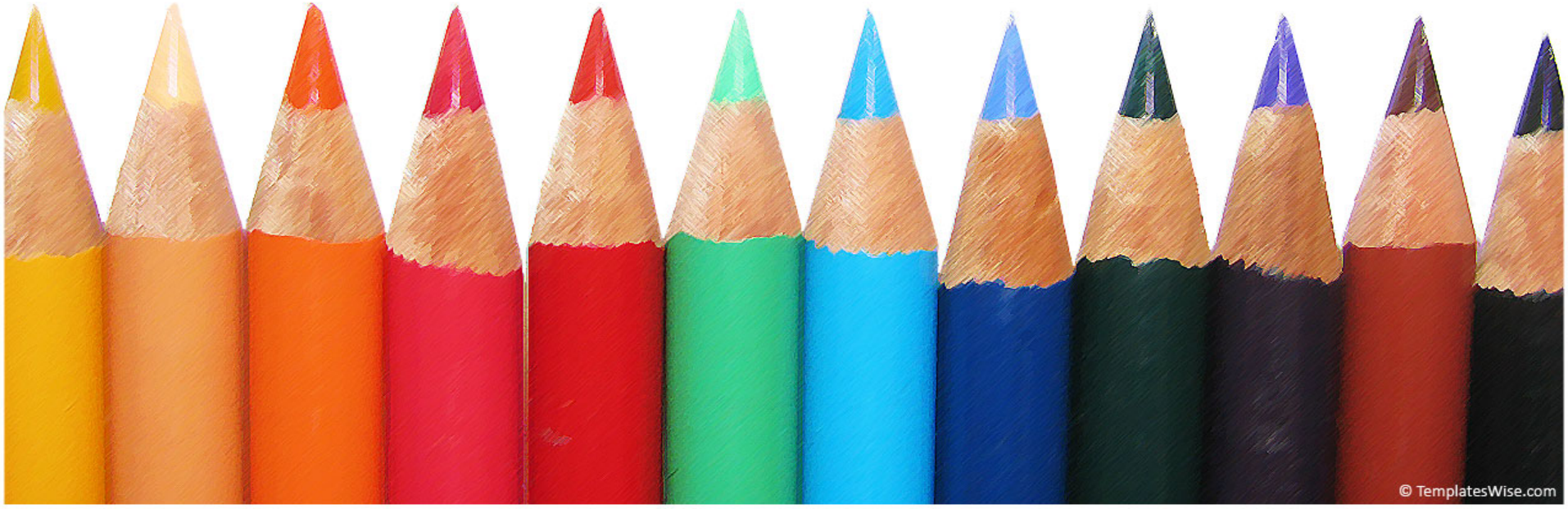
GUI BASICS

Chapter 14



Topics

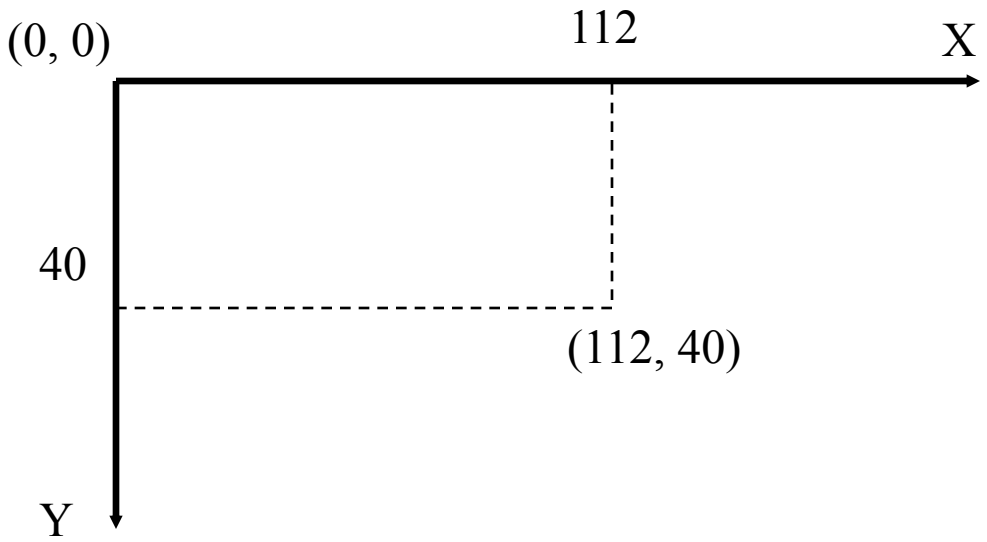
- Java Graphics- Pictures and Pixels
- Java Graphic Libraries, AWT, Swing and JavaFX
- Creating Interfaces within the GUI hierarchy
- Containers and Layout Managers
- Adding Functional Components

Graphics Basics

- A picture or drawing must be digitized for storage on a computer
- A picture is made up of *pixels* (picture elements), and each pixel is stored separately
- The number of pixels used to represent a picture is called the *picture resolution*
- The number of pixels that can be displayed by a monitor is called the *monitor resolution*
- *The more pixels the more the processor has to work to display them*

Coordinate Systems

- Each pixel can be identified using a two-dimensional coordinate system
- When referring to a pixel in a Java program, we use a coordinate system with the origin in the top-left corner



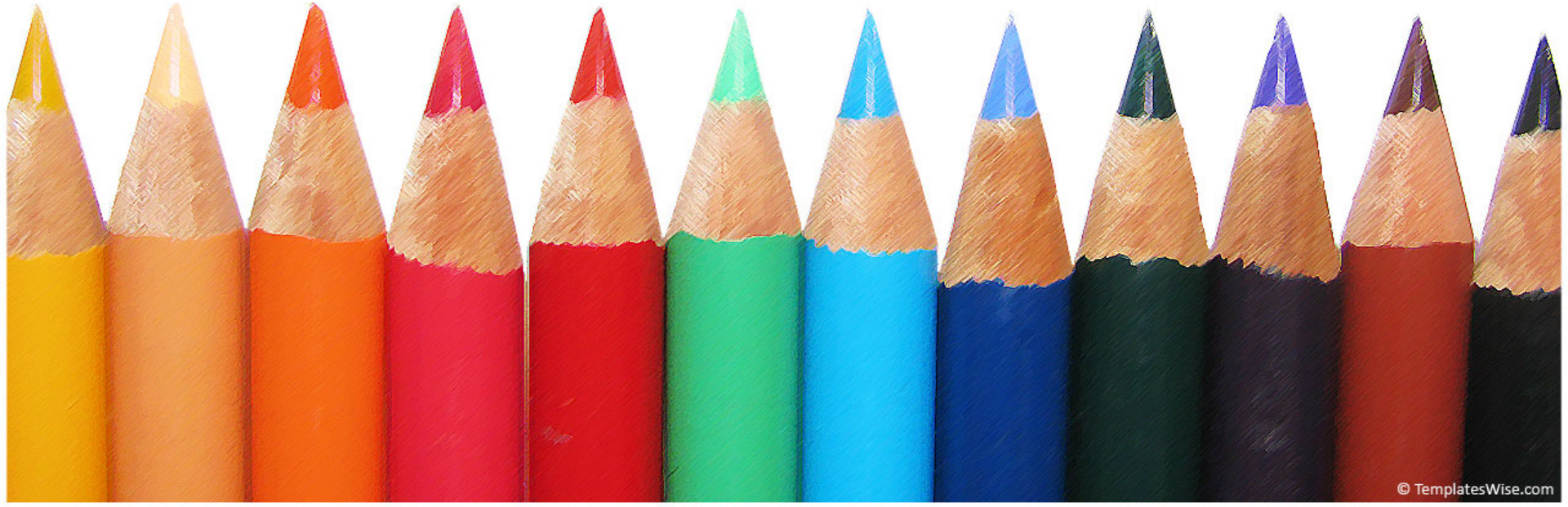
Representing Color

A black and white picture could be stored using one bit per pixel (0 = white and 1 = black)

A colored picture requires more information; there are several techniques for representing colors

For example, every color can be represented as a mixture of the three additive primary colors Red, Green, and Blue

Each color is represented by three numbers between 0 and 255 that collectively are called an *RGB value*



The old `Color` Class

You can set colors for GUI components by using the java.awt.Color class. 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). This is known as the *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);
```

Standard Colors

Thirteen standard colors (black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow) are defined as constants in java.awt.Color.

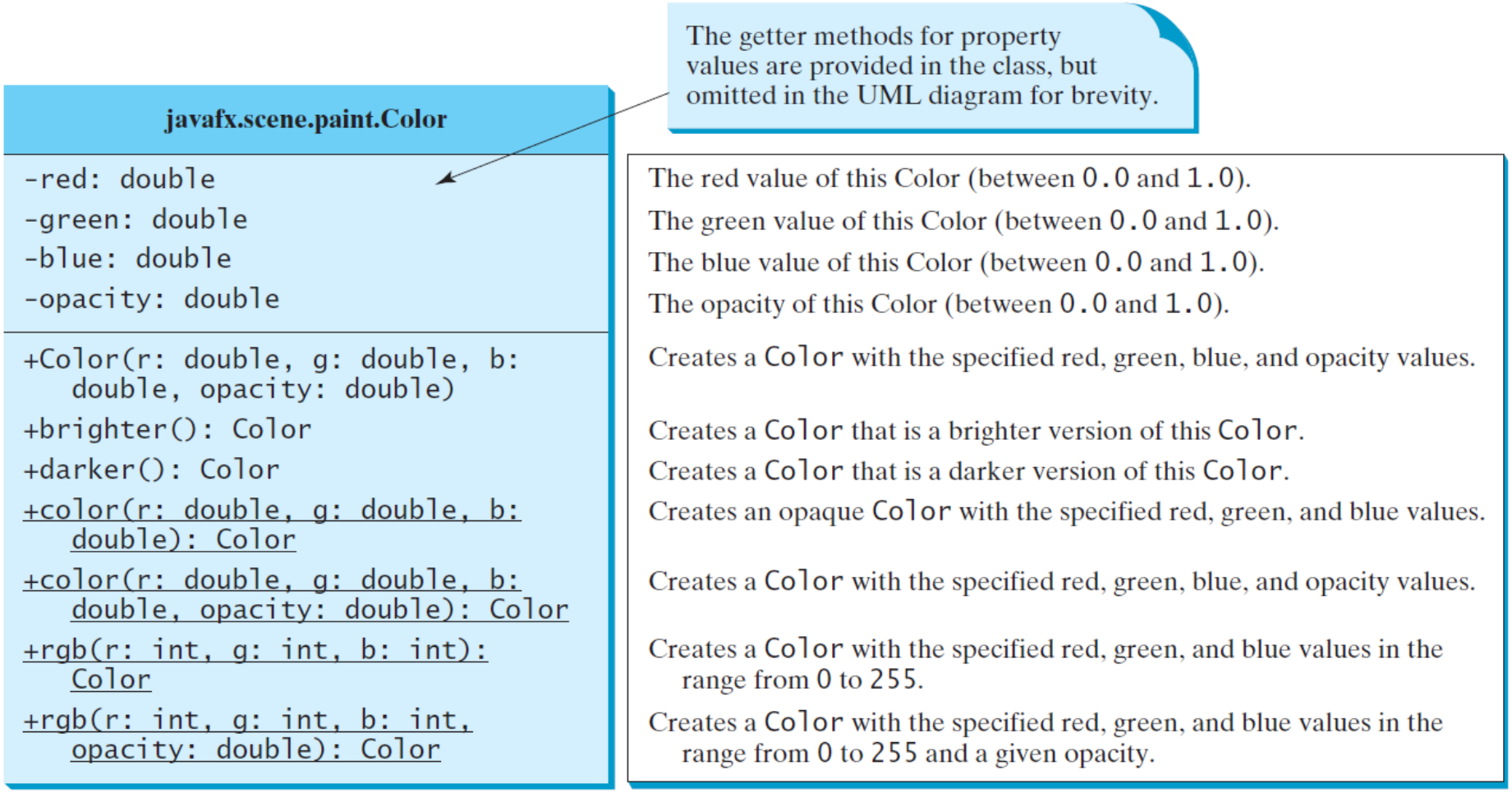
The standard color names are constants, but they are named as variables with lowercase for the first word and uppercase for the first letters of subsequent words. Thus the color names violate the Java naming convention. Since JDK 1.4, you can also use the new constants: BLACK, BLUE, NAVY, BROWN, BEIGE, CYAN, DARKGRAY, GRAY, LIGHTGRAY, SILVER, GREEN, GOLD, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW.

The OLD Color Class

- A color in a Java program is represented as an object created from the `Color` class
- The `Color` class also contains several predefined colors, including the following:

Object	RGB Value
Color.black	0, 0, 0
Color.blue	0, 0, 255
Color.cyan	0, 255, 255
Color.orange	255, 200, 0
Color.white	255, 255, 255
Color.yellow	255, 255, 0

The NEW Color Class



Setting Colors

- One way to set colors is to use the predfined color choices
  - thing1.setFill (Color.BLACK);
- Or create your own Color
  - Color c= Color.color(0.3, 0.7, 0.1); //assumes alpha of 1.0
  - Color c=Color.rgb(255,255,0, 0.5) //explicit alpha
- How many Colors are possible with RGB?

Motivations

JavaFX is a new framework for developing Java GUI programs. The JavaFX API is an excellent example of how the object-oriented principle is applied. This chapter serves two purposes. First, it presents the basics of JavaFX programming. Second, it uses JavaFX to demonstrate OOP. Specifically, this chapter introduces the framework of JavaFX and discusses JavaFX GUI components and their relationships.

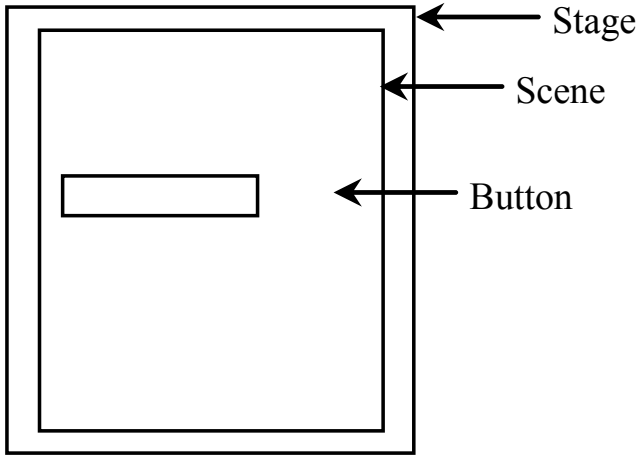
JavaFX vs Swing and AWT

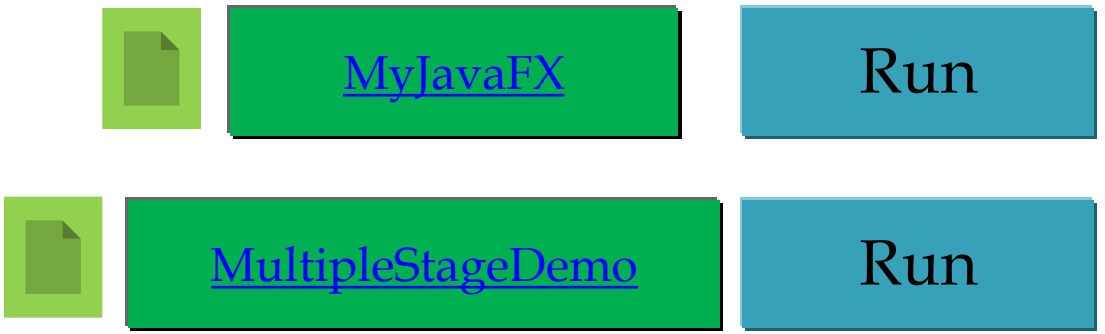
Swing and AWT are replaced by the JavaFX platform for developing rich Internet applications.

When Java was introduced, the GUI classes were bundled in a library known as the *Abstract Windows Toolkit (AWT)*. AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects. In addition, AWT is prone to platform-specific bugs. The AWT user-interface components were replaced by a more robust, versatile, and flexible library known as *Swing components*. Swing components are painted directly on canvases using Java code. Swing components depend less on the target platform and use less of the native GUI resource. With the release of Java 8, Swing is replaced by a completely new GUI platform known as *JavaFX*.

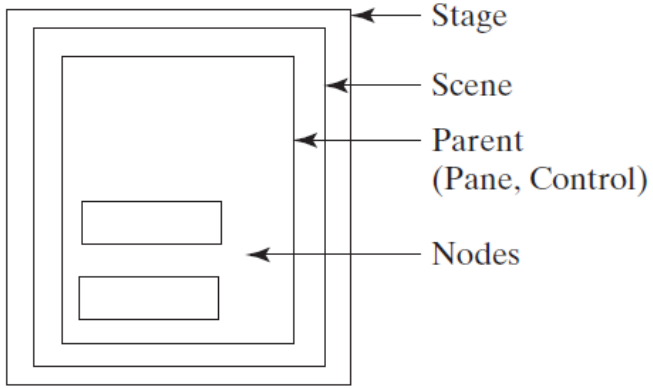
Basic Structure of JavaFX

- Application
- Override the start(Stage) method
- Stage, Scene, and Nodes

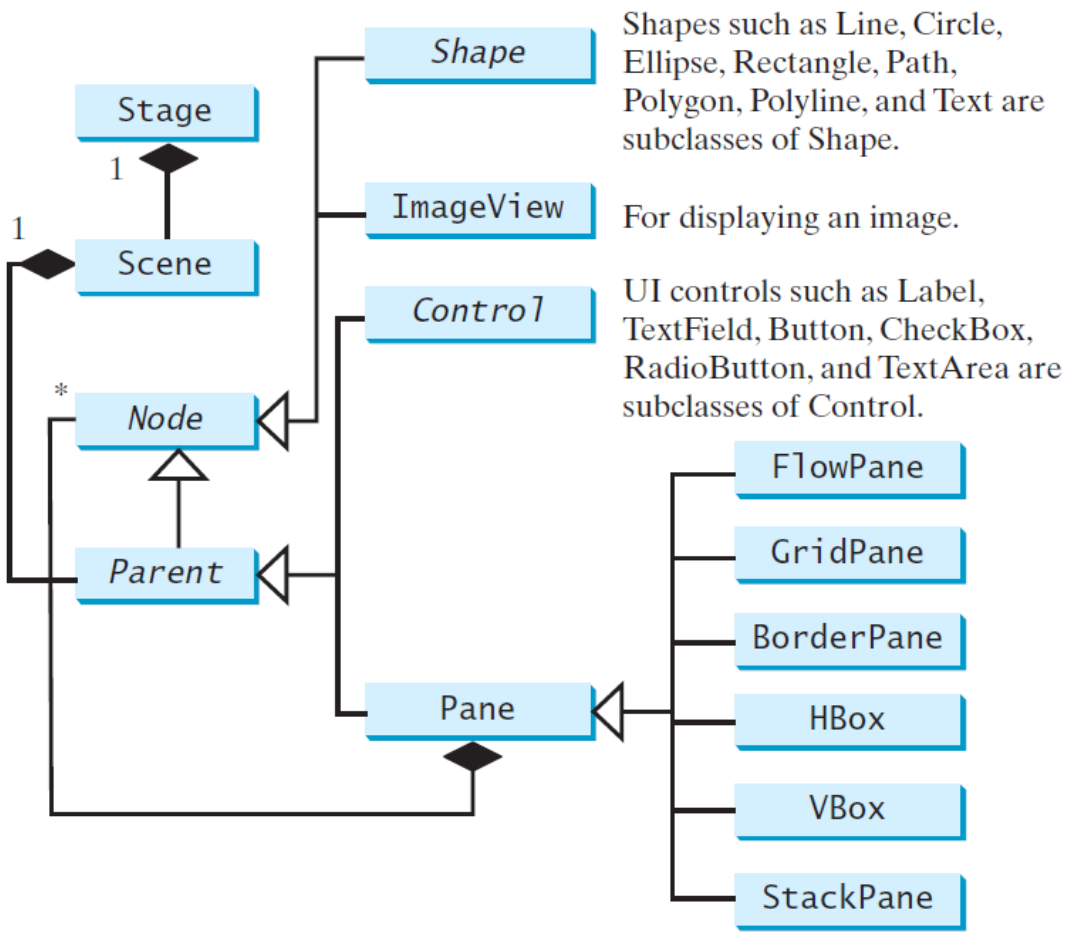




Panes, UI Controls, and Shapes



(a)



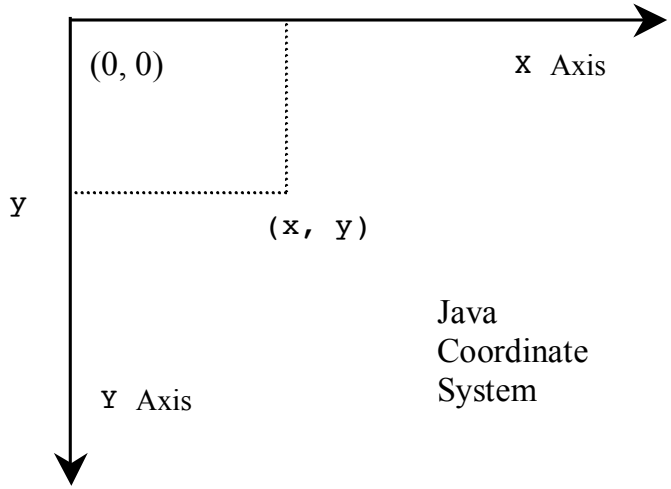
(b)

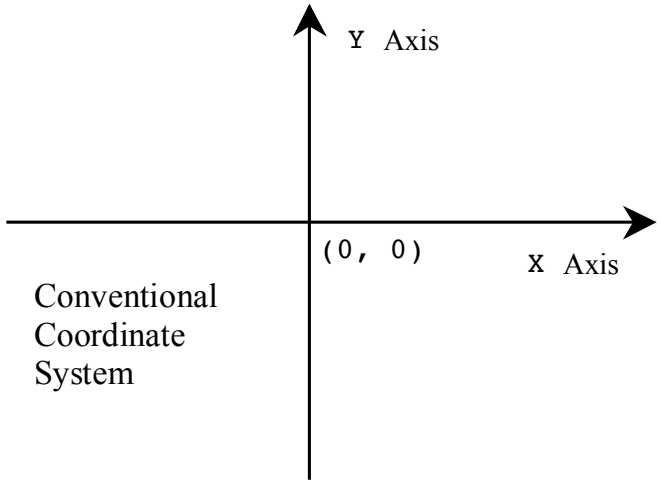
ButtonInPane

Run

Display a Shape

This example displays a circle in the center of the pane.





ShowCircle

Run

Binding Properties

JavaFX introduces a new concept called *binding property* that enables a *target object* to be bound to a *source object*. If the value in the source object changes, the target property is also changed automatically. The target object is simply called a *binding object* or a *binding property*.

ShowCircleCentered

Run

Binding Property: getter, setter, and property getter

