Java Tutorial 1999

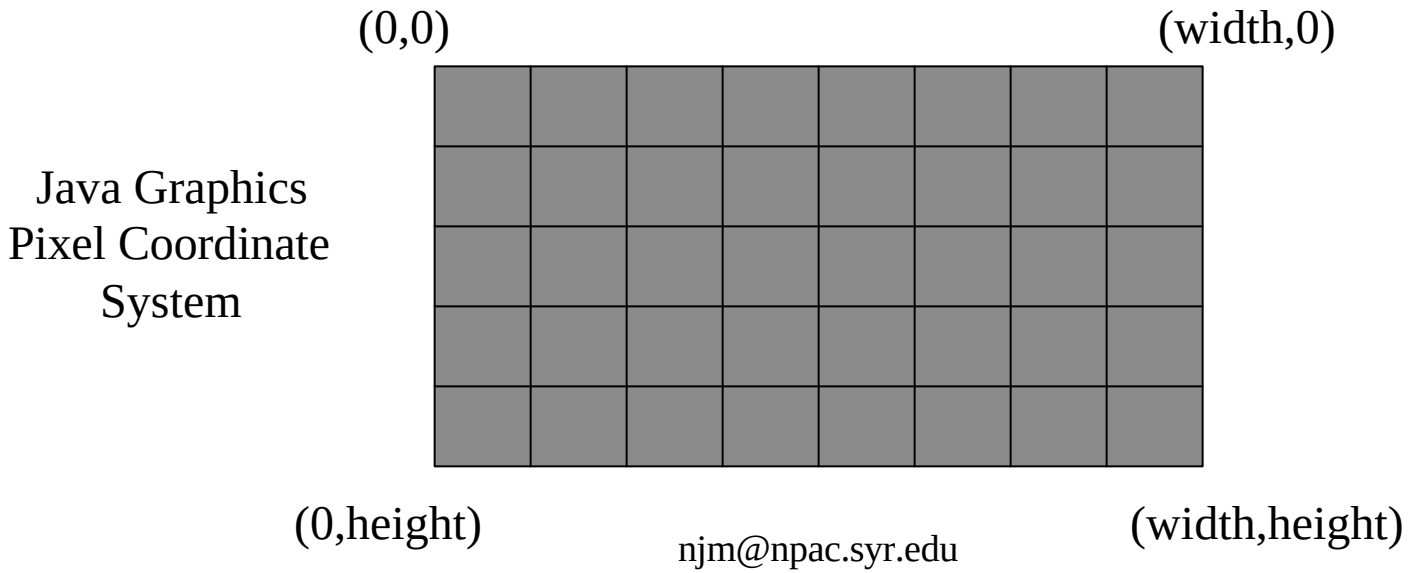
Part 3: Graphics, the Abstract Windowing Toolkit, and the Swing Set

Instructors: Geoffrey Fox , Nancy McCracken
Syracuse University
111 College Place
Syracuse
New York 13244-4100

Applets and Graphics

The java.awt.Graphics class

- The AWT has two different schemes for creating all or parts of applet windows:
  - A high-level scheme where a Layout manager decides how to place components like buttons and textfields in the window (described later).
  - A low-level scheme where methods draw graphics objects to the window and must say where to place them in terms of pixels.



Methods for drawing in the Graphics Class

- Graphics class has methods to construct basic two dimensional images
  - (all parameters are ints unless otherwise indicated)
  - x, y pixel locations refer to the upper left corner of the object to be drawn (except for strings)
- drawString (String text, x, y)
- drawLine (x1, y1, x2, y2)
- drawRect (x, y, width, height), fillRect (...),
- drawOval (...), fillOval (...),
- drawRoundRect (x, y, width, height, arcWidth, arcHeight)
  - for a rectangle with rounded corners!,
- draw3DRect (x, y, width, height, Boolean b)
  - (to get shadow effect as in buttons - if b is true, it's raised),
- drawArc (x, y, width, height, startAngle, arcAngle), fillArc(..),
- drawPolygon(int xPoints[], int yPoints[], n);

