseEvent event);
   // Called when the mouse enters a component
   void mouseExited(MouseEvent event);
   // Called when the mouse exits a component
}
```

Mouse Events

- `mousePressed`, `mouseReleased`: called when a mouse button is pressed or released
- `mouseClicked`: if button is pressed and released in quick succession, and mouse hasn't moved
- `mouseEntered`, `mouseExited`: mouse has entered or exited the component's area

Mouse Events

- Add a mouse listener to a component by calling the `addMouseListener` method:

```
public class MyMouseListener implements MouseListener
{
   // Implements five methods
}
MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

Continued…

2D Graphics - Drawing Shapes

- `paintComponent`: called whenever the component needs to be repainted:

```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      // Recover Graphics2D
      Graphics2D g2 = (Graphics2D) g;
      . . .
   }
}
```

Drawing Shapes

- `Graphics` class lets you manipulate the graphics state (such as current color)
- `Graphics2D` class has methods to draw shape objects
- Use a cast to recover the `Graphics2D` object from the `Graphics` parameter

```
Rectangle box = new Rectangle(5, 10, 20, 30);
g2.draw(box);
```

- `java.awt` package

A Frame Window

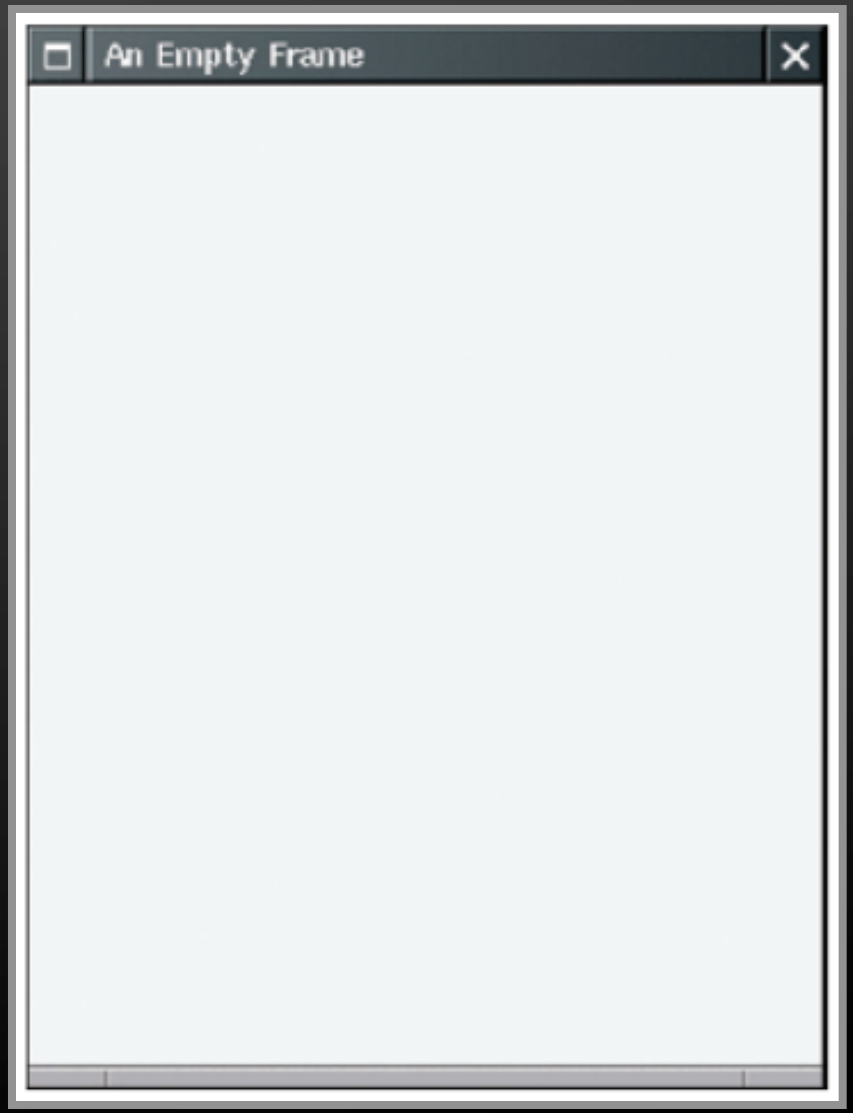


Figure 1:
A Frame Window

Drawing Rectangles

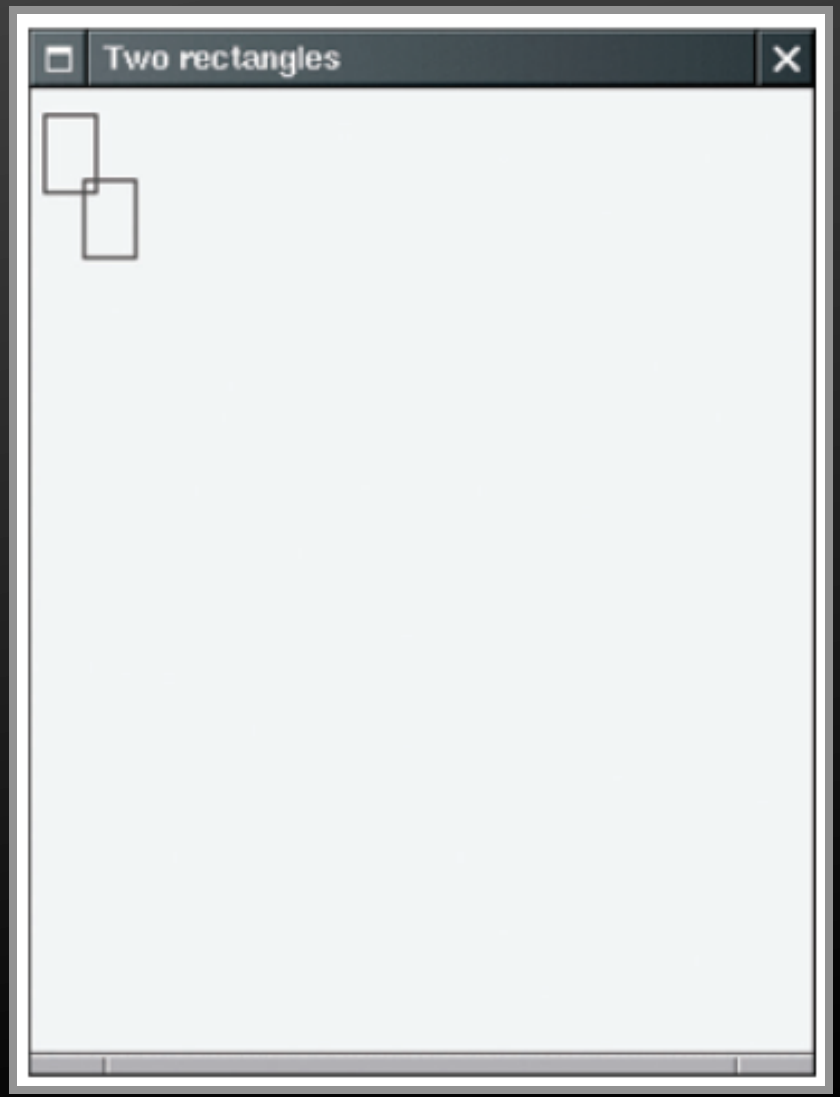


Figure 2:
Drawing Rectangles

Rectangle Drawing Program Classes