```
public class SomeClassName {

  private PropertyType x;

  /** Value getter method */
  public propertyValueType getX() { ... }

  /** Value setter method */
  public void setX(propertyValueType value) { ... }

  /** Property getter method */
  public PropertyType
    xProperty() { ... }
}
```

(a) x is a binding property

```
public class Circle {

  private DoubleProperty centerX;

  /** Value getter method */
  public double getCenterX() { ... }

  /** Value setter method */
  public void setCenterX(double value) { ... }

  /** Property getter method */
  public DoubleProperty centerXProperty() { ... }
}
```

(b) centerX is binding property

Common Properties and Methods for Nodes

- style: set a JavaFX CSS style
- rotate: Rotate a node

NodeStyleRotateDemo

Run

The Font Class

The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.

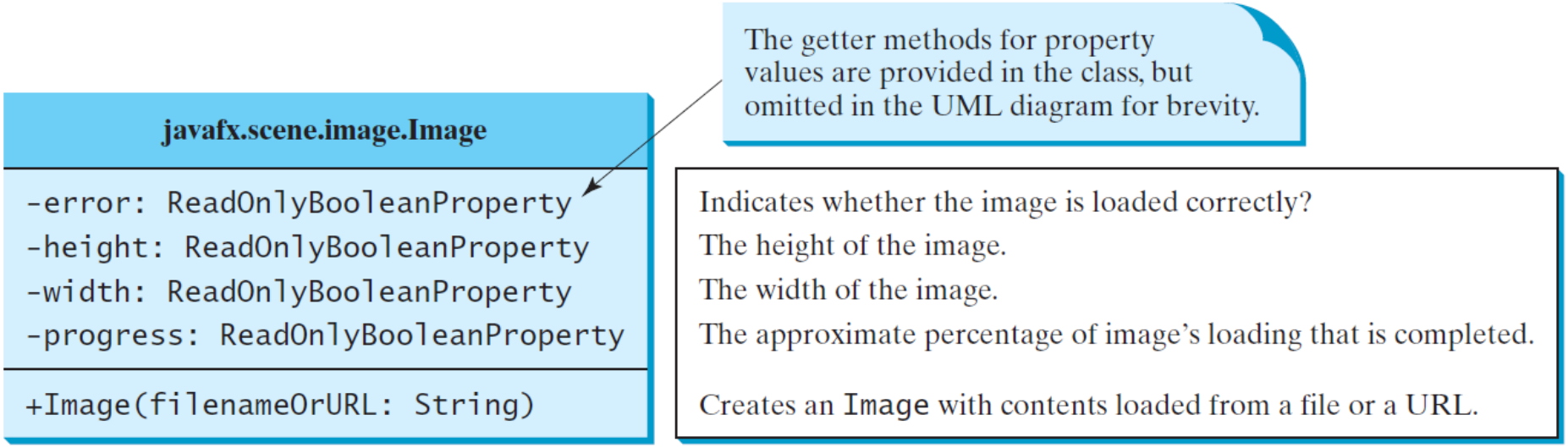
javafx.scene.text.Font

-size: double	The size of this font.
-name: String	The name of this font.
-family: String	The family of this font.
+Font(size: double)	Creates a Font with the specified size.
+Font(name: String, size: double)	Creates a Font with the specified full font name and size.
+font(name: String, size: double)	Creates a Font with the specified name and size.
+font(name: String, w: FontWeight, size: double)	Creates a Font with the specified name, weight, and size.
+font(name: String, w: FontWeight, p: FontPosture, size: double)	Creates a Font with the specified name, weight, posture, and size.
+getFamilies(): List<String>	Returns a list of font family names.
+getFontNames(): List<String>	Returns a list of full font names including family and weight.

FontDemo

Run

The Image Class



The ImageView Class

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.image.ImageView

```
-fitHeight: DoubleProperty
-fitWidth: DoubleProperty
-x: DoubleProperty
-y: DoubleProperty
-image: ObjectProperty<Image>
```

```
+ImageView()
+ImageView(image: Image)
+ImageView(filenameOrURL: String)
```

The height of the bounding box within which the image is resized to fit.
The width of the bounding box within which the image is resized to fit.
The x-coordinate of the ImageView origin.
The y-coordinate of the ImageView origin.
The image to be displayed in the image view.

Creates an `ImageView`.
Creates an `ImageView` with the specified image.
Creates an `ImageView` with image loaded from the specified file or URL.

ShowImage

Run

Layout Panes

JavaFX provides many types of panes for organizing nodes in a container.

Class	*Description*
`Pane`	Base class for layout panes. It contains the `getChildren()` method for returning a list of nodes in the pane.
`StackPane`	Places the nodes on top of each other in the center of the pane.
`FlowPane`	Places the nodes row-by-row horizontally or column-by-column vertically.
`GridPane`	Places the nodes in the cells in a two-dimensional grid.
`BorderPane`	Places the nodes in the top, right, bottom, left, and center regions.
`HBox`	Places the nodes in a single row.
`VBox`	Places the nodes in a single column.

FlowPane

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.FlowPane

```
-alignment: ObjectProperty<Pos>
-orientation:
   ObjectProperty<Orientation>
-hgap: DoubleProperty
-vgap: DoubleProperty
```

```
+FlowPane()
+FlowPane(hgap: double, vgap:
   double)
+FlowPane(orientation:
   ObjectProperty<Orientation>)
+FlowPane(orientation:
   ObjectProperty<Orientation>,
   hgap: double, vgap: double
```

The overall alignment of the content in this pane (default: `Pos.LEFT`).
The orientation in this pane (default: `Orientation.HORIZONTAL`).

The horizontal gap between the nodes (default: `0`).
The vertical gap between the nodes (default: `0`).

Creates a default `FlowPane`.
Creates a `FlowPane` with a specified horizontal and vertical gap.

Creates a `FlowPane` with a specified orientation.

Creates a `FlowPane` with a specified orientation, horizontal gap and vertical gap.

ShowFlowPane

Run

GridPane

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.GridPane

```
-alignment: ObjectProperty<Pos>
-gridLinesVisible:
   BooleanProperty
-hgap: DoubleProperty
-vgap: DoubleProperty
```

```
+GridPane()
+add(child: Node, columnIndex:
   int, rowIndex: int): void
+addColumn(columnIndex: int,
   children: Node...): void
+addRow(rowIndex: int,
   children: Node...): void
+getColumnIndex(child: Node):
   int
+setColumnIndex(child: Node,
   columnIndex: int): void
+getRowIndex(child:Node): int
+setRowIndex(child: Node,
   rowIndex: int): void
+setHalighnment(child: Node,
   value: HPos): void
+setValighnment(child: Node,
   value: VPos): void
```

The overall alignment of the content in this pane (default: `Pos.LEFT`).
Is the grid line visible? (default: `false`)

The horizontal gap between the nodes (default: 0).
The vertical gap between the nodes (default: 0).

Creates a `GridPane`.
Adds a node to the specified column and row.

Adds multiple nodes to the specified column.

Adds multiple nodes to the specified row.

Returns the column index for the specified node.

Sets a node to a new column. This method repositions the node.

Returns the row index for the specified node.
Sets a node to a new row. This method repositions the node.

Sets the horizontal alignment for the child in the cell.

Sets the vertical alignment for the child in the cell.

ShowGridPane

Run

BorderPane

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.BorderPane

```
-top: ObjectProperty<Node>
-right: ObjectProperty<Node>
-bottom: ObjectProperty<Node>
-left: ObjectProperty<Node>
-center: ObjectProperty<Node>
```

```
+BorderPane()
+setAlignment(child: Node, pos: Pos)
```

The node placed in the top region (default: `null`).
The node placed in the right region (default: `null`).
The node placed in the bottom region (default: `null`).
The node placed in the left region (default: `null`).
The node placed in the center region (default: `null`).

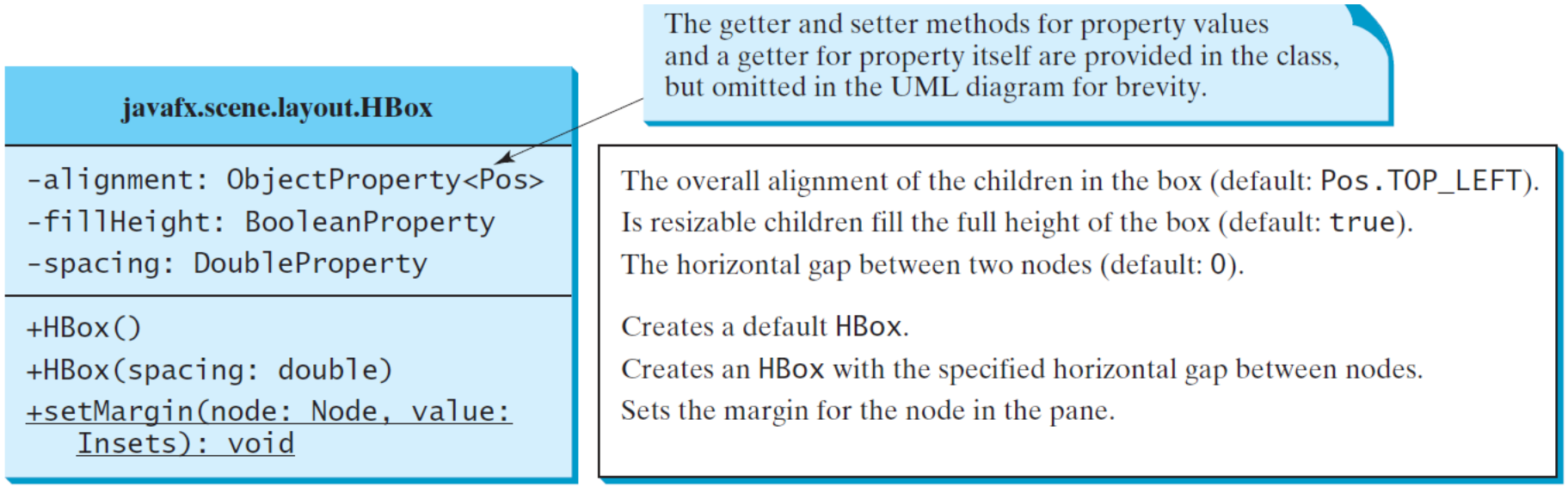
Creates a `BorderPane`.

Sets the alignment of the node in the `BorderPane`.

ShowBorderPane

Run

HBox


javafx.scene.layout.HBox
-alignment: ObjectProperty<Pos>
-fillHeight: BooleanProperty
-spacing: DoubleProperty
+HBox()
+HBox(spacing: double)
+setMargin(node: Node, value: Insets): void
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
The overall alignment of the children in the box (default: Pos.TOP_LEFT).
Is resizable children fill the full height of the box (default: true).
The horizontal gap between two nodes (default: 0).
Creates a default HBox.
Creates an HBox with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

VBox

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.VBox

```
-alignment: ObjectProperty<Pos>
-fillWidth: BooleanProperty
-spacing: DoubleProperty
```