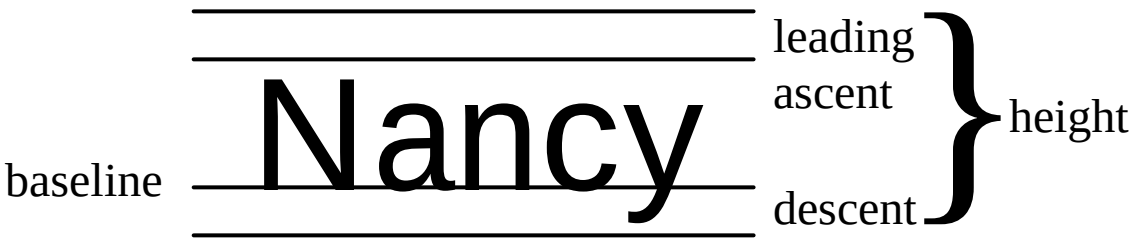
The java.awt.Font and FontMetrics Class

- Graphicsinstance.setFont(particularFont) will set the current Font in the instance Graphicsinstance of the Graphics class to the value particularFont of class Font.
  - The default call setFont(particularFont) sets the Font in the current applet.
  - setFont can also be used on instances of AWT components, as we shall see in later examples.
- The class Font has an important constructor used as in
- Font MyFont = new Font(”Serif", Font.PLAIN ,36);
  - where one can use specific font names TimesRoman, Courier, Helvetica, ZapfDingbats, but Sun would prefer you use the generic names Serif, SansSerif, and MonoSpaced. Additional font names include Dialog (the default for many AWT components).
  - Font.PLAIN, Font.BOLD, Font.ITALIC are possible text styles
- In an applet, the font specified may not be available in a particular browser. To find out which fonts are available, one may call
  - String fonts[] = Toolkit.getDefaultToolkit(). getFontList();

FontMetrics class

- FontMetrics fm = getFontMetrics(particularFont); / / allows one to find out about the font
  - fm.stringWidth("text"); / / returns pixel width of string "text"
  - fm.getHeight(); / / returns total height of highest Font character
  - getAscent(),
  - getDescent(),
  - getLeading(),

- Drawstring uses leftmost baseline point as (x,y)

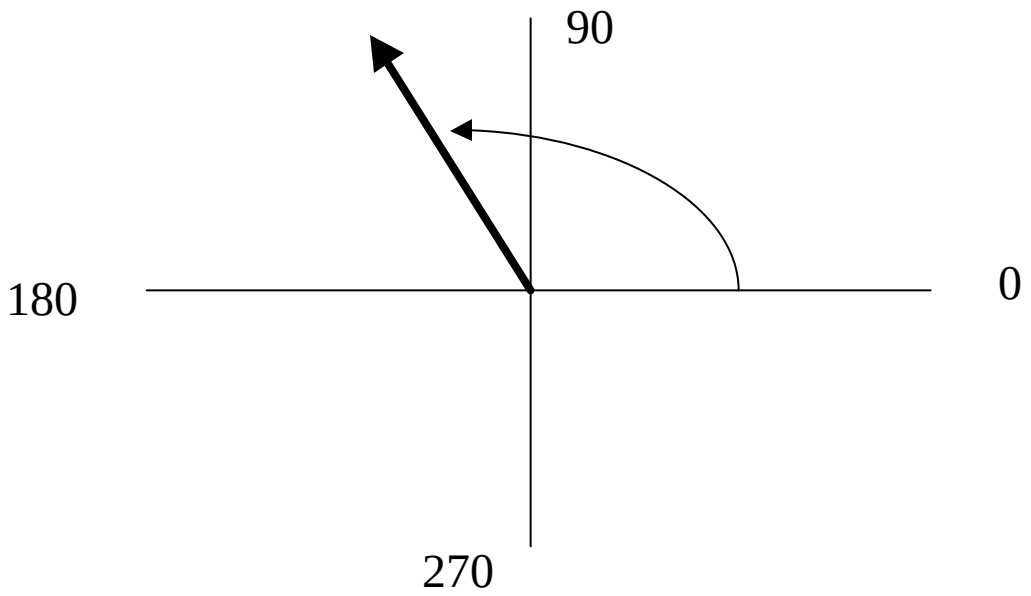


The java.awt.Color Classes

- Color c = new Color (redvalue, greenvalue, bluevalue); red/ green/ bluevalue can be specified as integers in 0 ... 255 or floating point numbers from 0 to 1.
- c is generated as a Color in RGB format.
- graphicsobject.setColor(c); / / sets current color in graphicsobject which is used for all subsequent operations
- There are particular Color instances already defined such as
  - Color.white equivalent to Color(255,255,255)
  - Color.black as equivalent to Color(0,0,0)
  - Color.pink as equivalent to Color(255,175,175)
  - also orange, cyan, magenta, yellow, gray, lightGray,
  - darkGray, red, green, blue