- `RectangleComponent`: its `paintComponent` method produces the drawing
- `RectangleViewer`: its `main` method constructs a frame and a `RectangleComponent`, adds the component to the frame, and makes the frame visible

Continued…

Rectangle Drawing Program Classes

- Construct a frame
- Construct an object of your component class:

```
RectangleComponent component = new RectangleComponent();
```

- Add the component to the frame

```
frame.add(component);
```

However, if you use an older version of Java (before Version 5), you must make a slightly more complicated call:

```
frame.getContentPane().add(component);
```

- Make the frame visible

File RectangleComponent.java

```
01: import java.awt.Graphics;
02: import java.awt.Graphics2D;
03: import java.awt.Rectangle;
04: import javax.swing.JPanel;
05: import javax.swing.JComponent;
06:
07: /**
08:    A component that draws two rectangles.
09: */
10: public class RectangleComponent extends JComponent
11: {
12:    public void paintComponent(Graphics g)
13:    {
14:       // Recover Graphics2D
15:       Graphics2D g2 = (Graphics2D) g;
16:
```

Continued…

File RectangleComponent.java

```
17:         // Construct a rectangle and draw it
18:         Rectangle box = new Rectangle(5, 10, 20, 30);
19:         g2.draw(box);
20:
21:         // Move rectangle 15 units to the right and 25 units
    // down
22:         box.translate(15, 25);
23:
24:         // Draw moved rectangle
25:         g2.draw(box);
26:      }
27: }
```