```
+VBox()
+VBox(spacing: double)
+setMargin(node: Node, value:
   Insets): void
```

The overall alignment of the children in the box (default: `Pos.TOP_LEFT`).

Is resizable children fill the full width of the box (default: `true`).

The vertical gap between two nodes (default: `0`).

Creates a default `VBox`.

Creates a `VBox` with the specified horizontal gap between nodes.

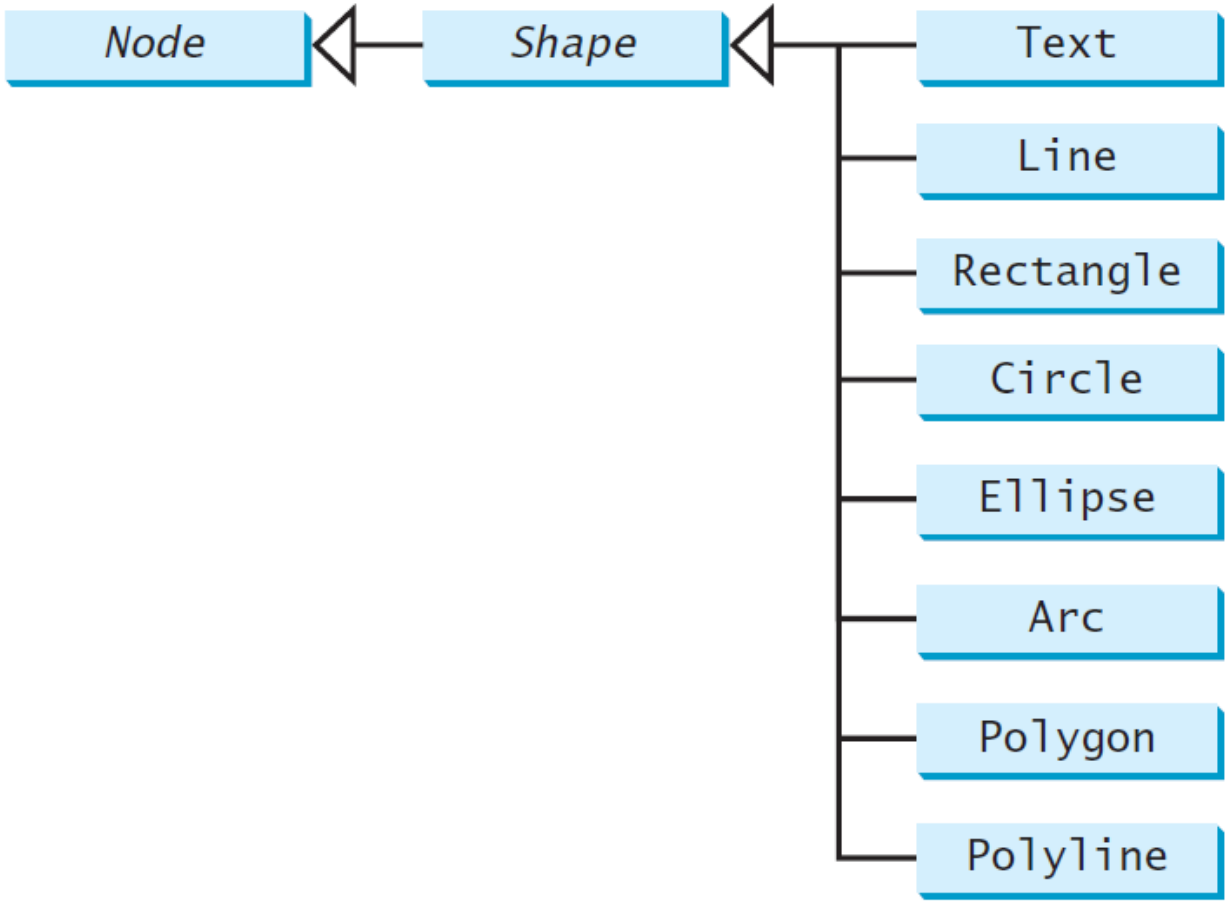
Sets the margin for the node in the pane.

ShowHBoxVBox

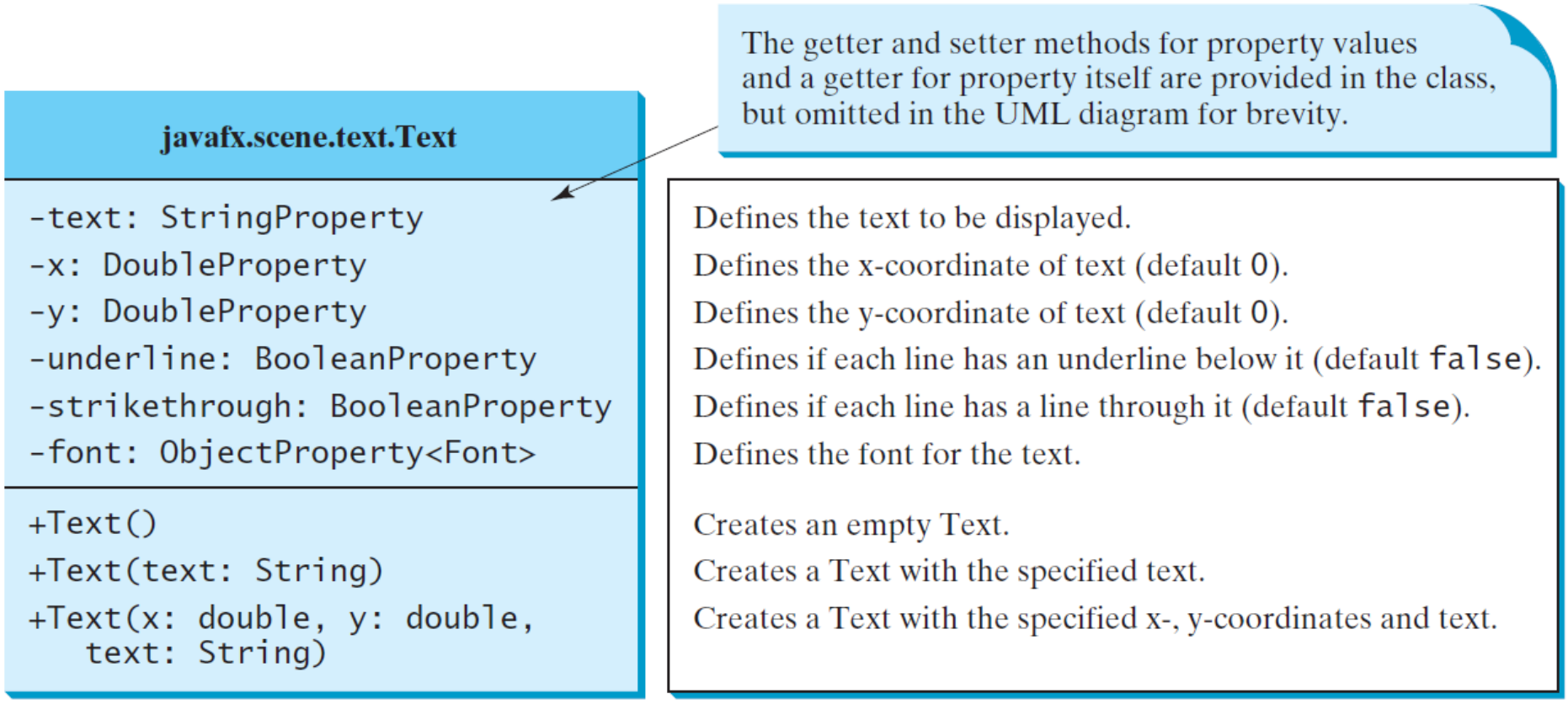
Run

Shapes

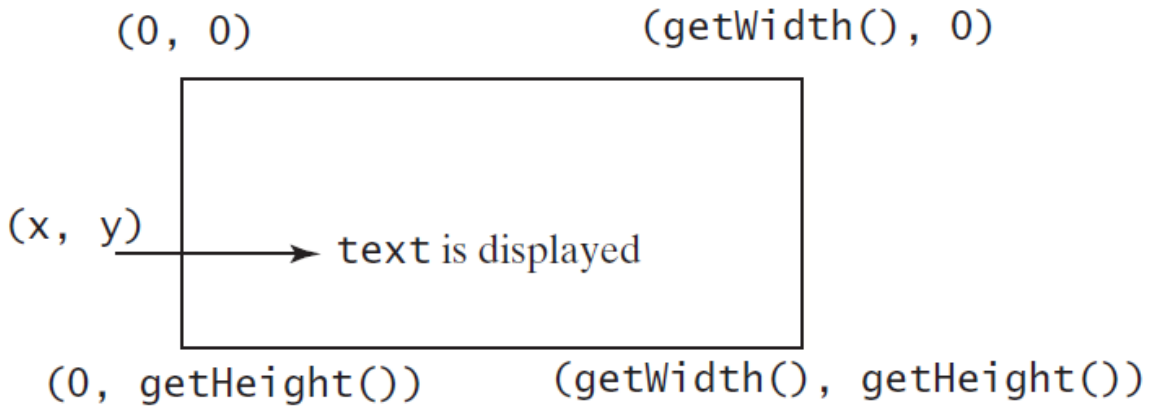
JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.



Text


javafx.scene.text.Text
-text: StringProperty
-x: DoubleProperty
-y: DoubleProperty
-underline: BooleanProperty
-strikethrough: BooleanProperty
-font: ObjectProperty<Font>
+Text()
+Text(text: String)
+Text(x: double, y: double, text: String)
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
Defines the text to be displayed.
Defines the x-coordinate of text (default 0).
Defines the y-coordinate of text (default 0).
Defines if each line has an underline below it (default false).
Defines if each line has a line through it (default false).
Defines the font for the text.
Creates an empty Text.
Creates a Text with the specified text.
Creates a Text with the specified x-, y-coordinates and text.

Text Example



(a) Text(x, y, text)

(b) *Three* Text *objects are displayed*

ShowText

Run

Line

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

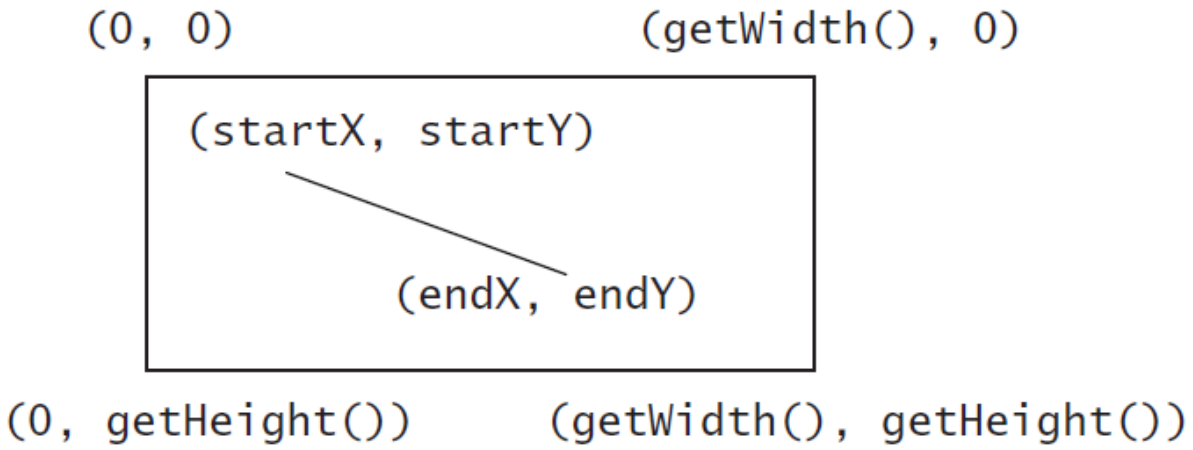
javafx.scene.shape.Line

```
-startX: DoubleProperty
-startY: DoubleProperty
-endX: DoubleProperty
-endY: DoubleProperty
```

```
+Line()
+Line(startX: double, startY:
   double, endX: double, endY:
   double)
```

The x-coordinate of the start point.
The y-coordinate of the start point.
The x-coordinate of the end point.
The y-coordinate of the end point.

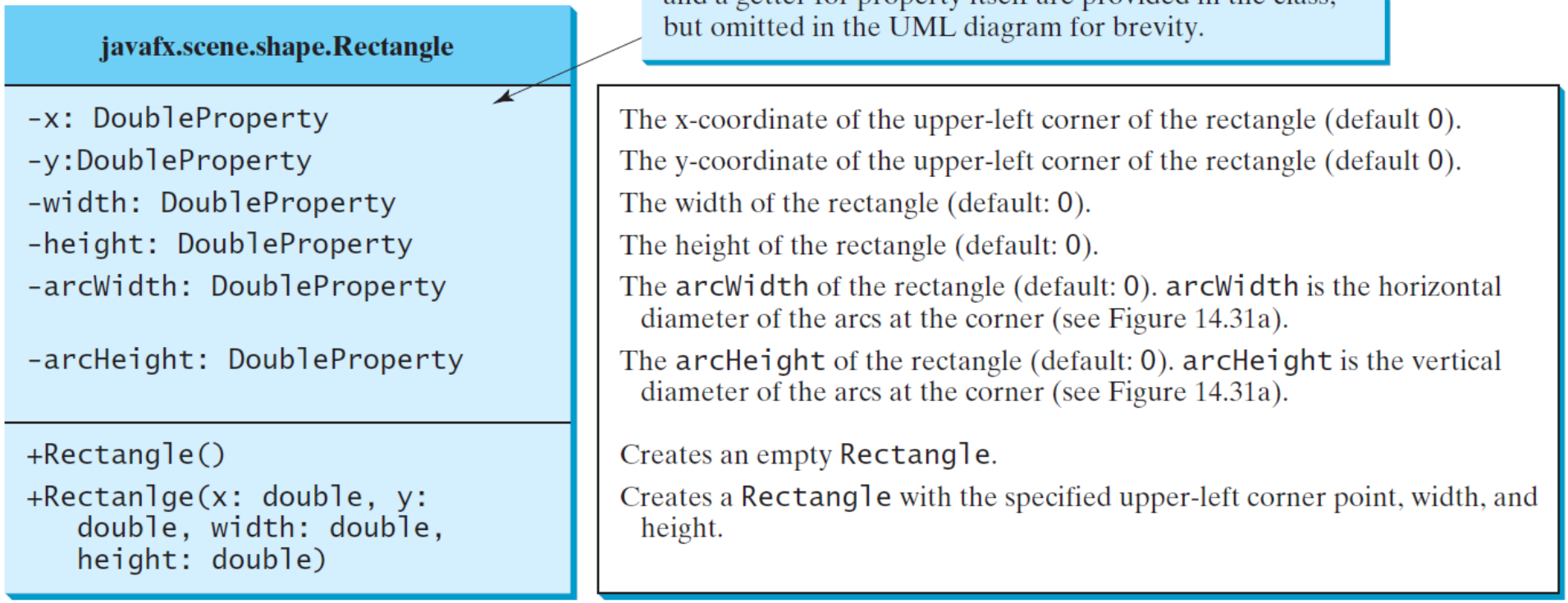
Creates an empty `Line`.
Creates a `Line` with the specified starting and ending points.



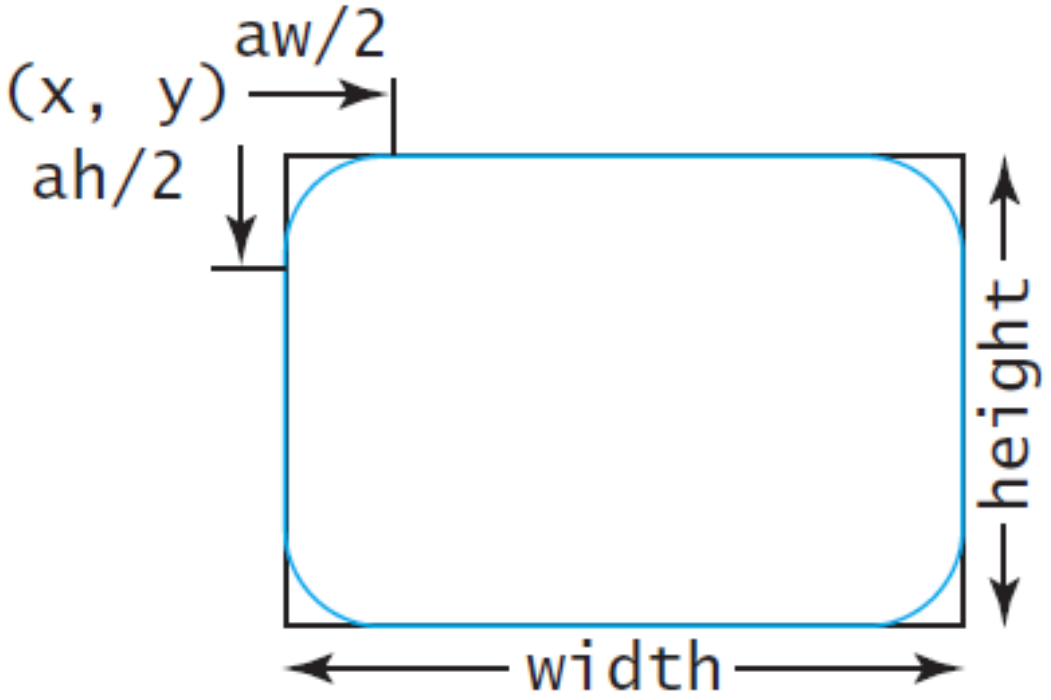
ShowLine

Run

Rectangle


javafx.scene.shape.Rectangle
-x: DoubleProperty
-y:DoubleProperty
-width: DoubleProperty
-height: DoubleProperty
-arcWidth: DoubleProperty
-arcHeight: DoubleProperty
+Rectangle()
+Rectanlge(x: double, y: double, width: double, height: double)
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
The x-coordinate of the upper-left corner of the rectangle (default 0).
The y-coordinate of the upper-left corner of the rectangle (default 0).
The width of the rectangle (default: 0).
The height of the rectangle (default: 0).
The arcWidth of the rectangle (default: 0). arcWidth is the horizontal diameter of the arcs at the corner (see Figure 14.31a).
The arcHeight of the rectangle (default: 0). arcHeight is the vertical diameter of the arcs at the corner (see Figure 14.31a).
Creates an empty Rectangle.
Creates a Rectangle with the specified upper-left corner point, width, and height.

Rectangle Example

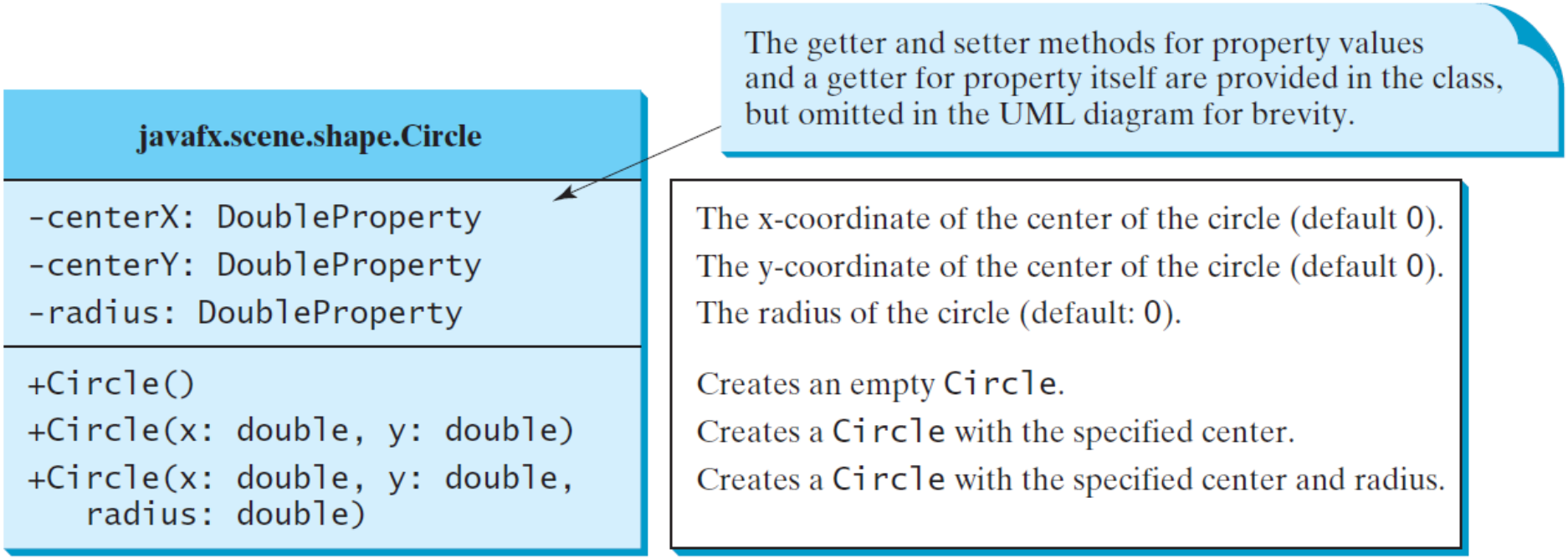


(a) Rectangle(x, y, w, h)

ShowRectangle

Run

Circle


javafx.scene.shape.Circle
-centerX: DoubleProperty
-centerY: DoubleProperty
-radius: DoubleProperty
+Circle()
+Circle(x: double, y: double)
+Circle(x: double, y: double, radius: double)
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
The x-coordinate of the center of the circle (default 0).
The y-coordinate of the center of the circle (default 0).
The radius of the circle (default: 0).
Creates an empty Circle.
Creates a Circle with the specified center.
Creates a Circle with the specified center and radius.

Ellipse

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

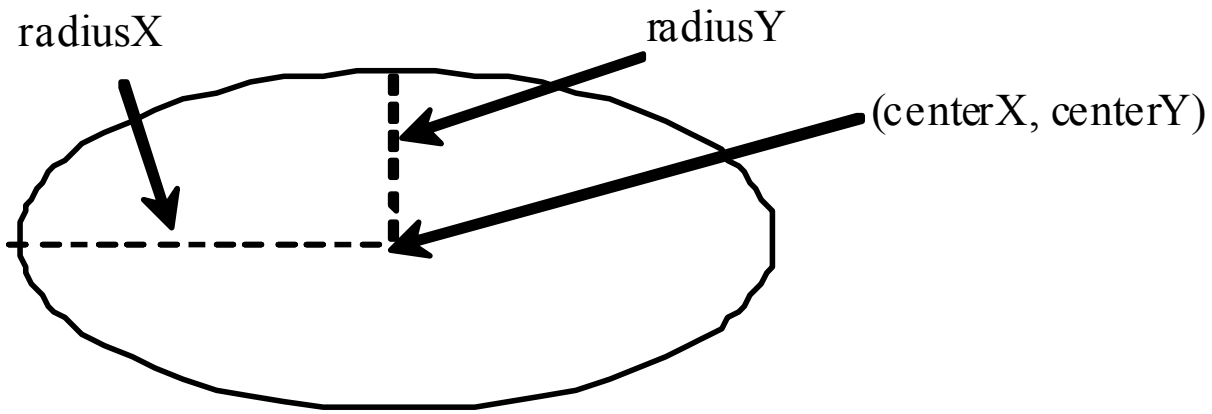
javafx.scene.shape.Ellipse