Graphics Examples

- Check out more examples
  - Color boxes generates colors using a random number generator.
  - Another example shows more graphics drawing. Note that drawing arcs is done with respect to the following angle representation:
  - drawArc (x, y, width, height, startAngle, arcAngle)



The Graphics2D class

- Additional graphics 2D drawing capabilities are found in the java.awt package added in JDK1.2. (This is not, apparently, part of the Swing set.)
- The Graphics2D class is a subclass of the Graphics class, so converting the graphics context to this class enables the additional methods to be called.

```
. . .
public void paint ( Graphics g )
    { Graphics2D g2d = ( Graphics2D ) g;   . . .   }
```

- Note that there are also packages for 3D graphics and other media frameworks not covered in this talk.

The Paint interface for filling objects

- With Graphics2D, the current graphics context includes the current “paint”:
  g2d.setPaint (p);
  where p is anything that implements the interface Paint.
- The Color class implements Paint.
- New classes include GradientPaint, SystemColor, and TexturePaint.
  - new GradientPaint (x1, y1, color1, x2, y2, color2, boolval)
    colors a gradient which starts at the first coordinate and shades in the direction of the section coordinate, shading from the first color to the second color. The boolean value determines whether the shading is cyclic (true) or acyclic (false).
  - new TexturePaint (image, rectangle)
    where the image is cropped to the size of the rectangle arg and used to tile the space.

The Stroke interface for borders and lines

- With Graphics2D, the current graphics context includes the current “stroke”:

  g2d.setStroke (s);

  where s is anything that implements the interface Stroke.
- Class BasicStroke provides a variety of constructors to specify the width of the line, how the line ends (the end caps), how lines join together (join lines), and the dash attributes of the line.
  - new BasicStroke (width, endcaps, joins, mitrelimit, dasharray, dashphase)
    where dasharray gives the lengths of a sequence of dashes and dashphase gives which dash to start with.

Shapes

- The new package java.awt.geom.*, also new in JDK1.2, includes a set of new classes that implement the interface Shape.
  - Ellipse2D.Double, Rectangle2D.Double, Arc2D.Double, RoundRectangle2D.Double, and Line2D.Double
  - There are also versions of each with .Float that takes float args.
- There is also a new class General Path for constructing shapes.
  - Create an empty path: gp = new GeneralPath ( );
  - Move to a position: gp.moveTo (x0, y0);
  - Extend the path in several ways:
    - » Straight lines: gp.lineTo (x1, y1);
    - » Bezier curves: gp.curveTo (x1, y1, x2, y2, x3, y3);
    - » Quadratic curves: gp.quadTo (x1, y1, x2, y2);
  - End the path: gp.closePath ( );

Graphics2D drawing methods

- Since Graphics2D is a subclass of Graphics, all the previous drawing methods are still there:
  g2d.drawRect (. . .), g2d.fillRect (. . .), etc.
- The new methods for drawing shapes are:
  g2d.draw (shape);
  which uses the current Stroke for the boundary line.
  g2d.fill (shape);
  which uses the current Paint for the fill of the shape.
- There are also new methods for changing the origin of the drawing:
  g2d.translate (x0, y0);
  g2d.rotate (angleinradians);
  This enables each shape (on previous slide) to be defined with respect to its own (0, 0) origin.

Applet methods

- These applet methods are called automatically by the appletviewer or browser during an applet's execution.
  - init() - called once when an applet is loaded for execution.
  - start() - called once after init and again every time the user returns to the HTML page of the applet (or every time the browser restarts the page).
  - stop() - called every time the applet's execution is suspended, mainly whenever the user leaves the HTML page.
  - destroy() - called once whenever the applet is removed from memory, normally when the browser exits.