Graphical Shapes

- `Rectangle`, `Ellipse2D.Double`, and `Line2D.Double` describe graphical shapes
- We won't use the `.Float` classes
- These classes are inner classes–doesn't matter to us except for the `import` statement:

```
import java.awt.geom.Ellipse2D; // no .Double
```

- Must construct *and draw* the shape

```
Ellipse2D.Double ellipse = new Ellipse2D.Double(x, y, width, height);
g2.draw(ellipse);
```

An Ellipse

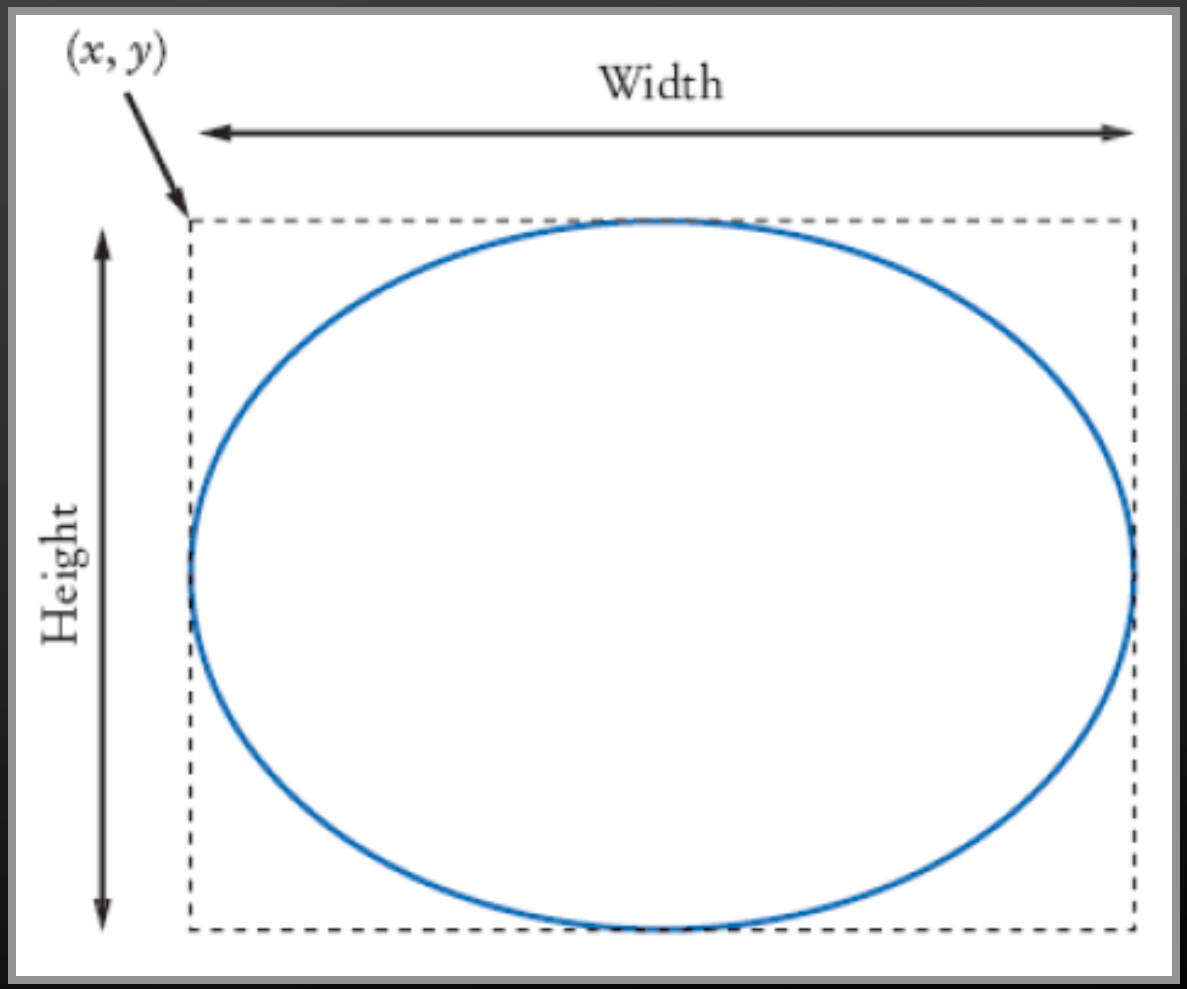


Figure 6:
An Ellipse and Its Bounding Box

Drawing Lines

- **To draw a line:**

```
Line2D.Double segment = new Line2D.Double(x1, y1, x2, y2);
```

or,

```
Point2D.Double from = new Point2D.Double(x1, y1);
Point2D.Double to = new Point2D.Double(x2, y2);
Line2D.Double segment = new Line2D.Double(from, to);
```