```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
```

```
+Ellipse()
+Ellipse(x: double, y: double)
+Ellipse(x: double, y: double,
   radiusX: double, radiusY:
   double)
```

The x-coordinate of the center of the ellipse (default 0).
The y-coordinate of the center of the ellipse (default 0).
The horizontal radius of the ellipse (default: 0).
The vertical radius of the ellipse (default: 0).

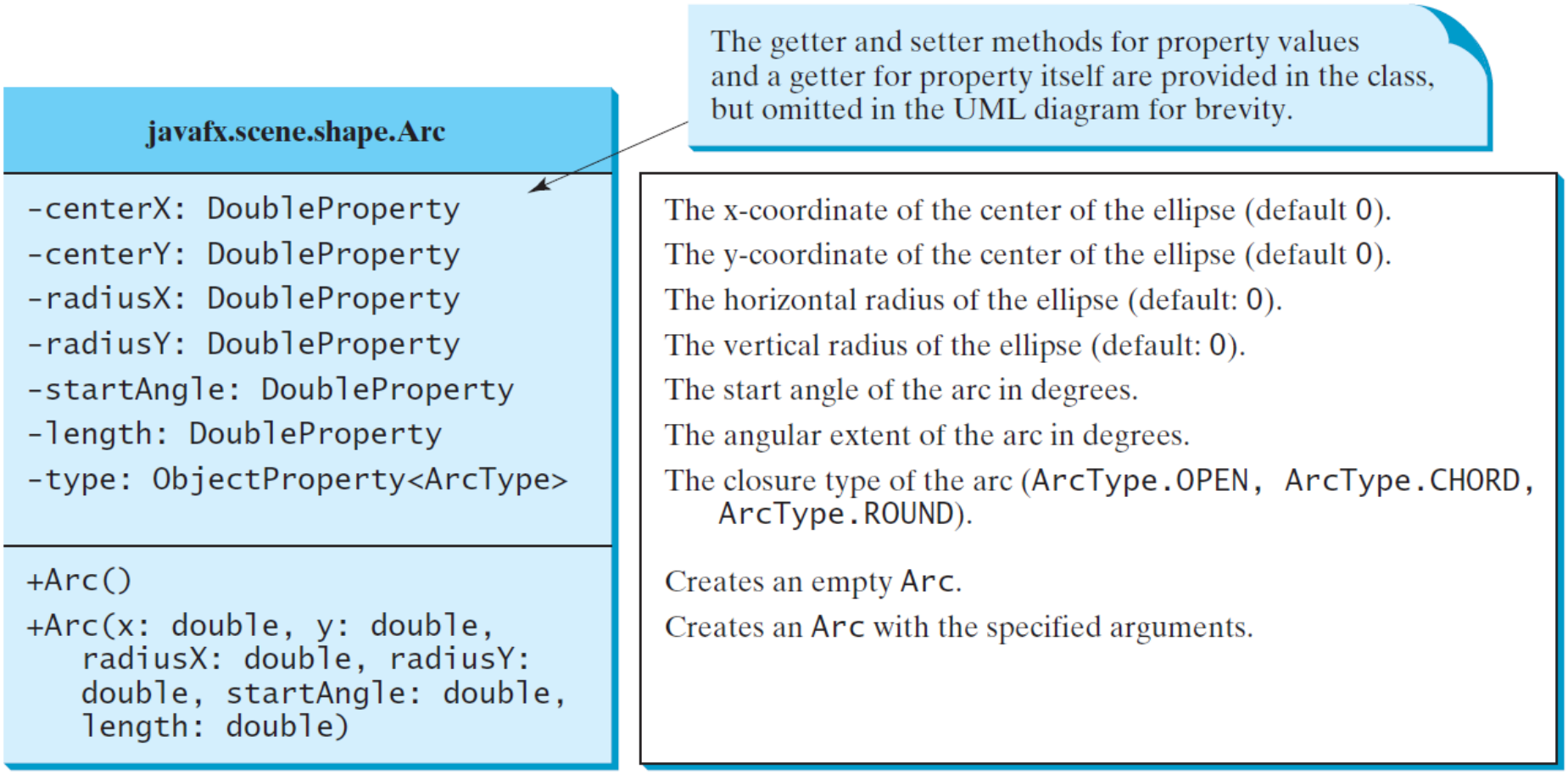
Creates an empty `Ellipse`.
Creates an `Ellipse` with the specified center.
Creates an `Ellipse` with the specified center and radiuses.



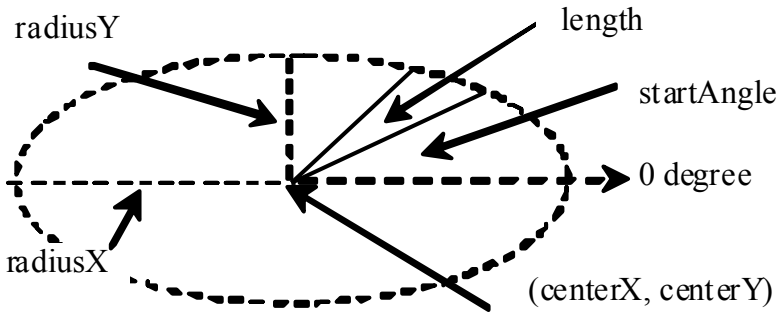
ShowEllipse

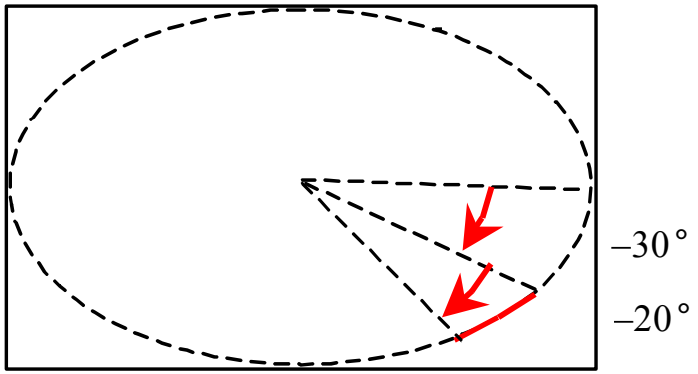
Run

Arc

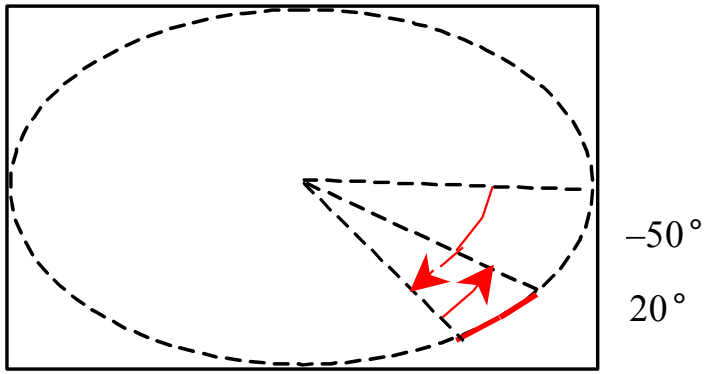

javafx.scene.shape.Arc
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
-startAngle: DoubleProperty
-length: DoubleProperty
-type: ObjectProperty<ArcType>
+Arc()
+Arc(x: double, y: double, radiusX: double, radiusY: double, startAngle: double, length: double)
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.
The x-coordinate of the center of the ellipse (default 0).
The y-coordinate of the center of the ellipse (default 0).
The horizontal radius of the ellipse (default: 0).
The vertical radius of the ellipse (default: 0).
The start angle of the arc in degrees.
The angular extent of the arc in degrees.
The closure type of the arc (ArcType.OPEN, ArcType.CHORD, ArcType.ROUND).
Creates an empty Arc.
Creates an Arc with the specified arguments.

Arc Examples





(a) Negative starting angle –30° and negative spanning angle –20°

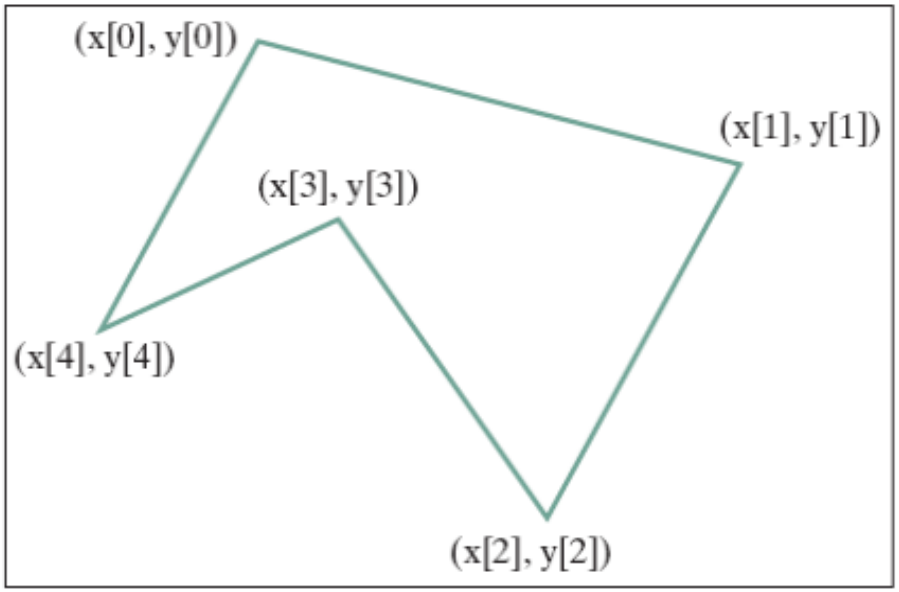


(b) Negative starting angle –50° and positive spanning angle 20°

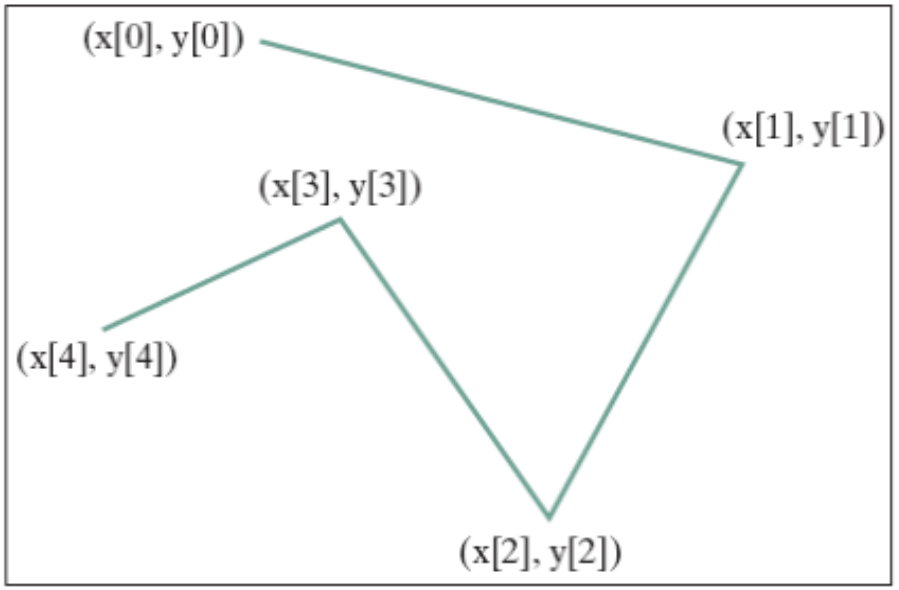
ShowArc

Run

Polygon and Polyline



(a) Polygon



(b) Polyline

ShowArc

Run

Polygon

javafx.scene.shape.Polygon
+Polygon()
+Polygon(double... points)
+getPoints(): ObservableList<Double>

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Creates an empty polygon.

Creates a polygon with the given points.

Returns a list of double values as x- and y-coordinates of the points.

ShowPolygon

Run

Old GUI Elements

- Three kinds of objects are needed to create a GUI in Java
  - Components (we start to cover them in this chapter)
  - Events (future work)
  - Listeners (future work)
- *Component* – an object that defines a screen element used to display information or allow the user to interact with the program
  - A *container* is a special type of component that is used to hold and organize other components

Old GUI Components

- A *GUI component* is an object that represents a screen element such as a button or a text field
- GUI-related classes are defined primarily in the `java.awt` and the `javax.swing` packages
- The *Abstract Windowing Toolkit* (AWT) was the original Java GUI package
- The *Swing* package provides additional and more versatile components
- Both packages are needed to create a Java GUI-based program

GUI Containers

- A *GUI container* is a component that is used to hold and organize other components
- A *frame* is a container that is used to display a GUI-based Java application
- A frame is displayed as a separate window with a title bar – it can be repositioned and resized on the screen as needed
- A *panel* is a container that cannot be displayed on its own but is used to organize other components