Graphics is Event-Driven: paint method

- In every applet or windows application, the windowing system creates an Image with a Graphics object to keep track of the state of the window.
- In order to draw or write text to the window, you must override the paint method:
  - public void paint(Graphics g)
- The Graphics object g has a current color and font that can be changed by methods
- The window system can be interrupted for various reasons - the user resized it or some other window was put on top of it and then removed - and it does not save a copy of the pixels. Instead it calls the paint method. So even if you only draw one window, paint can be called many times.

Changing Graphics: repaint method

- Most applets and windows applications want to change what is drawn on the screen over its lifetime. This can be a sequenced animation, response to user input or mouse events, and so on.
- Whenever you want to redraw the screen, call
  - public void repaint(); // note no arguments
- Repaint gets the graphics context g and creates a thread to call update(g), which calls your paint method. So all your drawing changes can also be put in paint.
- One draws a sequence of text and shapes to define the screen, where the position of the object in the screen is given by pixel coordinates. If one object overlaps another, the latest one drawn covers up the area of overlap.
  - The exception to this is XOR graphics, which may be used to temporarily highlight a particular color. This is an advanced technique as other colors will also be affected.

Introducing a Single Thread

(See later for in-depth discussion of thread use)

Introduction to Threads

- A thread is a single sequential flow of control within a process.
- If a process has more than one thread, then each thread executes concurrently.
- Any Java applet that has extensive execution or loops to repaint the window must run as a concurrent thread with the browser window.
- To make an applet with a thread:
  - Change your applet definition to add "implements Runnable", the interface for threads.
  - Include an instance variable for the thread of your applet.
  - Have a start() method that creates a thread and starts it running and a stop() method which stops it running by making it null.
  - Implement a run() method containing the body of your applet code.

Example showing the standard thread methods

- These applet start and stop methods can always be used to start and stop the thread.

```
import java.awt.*;
import java.util.Date;
public class DigitalClock extends java.applet.Applet
                                                    implements Runnable
{  Font theFont = new Font("TimesRoman", Font.BOLD, 24);
   Date theDate;
   Thread runner;

  public void start ( )
     {  runner = new Thread(this);
        runner.start ( );
     }
  public void stop ( )
     {
        runner = null; }
     }
```

Declare a thread

Example showing thread methods, continued

- The body of the applet is in the run method, in this case a loop to keep showing the date.

```
public void run ( )
  {  Thread thisThread = Thread.currentThread();
     // loop as long as thread is defined
     while runner == thisThread)
     {  theDate = new Date ( );
        repaint ( );
        try {  thisThread.sleep ( 1000);  }
           catch ( InterruptedExecution e) { }
      }
    }
public void paint ( Graphics g )
   {  g.setFont ( theFont );
      g.drawString ( theDate.toString ( ), 10, 50 );
    }
}
```