Drawing Strings

```
g2.drawString("Message", 50, 100);
```

Figure 7:
Basepoint and Baseline

Colors

- **Standard colors** `Color.BLUE, Color.RED, Color.PINK` **etc.**
- **Specify red, green, blue between** `0.0F` **and** `1.0F`

```
Color magenta = new Color(1.0F, 0.0F, 1.0F); // F = float
```

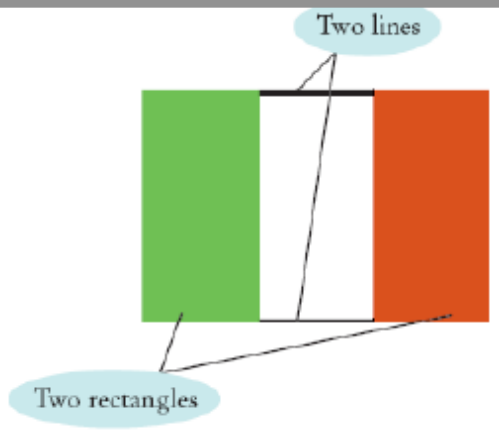
- **Set color in graphics context**

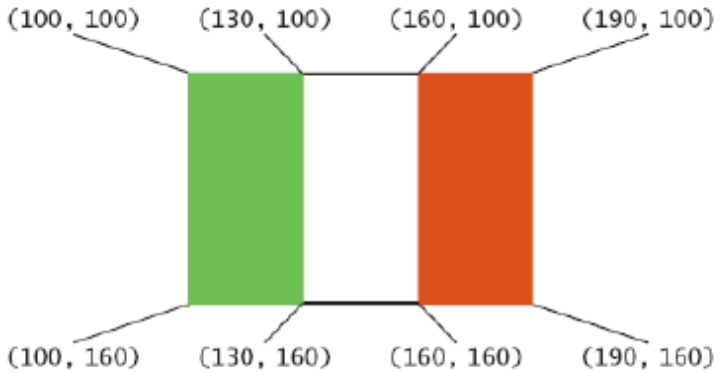
```
g2.setColor(magenta);
```

- **Color is used when drawing and filling shapes**

```
g2.fill(rectangle); // filled with current color
```

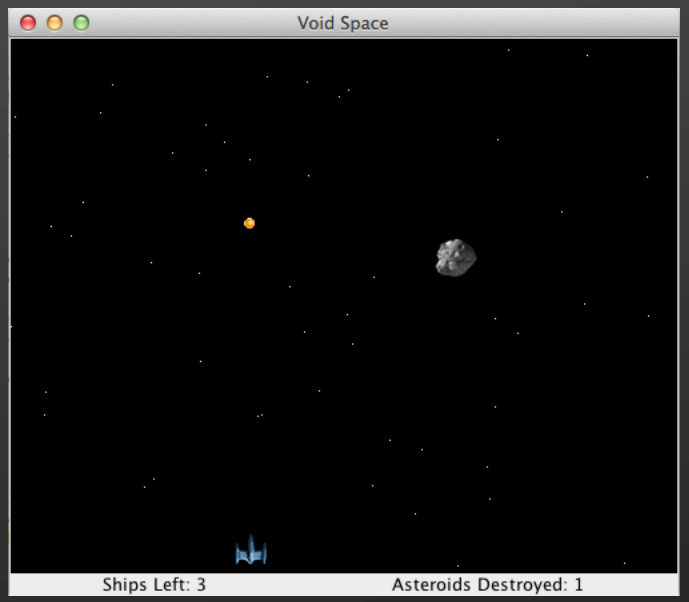
Drawing Graphical Shapes






```
Rectangle leftRectangle = new Rectangle(100, 100, 30, 60);
Rectangle rightRectangle = new Rectangle(160, 100, 30, 60);
Line2D.Double topLine
   = new Line2D.Double(130, 100, 160, 100);
Line2D.Double bottomLine
   = new Line2D.Double(130, 160, 160, 160);
```

Void Space Game



- **Main Game Loop**
  - **GameLogic:** Check Conditions
  - **GameScreen.updateScreen():** Update the game graphics and handle events
    - **GraphicsManager**: Draw Graphic Shapes
    - Detect collisions and draw explosions
    - **SoundManager**: Play sound effects
  - **InputHandler**: Handle game controls/input
  - **GameScreen.repaint()**: Repaint the screen using updated image