- A panel must be added to another container to be displayed
- Panels can be contained within other panels!

GUI Libraries

- JavaFX- now standard with Java 8 and later
  - Newer version using terms like Scene and Stage like in a theater.
- Java AWT and Swing- older more prevalent libraries- since most of the underlying code you may see is composed of these we will start here.

AWT

- When Java was introduced, the GUI classes were bundled in a library known as the
- 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*. AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects. Besides, AWT is prone to platform-specific bugs because its peer-based approach relies heavily on the underlying platform.

Swing

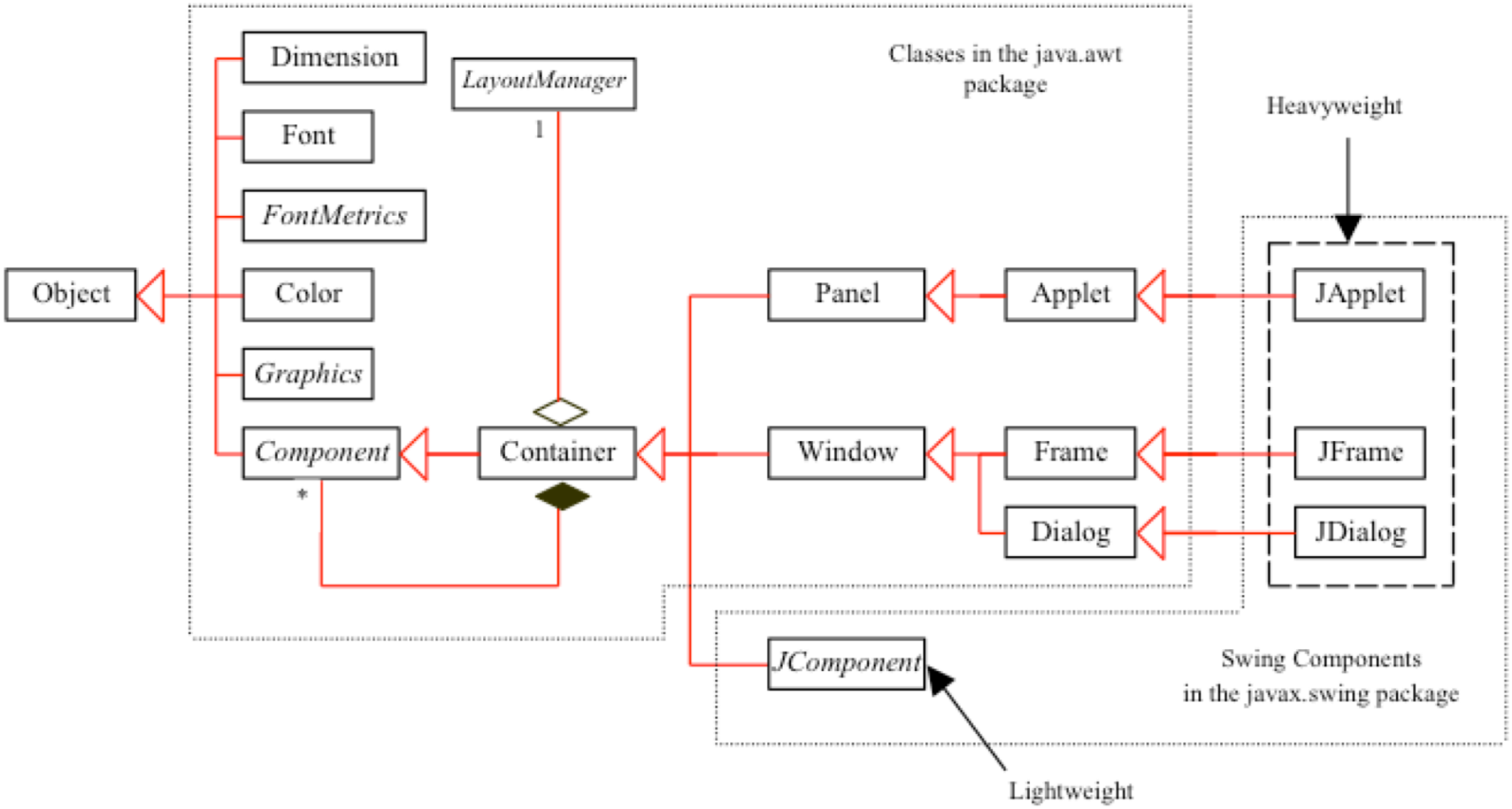
In Java 2 introduced a more robust, versatile, and flexible library known as *Swing components*.

Swing components are painted directly on canvases using Java code, except for components that are subclasses of java.awt.Window or java.awt.Panel, which must be drawn using native GUI on a specific platform.

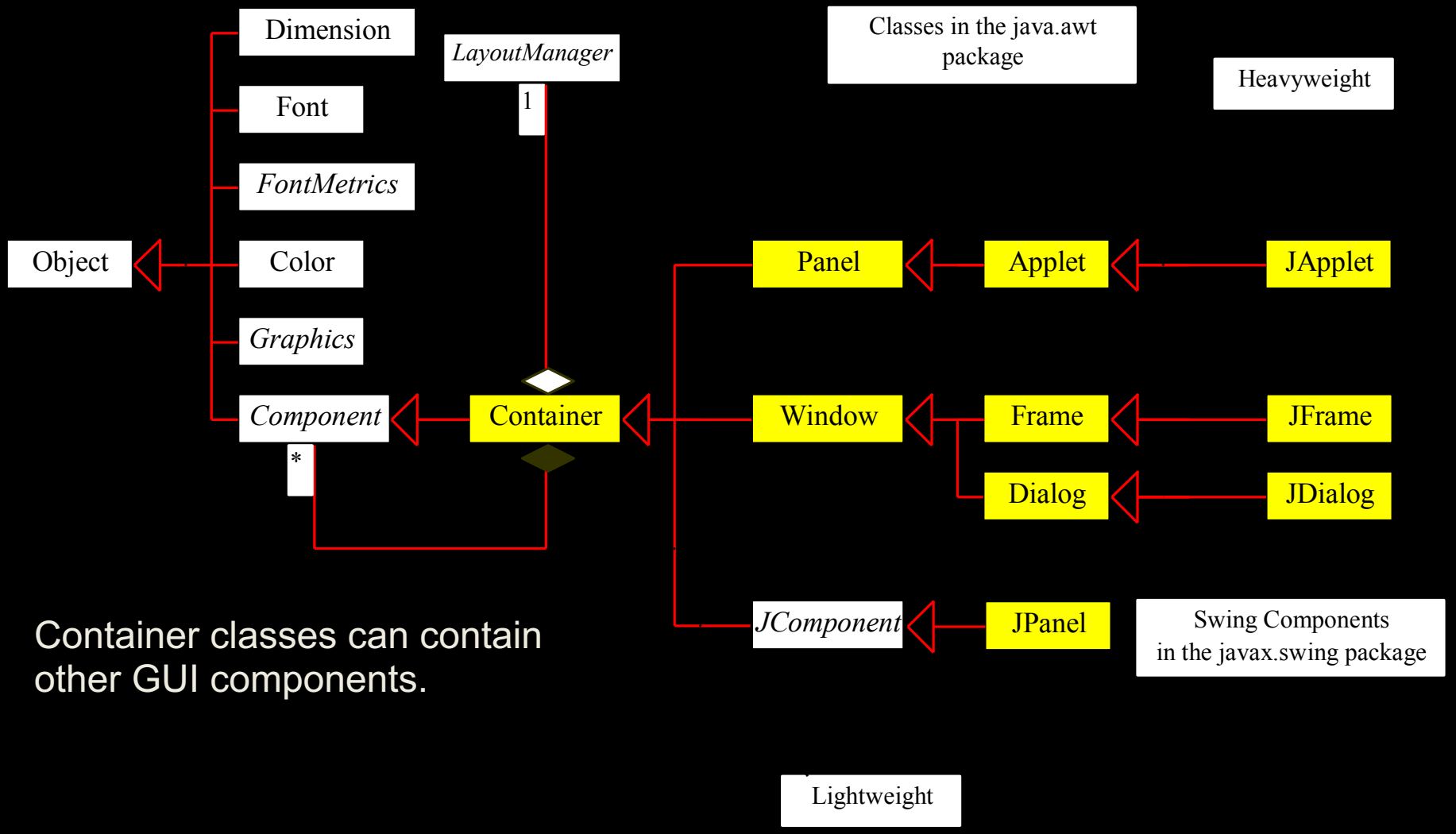
Heavy vs Lightweight Components

- Swing components are less dependent on the target platform and use less of the native GUI resource than AWT components. For this reason, Swing components that don't rely on native GUI are referred to as *lightweight components*, and AWT components are referred to as *heavyweight components*.
- Heavyweight components depend on the operating system for control of the screen- A MacOS screen will appear and function differently than a MS Windows screen.

GUI Class Hierarchy

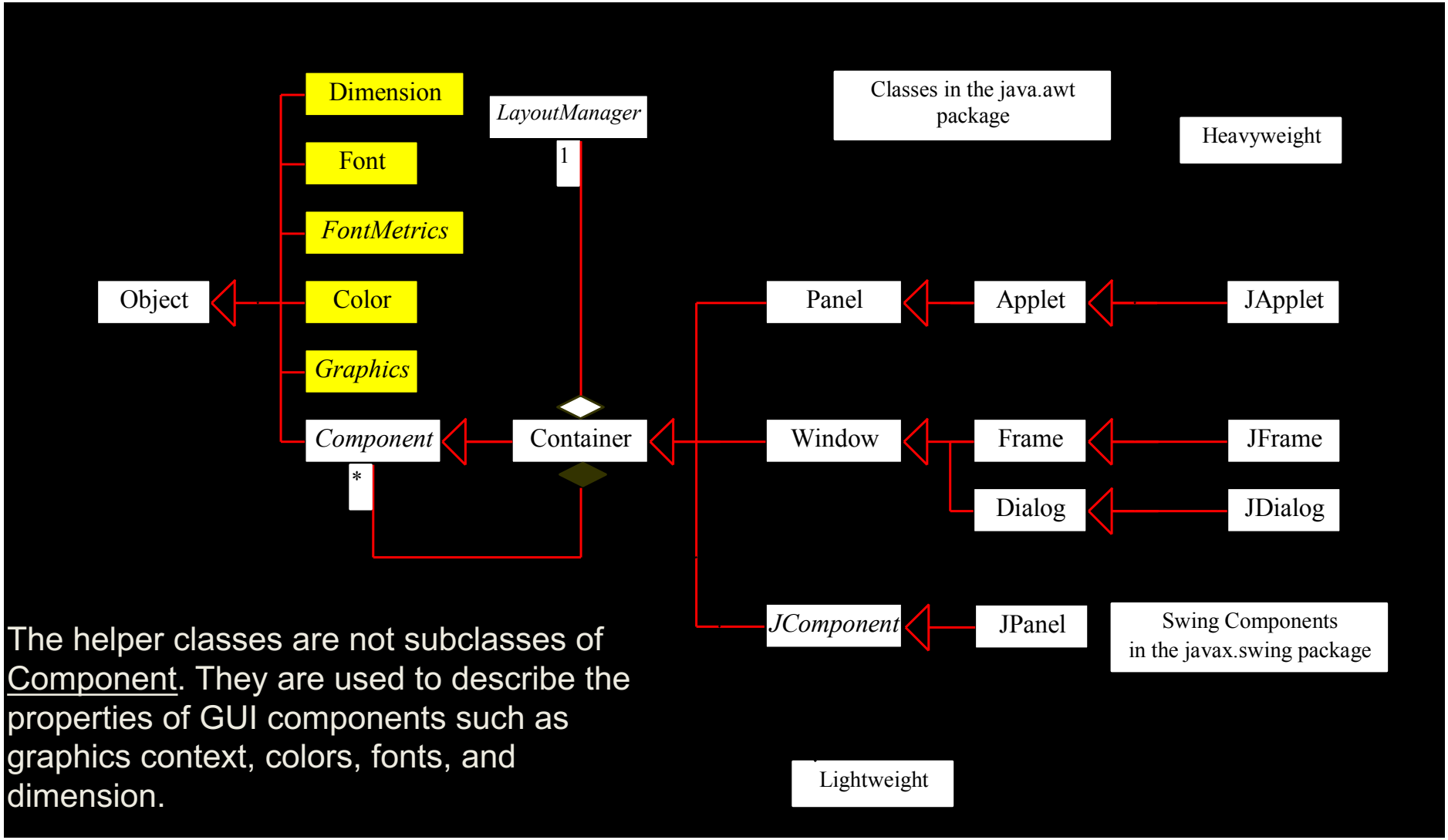


Container Classes



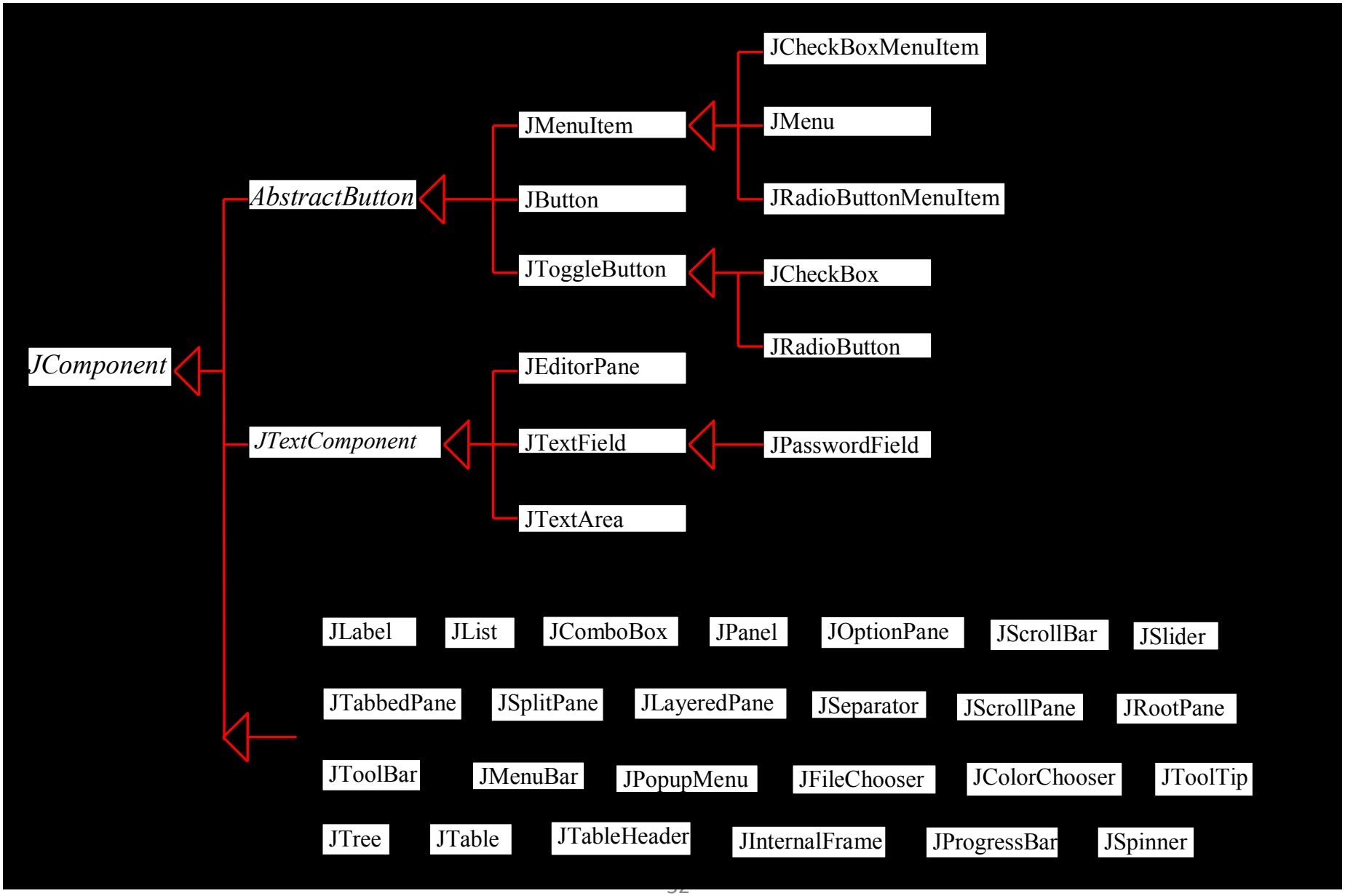
Container classes can contain other GUI components.

GUI Helper Classes

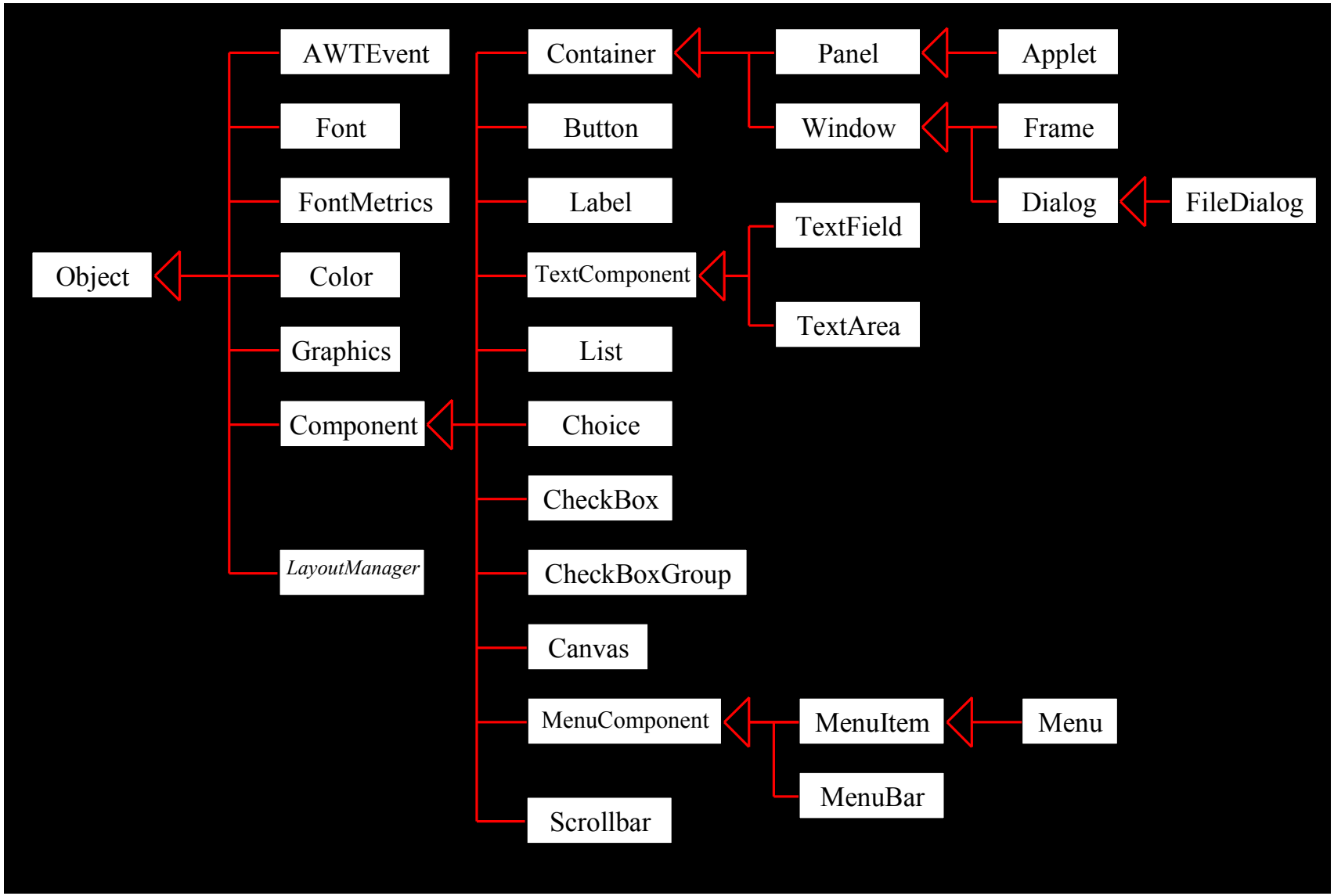


The helper classes are not subclasses of Component. They are used to describe the properties of GUI components such as graphics context, colors, fonts, and dimension.

Swing GUI Components



AWT (Optional)



Frames

- Frame is a window that is not contained inside another window. Frame is the basis to contain other user interface components in Java GUI applications.
- The JFrame class can be used to create windows.
- Frames are Heavyweight Components under OS control

JFrame Class

javax.swing.JFrame	
+JFrame()	Creates a default frame with no title.
+JFrame(title: String)	Creates a frame with the specified title.
+setSize(width: int, height: int): void	Specifies the size of the frame.
+setLocation(x: int, y: int): void	Specifies the upper-left corner location of the frame.
+setVisible(visible: boolean): void	Sets true to display the frame.
+setDefaultCloseOperation(mode: int): void	Specifies the operation when the frame is closed.
+setLocationRelativeTo(c: Component): void	Sets the location of the frame relative to the specified component. If the component is null, the frame is centered on the screen.
+pack(): void	Automatically sets the frame size to hold the components in the frame.

Creating Frames

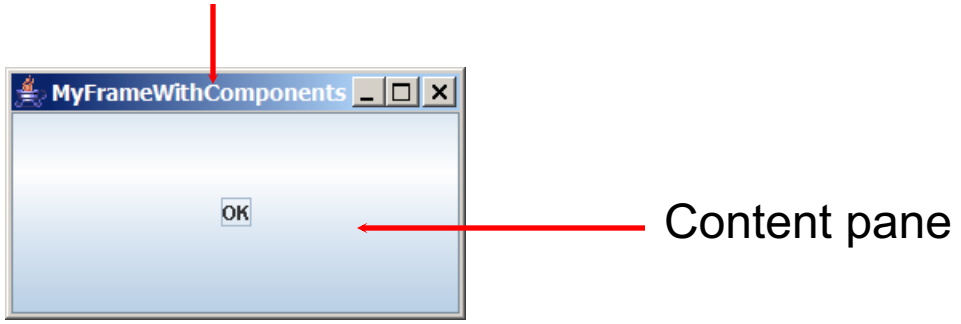
```
import javax.swing.*;
public class MyFrame {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Test Frame");
    frame.setSize(400, 300);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(
      JFrame.EXIT_ON_CLOSE);
  }
}
```

MyFrame

Run

Adding Components into a Frame