Wait 1 second

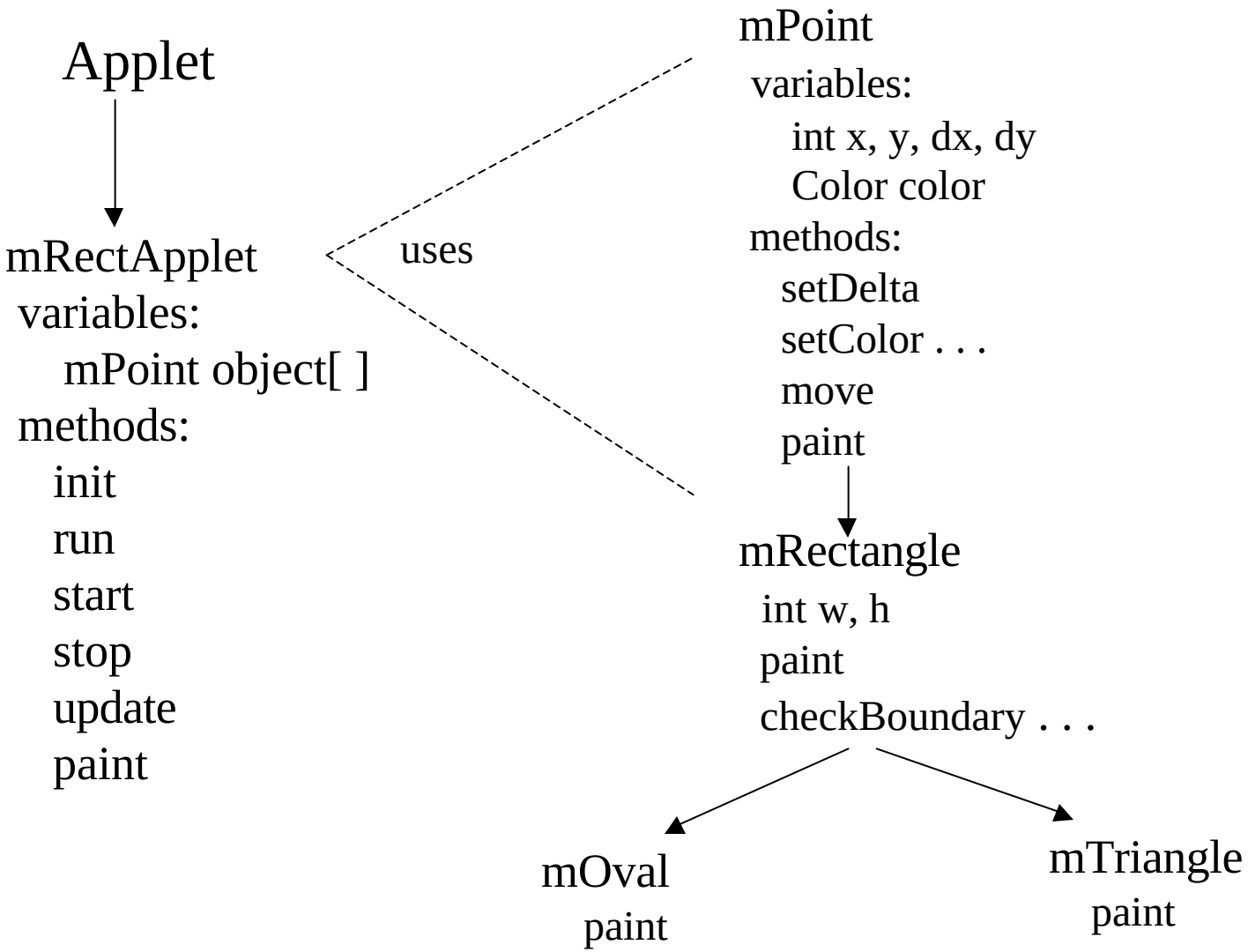
Do nothing for an exception

Motion (or Animation) Example using threads and parent/ child hierarchy

Designing the Classes

- We want to make applets that can have objects of different shape moving across the applet window.
  - –The first applet will have two rectangles.
  - –We will represent each rectangle as an instance of a class mRectangle.
  - –Then the applet will have a thread that loops, and each time around the loop, will draw the rectangles in a new position.
- We will design a hierarchy of classes to represent various shapes (including rectangles).
  - –We define a parent class for movable point objects. Each instance of this class is represented by an x,y location, by a dx,dy offset for the object to move, and a color for the object. This example also illustrates data encapsulation, where users of the class must use methods to get or set data of the class.
  - –A child class will represent rectangles by adding variables for width and height. It will also override the paint method to draw rectangles.
  - –Other shapes are easily constructed as children of the rectangle class by overriding the paint method to draw a new shape.

The Class Hierarchy of this example



Images and Double Buffering

Getting Images Downloaded

- The Applet class provides a method getImage, which retrieves an image from a web server and creates an instance of the Image class.
- Image img =
  - getImage(new URL("http:/ / www.tc.com/ image.gif"));
- Another form of getImage retrieves the image file relative to the directory of the HTML or the directory of the java code.
  - Image img = getImage(getDocumentBase(), "images/ image.gif");
  - Image img = getImage(getCodeBase(), "images/ image.gif");
- Note that the Image class in java.awt has many methods for images, such as getWidth(imageobserver) and getHeight(imageobserver), which return the width and height in pixels.
- Images can be created from either gifs or jpegs.

Drawing Images to the applet window

- The Graphics class provides a method drawImage to actually display the image on the browser screen.
  - void paint ( )
    - » { g.drawImage (img, 10, 10, this) ; }
  - where the top left corner of the image will be drawn at (x,y) position (10,10)
  - use “this” as the ImageObserver argument, it is the component in which the image is displayed.
- You can also scale the image to a particular width and height.
  - void paint ( )
    - » { g.drawImage (img, 10, 10, w, h, this) ; }

ImageIcon class in Java2

- Another class for using images in Java2 is ImageIcon, from javax.swing, which can again use both gifs and jpegs.