```
// Add a button into the frame
frame.getContentPane().add(
  new JButton("OK"));
```



Layout Managers

- Java's layout managers provide a level of abstraction to automatically map your user interface on all window systems.

- The UI components are placed in containers. Each container has a layout manager to arrange the UI components within the container.

- Layout managers are set in containers using the setLayout(LayoutManager) method in a container.

Layout Managers

- Every container is managed by an object known as a *layout manager* that determines how the components in the container are arranged visually
- The layout manager is consulted when needed, such as when the container is resized or when a component is added
- Every container has a default layout manager, but we can replace it if desired
- A layout manager determines the size and position of each component
- For some layout managers, the order in which you add the components affects their positioning
- We use the `setLayout` method of a container to change its layout manager

Predefined Layout Managers

Layout Manager	Description
Border Layout	Organizes components into five areas (North, South, East, West, and Center)
Box Layout	Organizes components into a single row or column
Card Layout	Organizes components into one area such that only one area is visible at a time
Flow Layout	Organizes components from left to right, starting new rows as necessary
Grid Layout	Organizes components into a grid of rows and columns
GridBag Layout	Organizes components into a grid of cells, allowing components to span more than one cell

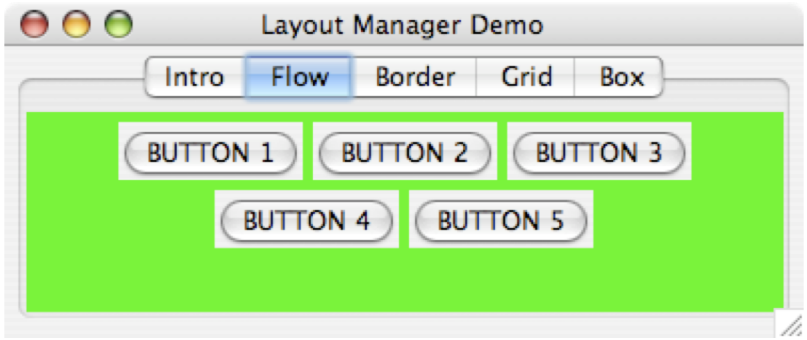
Flow Layout

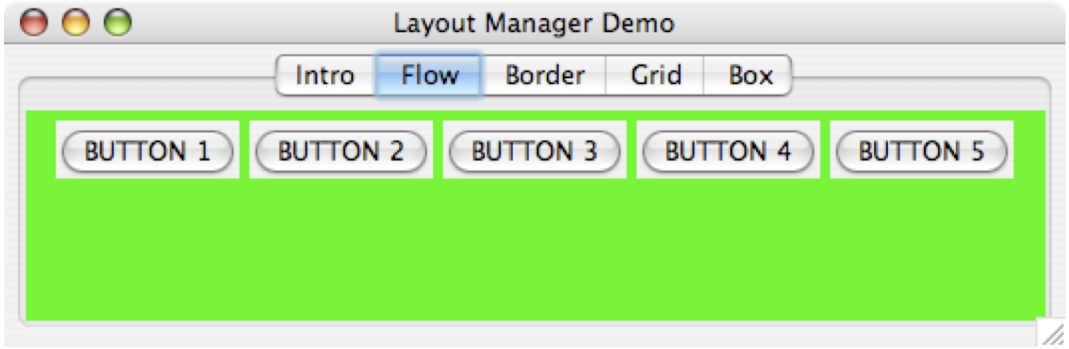
- One of the easiest layout managers to use
- The `JPanel` class uses flow layout by default
- Puts as many components as possible on a row, at their preferred size
- When a component can not fit on a row, it is put on the next row

FlowPanel.java

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

public class FlowPanel extends Jpanel {
public FlowPanel ()    {
      setLayout (new FlowLayout());

      setBackground (Color.green);

      JButton b1 = new JButton ("BUTTON 1");
      JButton b2 = new JButton ("BUTTON 2");
      JButton b3 = new JButton ("BUTTON 3");
      JButton b4 = new JButton ("BUTTON 4");
      JButton b5 = new JButton ("BUTTON 5");

      add (b1);
      add (b2);
      add (b3);
      add (b4);
      add (b5);
   }}
```

Layout Manager Demo

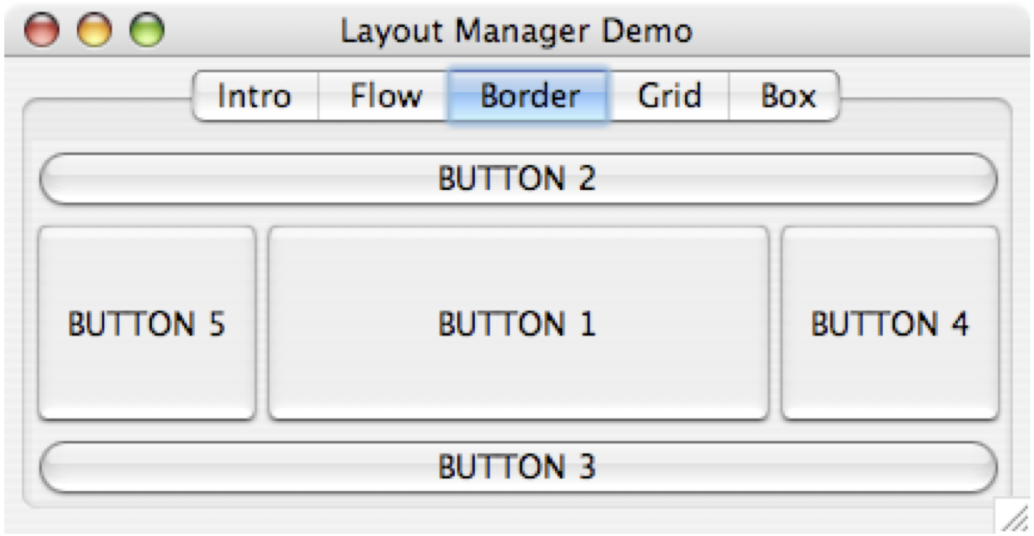


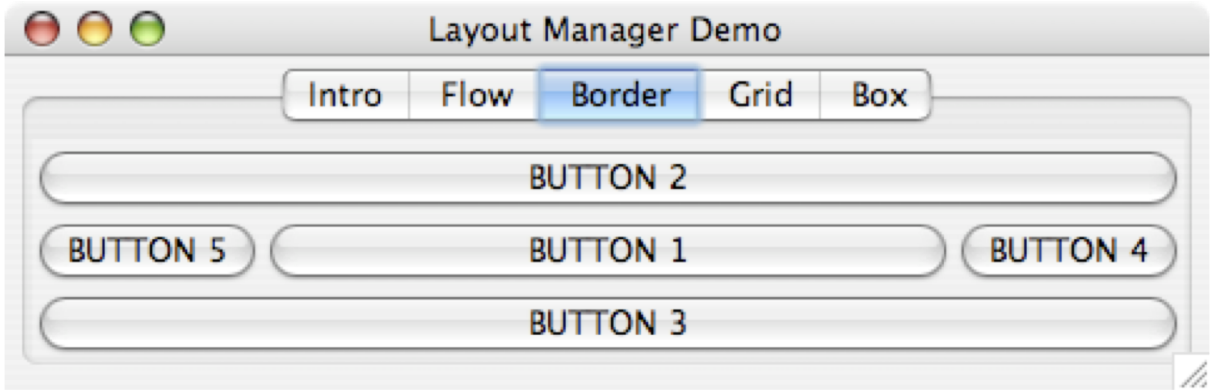


Border Layout

- A *border layout* has five areas to which components can be added: North, South, East, West, and Center
- The four outer areas are as large as needed in order to accommodate the component they contain
- If no components are added to a region, the region takes up no room in the overall layout
- The Center area expands to fill any available space

Layout Manager Demo





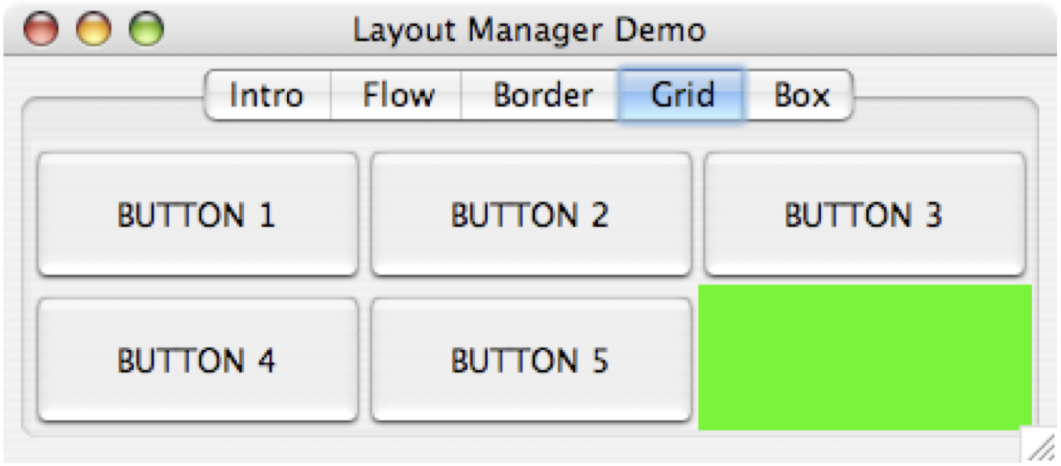
BorderPanel.java

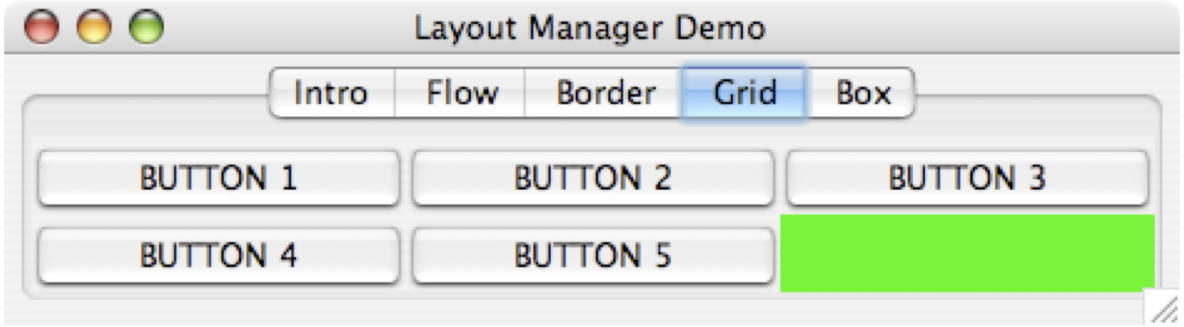
```
import java.awt.*;
import javax.swing.*;

public class BorderPanel extends Jpanel {
public BorderPanel()    {
      setLayout (new BorderLayout());
      setBackground (Color.green);
 JButton b1 = new JButton ("BUTTON 1");
      JButton b2 = new JButton ("BUTTON 2");
      JButton b3 = new JButton ("BUTTON 3");
      JButton b4 = new JButton ("BUTTON 4");
      JButton b5 = new JButton ("BUTTON 5");

      add (b1, BorderLayout.CENTER);
      add (b2, BorderLayout.NORTH);
      add (b3, BorderLayout.SOUTH);
      add (b4, BorderLayout.EAST);
      add (b5, BorderLayout.WEST);
   }
}
```