  ImageIcon icon = new ImageIcon (picture.gif);

- In addition to creating an ImageIcon from a file, there are constructors to create an ImageIcon from a URL or another Image.
- ImageIcons are used in other swing components, but they can also be painted to any component, such as an applet:

  icon.paintIcon (this, g, x, y);

  where this is the component in which it will be painted, g is the graphics context, and x and y are the coordinates of the upper left-hand corner.

Image Downloading -- imageObserver, MediaTracker

- When drawImage is called, it draws only the pixels of the image that are already available.
- Then it creates a thread for the imageObserver. Whenever more of the image becomes available, it activates the method imageUpdate, which in turn call paint and drawImage, so that more of the image will show on the screen.
- The default imageUpdate doesn't work if you are double buffering the window in which the image appears.
- More control over showing the image as it downloads can be obtained by working with the MediaTracker class, using methods that tell you when the image has fully arrived.
  - Another method is prepareImage(MyImage, this);
    - » which returns a boolean that is true when image is fully downloaded.

An Image Drawing Example

- This example shows how to use the getWidth and getHeight methods of the Image class to use in scaling the image under java program control.

```
import java.awt.*
public void class Drawleaf extends java.applet.Applet
{  Image leafimg;

   public void init ( )
   {  leafimg = getImage(getCodeBase( ),"images/Leaf.gif");
    }
   public void paint ( Graphics g )
   {  int w = leafimg.getWidth(this);
     int h = leafimg.getHeight(this);
      g.drawImage ( leafimg ,10, 10, w/4, h/4, this);
    }
}
```

Flickering in Applets and its Solution

- Unless you are careful, dynamic applets will give flickering screens (in the regular AWT, not in Swing set).
- This is due to the cycle
  - repaint()
  - update(g) clearing screen
  - paint(g) drawing new screen .....
  - where flicker is caused by the rapid clear-paint cycle.
- There are two ways to solve this problem which involve changing update() in different ways
  - 1: Change update() either not to clear screen at all (because you know paint() will write over parts that are to be changed) or to just clear the parts of the screen that are changed
  - or 2:Double Buffering

The default Update(Graphics g) Method

- This sets background color and initializes applet bounding rectangle to this color
  - public void update(Graphics g)
    - {
    - g.setColor(getBackground());
    - g.fillRect(0,0,getSize().width,getSize().height);
    - g.setColor(getForeground());
    - paint(g);
    - }
  - getBackground() and getForeground() are methods in component class
  - fillRect() is a method in Graphics class

Double Buffering to Reduce Flicker - I

- Here you have two "graphics contexts" (frame buffers of the size of the applet), and you construct the next image for an animation "off-line" in the second frame buffer.
- This frame buffer is then directly copied to the main applet Graphics object without clearing image as in default update()
- In init(), you would create the frame buffer:
  - Image OffscreenImage; / / Place to hold Image
  - Graphics offscreenGraphics; / * The second graphics context of offscreenImage */
  - offscreenImage = createImage(getSize().width,getSize().height);
    - » offscreenGraphics = offscreenImage.getGraphics();

Double Buffering to Reduce Flicker - II

- In paint(), one will construct applet image in offscreenGraphics as opposed to the argument g of paint(). So one would see statements such as:
  - offscreenGraphics.drawRect(x, y, w, h);
- Finally at end of paint(), one could transfer the off-screen image to g by
  - g.drawImage(offscreenImage,0,0,this);
- One would also need to override the update() method by
  - public void update(Graphics g)
  - { paint(g);
  - }

Double Buffering

- The DigitalClock doesn't flicker, but this illustrates the technique on a short example.

```
public void class DigitalClock extends java.applet.Applet
                                                    implements Runnable
{  . . .
   Image offscreenImg;
   Graphics og;
   public void init ( )
   {  offscreenImg = createImage (getSize( ).width, getSize( ).height);
      og = offscreenImg.getGraphics ( );
    }
   public void paint ( Graphics g )
   {  og.setFont ( theFont );
      og.drawString ( theDate.toString ( ), 10, 50 );
      g.drawImage ( offscreenImg, 0, 0, this);
    }
   public void update ( Graphics g)
   {  paint ( g);  }
}
```