Grid Layout

- A *grid layout* presents a container's components in a rectangular grid of rows and columns
- One component is placed in each cell, and all cells are the same size
- The number of rows and columns in a grid layout is established by using parameters to the constructor when the layout manager is created

Layout Manager Demo

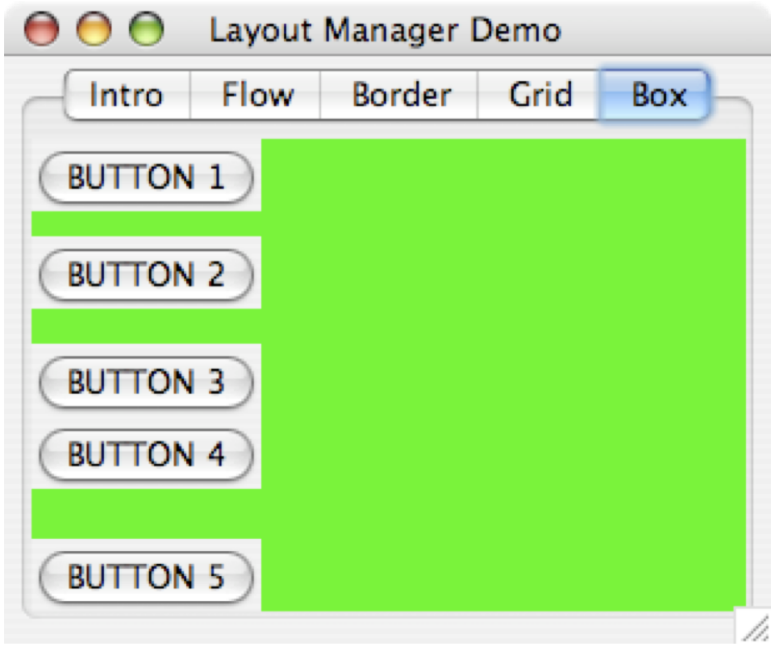


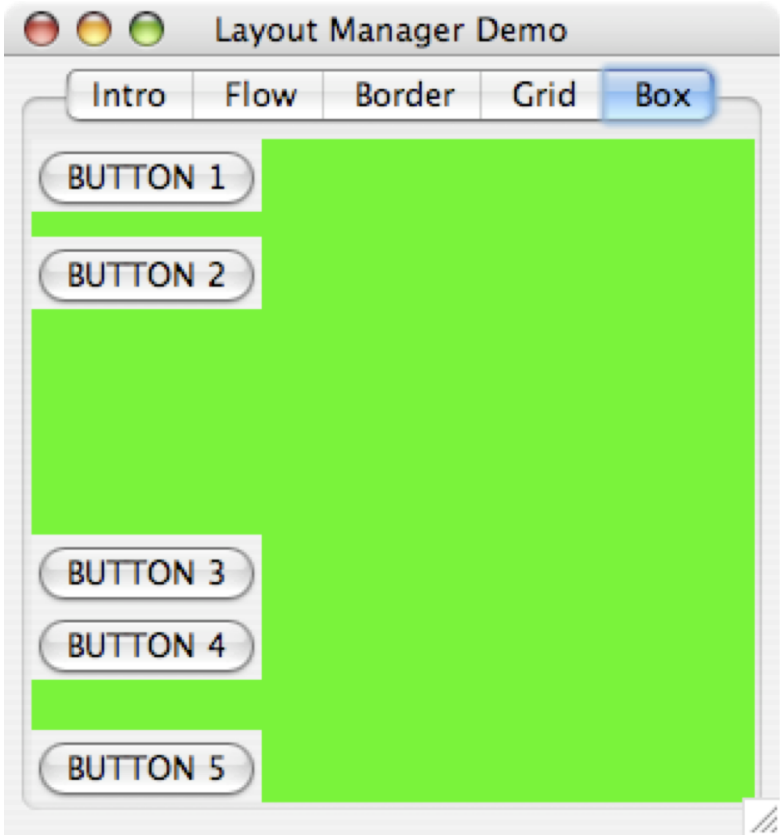


Box Layout

- A *box layout* organizes components either vertically or horizontally, in one row or one column
- It is easy to use and when combined with other layout managers, can produce complex GUI designs
- Components are organized in the order in which they are added to the container
- There are no gaps between the components in a box layout

Layout Manager Demo





The `Font` Class

Font Names

Standard font names that are supported in all platforms are: SansSerif, Serif, Monospaced, Dialog, or DialogInput.

Font Style

Font.PLAIN (0), Font.BOLD (1), Font.ITALIC (2), and Font.BOLD + Font.ITALIC (3)

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

Example:

```
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);
```

Finding All Available Font Names

```
GraphicsEnvironment e =
  GraphicsEnvironment.getLocalGraphicsEnvironment();
String[] fontnames =
  e.getAvailableFontFamilyNames();
for (int i = 0; i < fontnames.length; i++)
  System.out.println(fontnames[i]);
```

Containment Hierarchies

- The way components are grouped into containers, and the way those containers are nested within each other, establishes the *containment hierarchy* for a GUI
- For any Java GUI program, there is generally one primary (top-level) container, such as a frame or applet
- The top-level container often contains one or more containers, such as panels
- These panels may contain other panels to organize the other components as desired

Using Panels as Sub-Containers

- Panels act as sub-containers for grouping user interface components.
- It is recommended that you place the user interface components in panels and place the panels in a frame. You can also place panels in a panel.
- To add a component to JFrame, you actually add it to the content pane of JFrame. To add a component to a panel, you add it directly to the panel using the add method.

Creating a JPanel

You can use new JPanel() to create a panel with a default FlowLayout manager or new JPanel(LayoutManager) to create a panel with the specified layout manager. Use the add(Component) method to add a component to the panel. For example,

```
JPanel p = new JPanel();
p.add(new JButton("OK"));
```

Borders

You can set a border on any object of the JComponent class. Swing has several types of borders. To create a titled border, use

new TitledBorder(String title).

To create a line border, use

new LineBorder(Color color, int width),

where width specifies the thickness of the line. For example, the following code displays a titled border on a panel:

```
JPanel panel = new JPanel();
panel.setBorder(new TitleBorder(“My Panel”));
```

Borders

Border	Description
Empty Border	Puts a buffering space around the edge of a component, but otherwise has no visual effect
Line Border	A simple line surrounding the component
Etched Border	Creates the effect of an etched groove around a component
Bevel Border	Creates the effect of a component raised above the surface or sunken below it
Titled Border	Includes a text title on or around the border
Matte Border	Allows the size of each edge to be specified. Uses either a solid color or an image
Compound Border	A combination of two borders

Borders

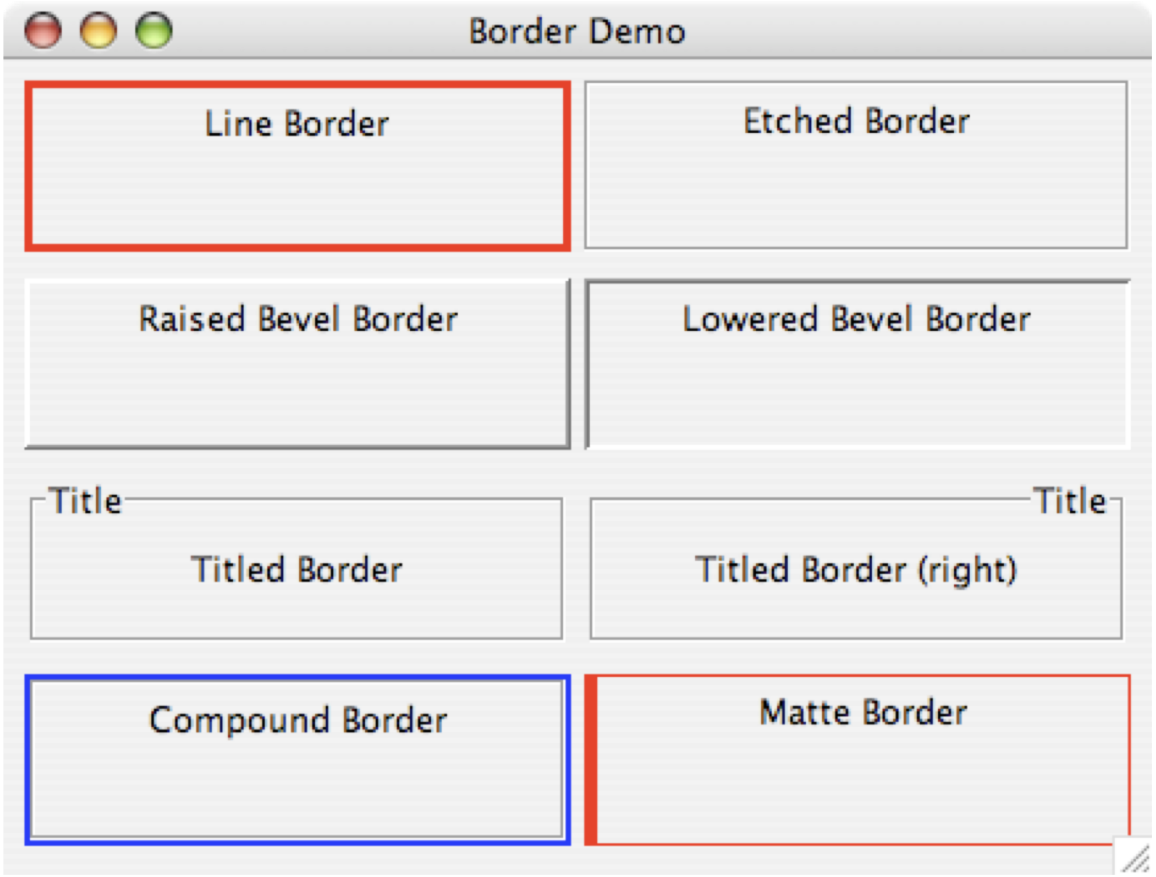

Border Demo
Line Border
Etched Border
Raised Bevel Border
Lowered Bevel Border
Title
Titled Border
Title
Titled Border (right)
Compound Border
Matte Border

Image Icons

Java uses the javax.swing.ImageIcon class to represent an icon. An icon is a fixed-size picture; typically it is small and used to decorate components. Images are normally stored in image files. You can use new ImageIcon(filename) to construct an image icon. For example, the following statement creates an icon from an image file us.gif in the image directory under the current class path:

```
ImageIcon icon = new ImageIcon("image/us.gif");
```

Splash Screen

A *splash screen* is an image that is displayed while the application is starting up. If your program takes a long time to load, you may display a splash screen to alert the user. For example, the following command:

```
java –splash:image/us.gf TestImageIcon
```

displays an image while the program TestImageIcon is being loaded.

Buttons

A *button* is a component that triggers an action event when clicked. Swing provides regular buttons, toggle buttons, check box buttons, and radio buttons. The common features of these buttons are generalized in javax.swing.AbstractButton.

Chapter 12 shows you how to put a button on the screen- later chapters work on adding events and listeners to the buttons so they actually perform a task.

Buttons and Action Events

- `PushCounter` displays
  - *label* - displays a line of text
    - can be used to also display an image
    - labels are non-interactive
  - *push button* – allows the user to initiate an action with a press of the mouse
    - generates an *action event*
    - different components generate different types of events
- `ButtonListener` class represents the action listener for the button

JButton Constructors

The following are `JButton` constructors:

```
JButton()

JButton(String text)

JButton(String text, Icon icon)

JButton(Icon icon)
```

More Components

- In addition to push buttons, there are variety of other interactive components
  - *text fields* – allows the user to enter typed input from the keyboard
  - *check boxes* – a button that can be toggled on or off using the mouse (indicates a boolean value is set or unset)
  - *radio buttons* – used with other radio buttons to provide a set of mutually exclusive options
  - *sliders* – allows the user to specify a numeric value within a bounded range
  - *combo boxes* – allows the user to select one of several options from a "drop down" menu
  - *timers* – helps us manage an activity over time, has no visual representation

Text Fields

- A text field generates an action event when the Enter or Return key is pressed (and the cursor is in the field)
- Note that the push button and the text field generate the same kind of event – an action event
- An alternative implementation could involve adding a push button to the panel which causes the conversion to occur when the user pushes the button

Default Icons, Pressed Icon, and Rollover Icon

A regular button has a default icon, pressed icon, and rollover icon. Normally, you use the default icon. All other icons are for special effects. A pressed icon is displayed when a button is pressed and a rollover icon is displayed when the mouse is over the button but not pressed.

(A) Default icon icon

(B) Pressed icon

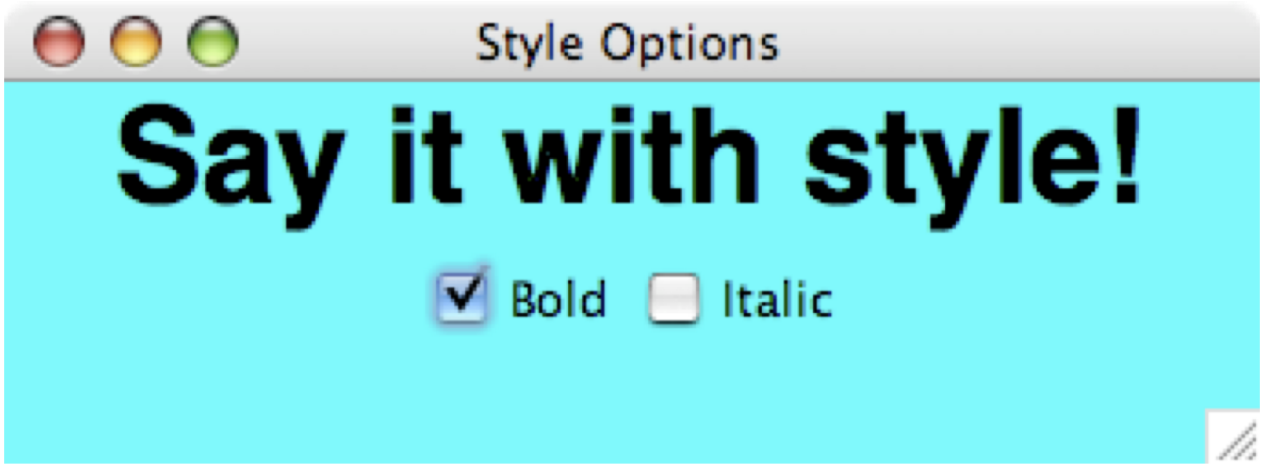
(C) Rollover

Demo

Check Boxes

- A check box generates an *item event* when it changes state from selected (checked) to deselected (unchecked) and vice versa
- In the example, we use the same listener to handle both check boxes (bold and italic)
- The example also uses the `Font` class to display and change the text label
- Style of the font is represented as an integer, integer constants defined in the class are used to represent aspects of style
- The `JCheckBox` class is used to define check boxes

Style Options Application



StyleOptionsPanel.java

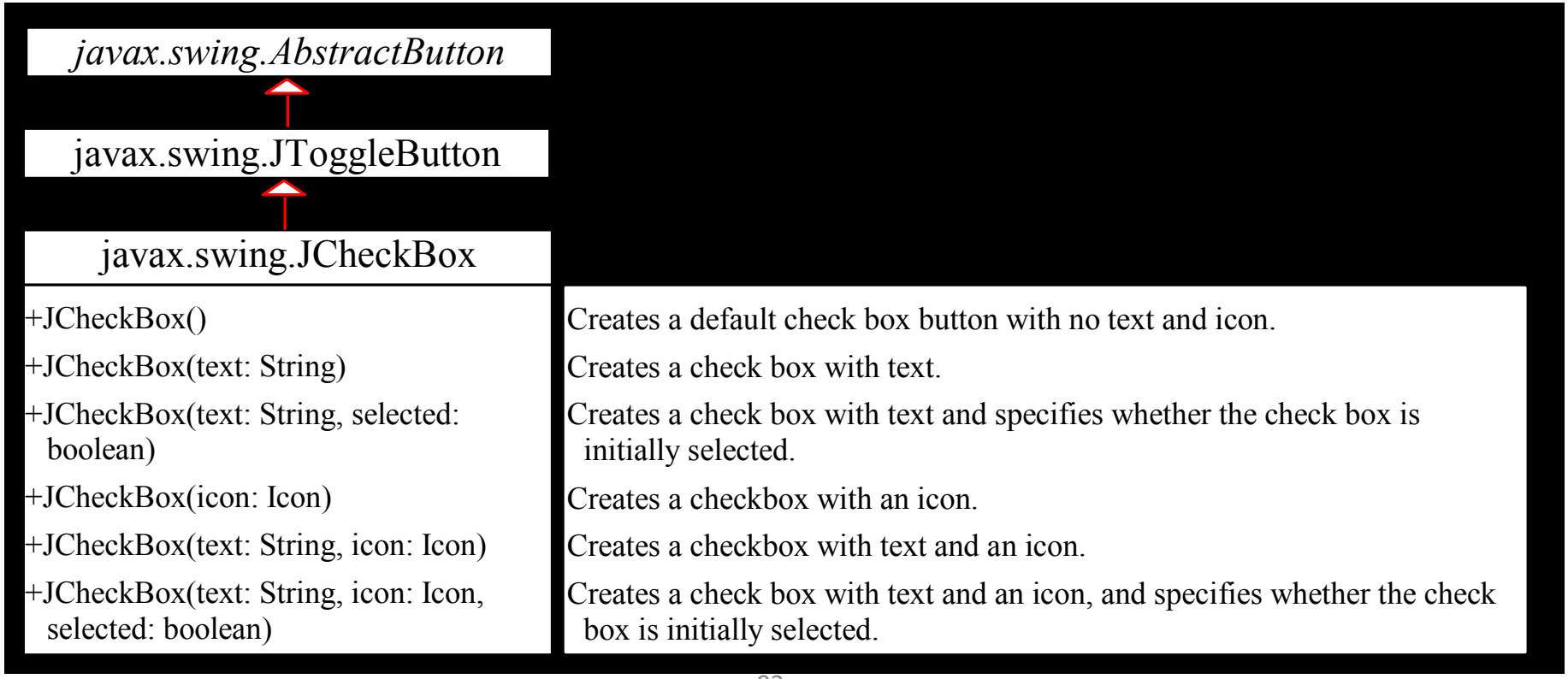
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class StyleOptionsPanel extends JPanel{
   private JLabel saying;
   private JCheckBox bold, italic;
 public StyleOptionsPanel()    {
      saying = new JLabel ("Say it with style!");
      saying.setFont (new Font ("Helvetica", Font.PLAIN, 36));

      bold = new JCheckBox ("Bold");
      bold.setBackground (Color.cyan);
      italic = new JCheckBox ("Italic");
      italic.setBackground (Color.cyan);

      StyleListener listener = new StyleListener();
      bold.addItemListener (listener);
      italic.addItemListener (listener);

      add (saying);
      add (bold);
      add (italic);

      setBackground (Color.cyan);
       setPreferredSize (new Dimension(300, 100));    }
```

StyleOptionsPanel.java

```
   //*****************************************************************
   //  Represents the listener for both check boxes.
   //*****************************************************************
   private class StyleListener implements ItemListener
   {
public void itemStateChanged (ItemEvent event)
      {
         int style = Font.PLAIN;

         if (bold.isSelected())
            style = Font.BOLD;

         if (italic.isSelected())
            style += Font.ITALIC;

         saying.setFont (new Font ("Helvetica", style, 36));
      } } }
```

JCheckBox

JCheckBox inherits all the properties such as text, icon, mnemonic, verticalAlignment, horizontalAlignment, horizontalTextPosition, verticalTextPosition, and selected from AbstractButton, and provides several constructors to create check boxes.



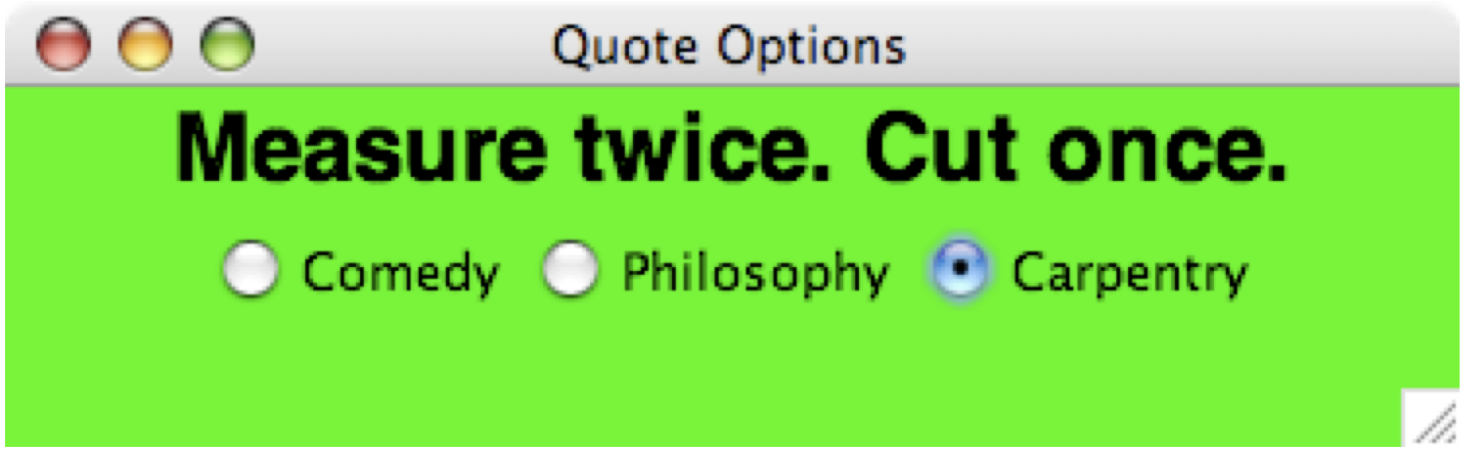
Radio Buttons

- A *radio button* is used with other radio buttons to provide a set of mutually exclusive options
- Radio buttons have meaning only when used with one or more other radio buttons
- At any point in time, only one button of the group is selected (on)
- Radio buttons produce an action event when selected
- Radio buttons are defined by the `JRadioButton` class
- The `ButtonGroup` class is used to define a set of related radio buttons

Grouping Radio Buttons

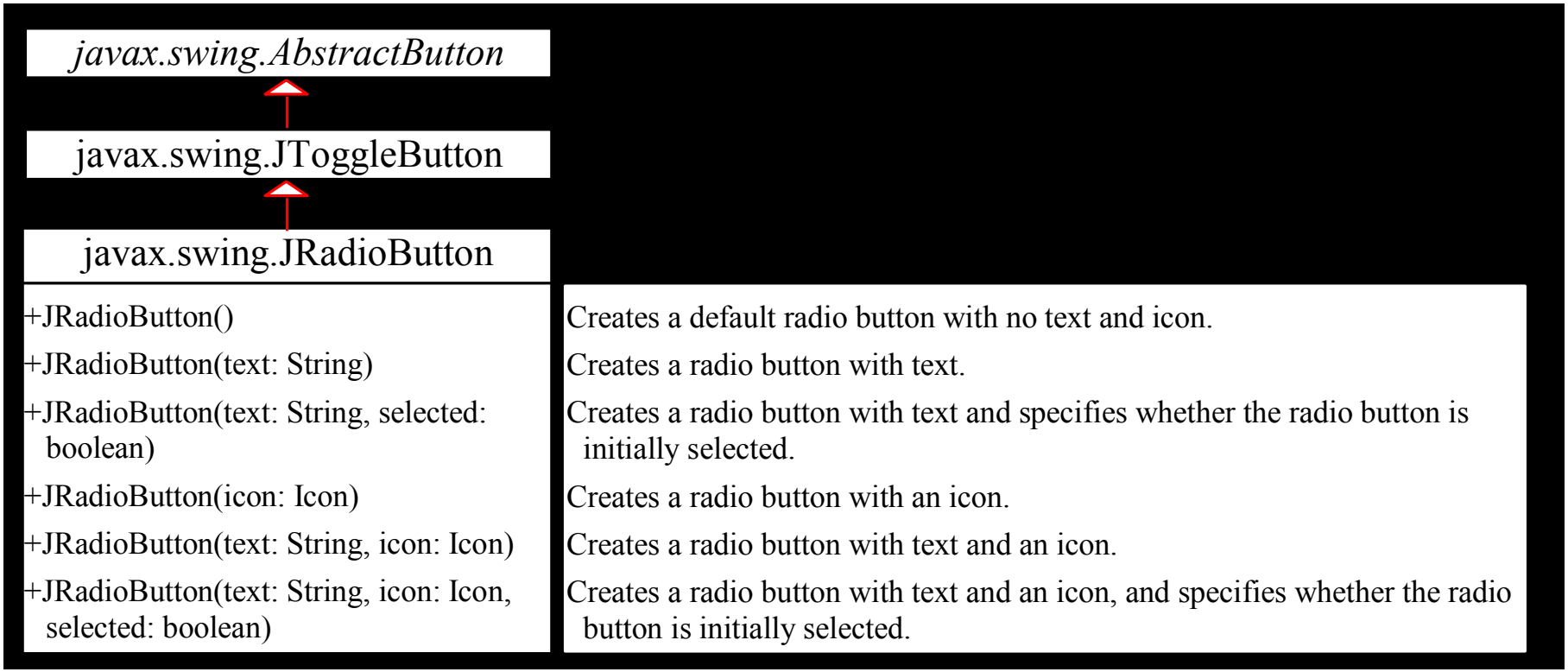
```
ButtonGroup btg = new ButtonGroup();
btg.add(jrb1);
btg.add(jrb2);
```

Quote Options Application



JRadioButton

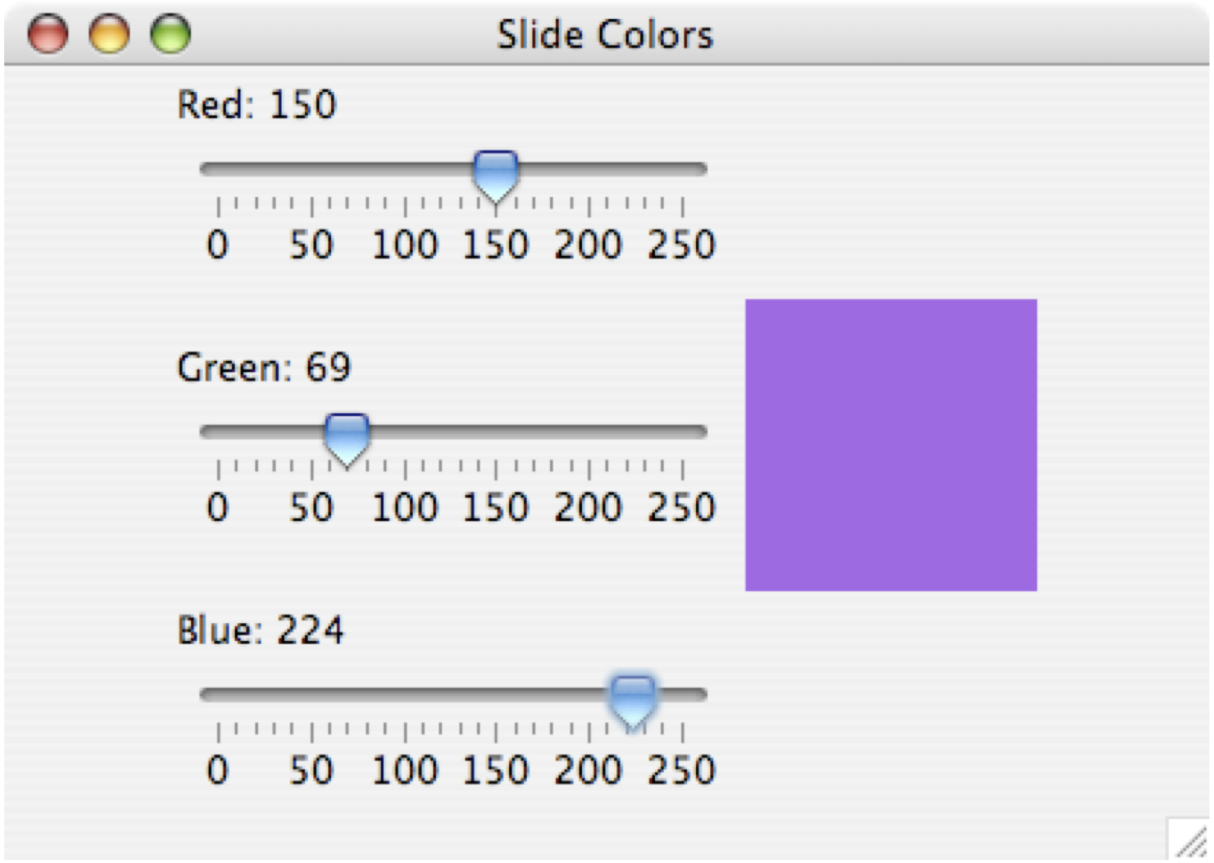
Radio buttons are variations of check boxes. They are often used in the group, where only one button is checked at a time.



Sliders

- *Sliders* allow the user to specify a numeric value within a bounded range
- A slider can be presented either vertically or horizontally
- Optional features include
  - tick marks on the slider
  - labels indicating the range of values
- A slider produces a *change event*, indicating that the position of the slider and the value it represents has changed
- A slider is defined by the `JSlider` class

Slide Colors Application



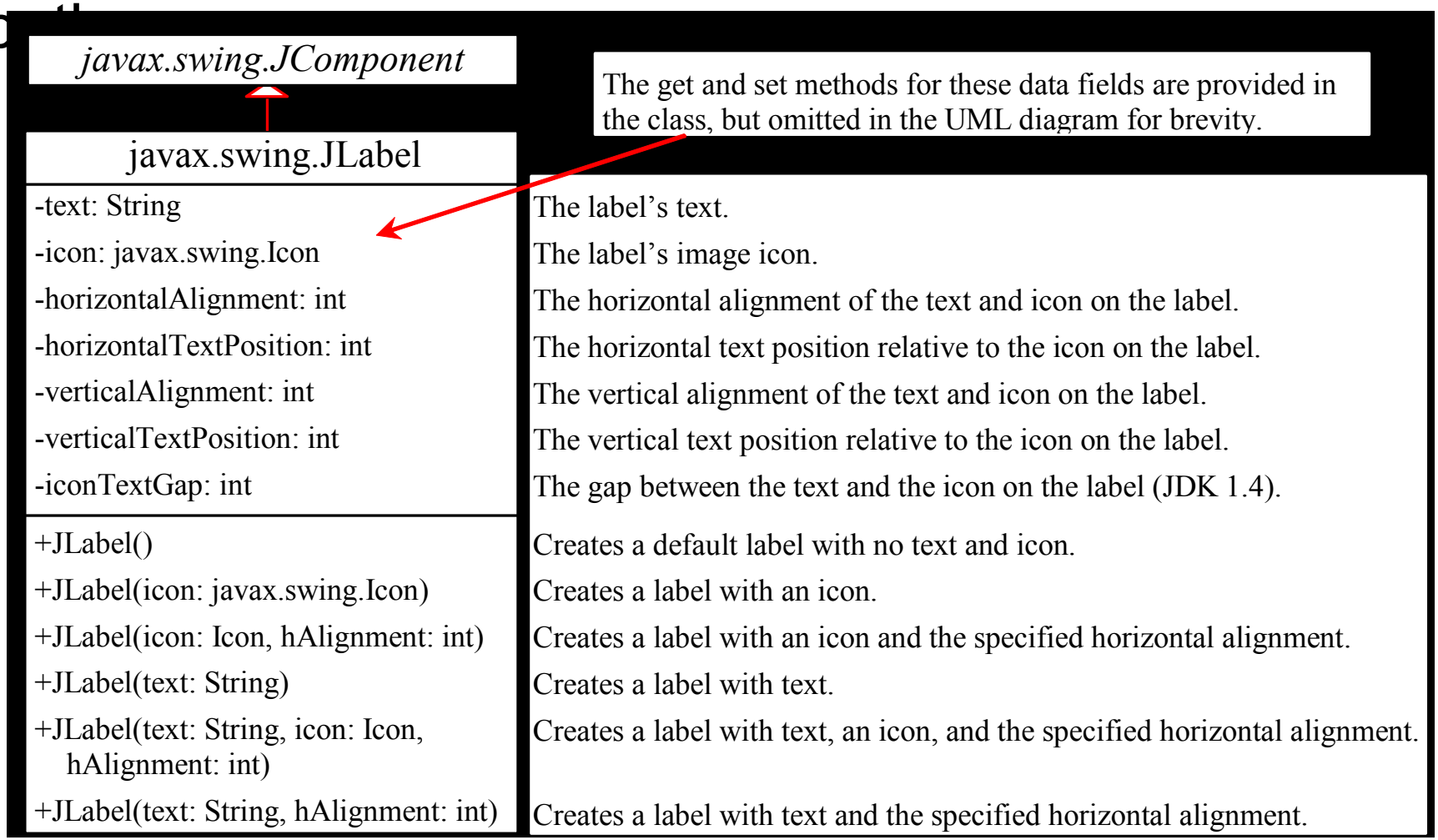
Combo Boxes

- A *combo box* allows a user to select one of several options from a “drop down” menu
- When the user presses a combo box using a mouse, a list of options is displayed from which the user can choose
- A combo box is defined by the `JComboBox` class
- Combo boxes generate an action event whenever the user makes a selection from it

Juke Box Application

JLabel

A *label* is a display area for a short text, an image, or b



JLabel Constructors

The constructors for labels are as follows:

```
JLabel()

JLabel(String text, int horizontalAlignment)

JLabel(String text)

JLabel(Icon icon)

JLabel(Icon icon, int horizontalAlignment)

JLabel(String text, Icon icon, int
horizontalAlignment)
```

`JTextField` Constructors

- `JTextField(int columns)`

  Creates an empty text field with the specified number of columns.

- `JTextField(String text)`

  Creates a text field initialized with the specified text.

- `JTextField(String text, int columns)`

  Creates a text field initialized with the specified text and the column size.

`JTextField` Properties

- `text`
- `horizontalAlignment`
- `editable`
- `columns`

`JTextField` Methods

- `getText()`

  Returns the string from the text field.

- `setText(String text)`

  Puts the given string in the text field.

- `setEditable(boolean editable)`

  Enables or disables the text field to be edited. By default, `editable` is `true`.

- `setColumns(int)`

  Sets the number of columns in this text field.
  The length of the text field is changeable.