Abstract Windowing Toolkit (AWT):

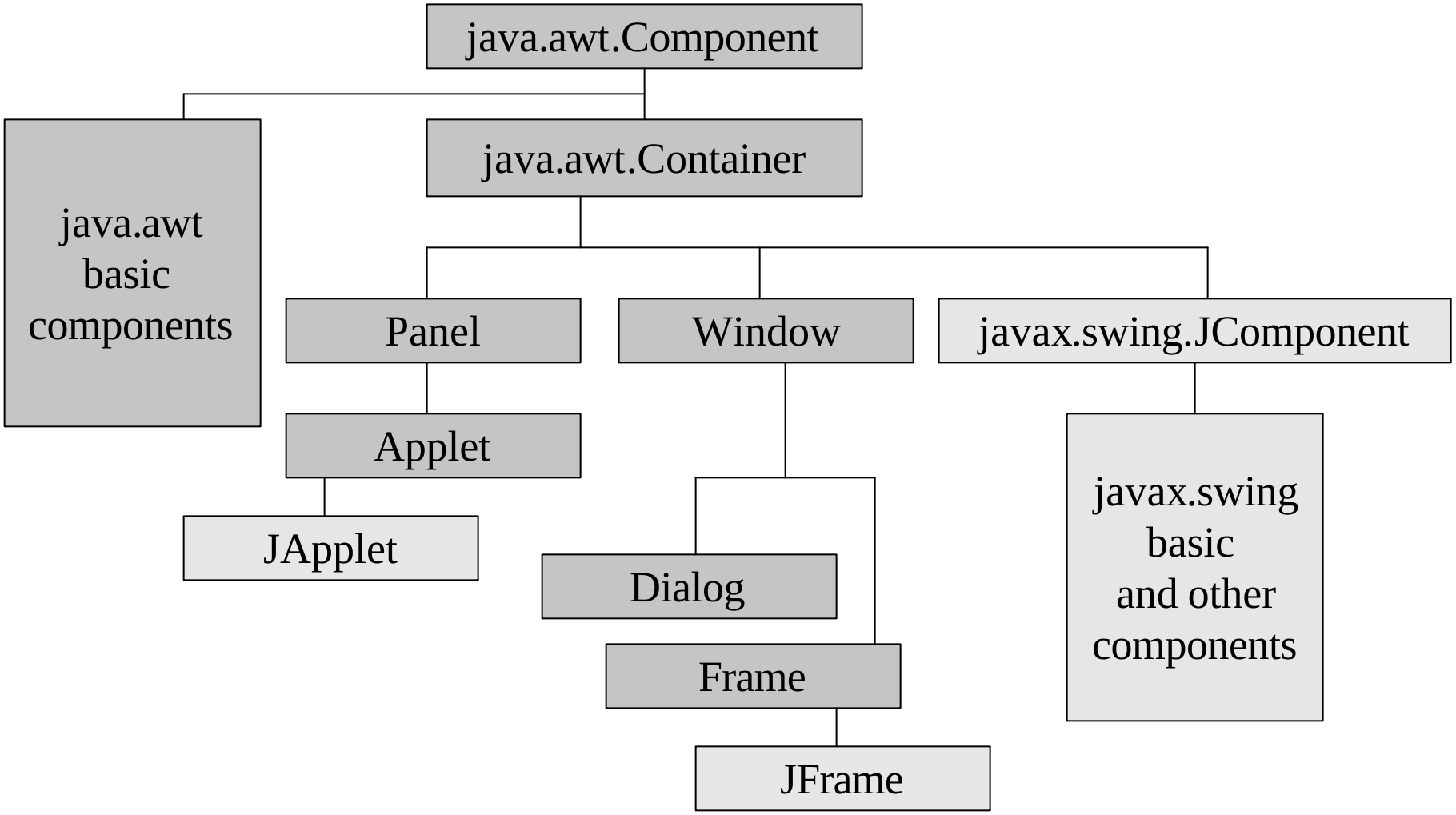
Components such as buttons, textfields, etc.

Java 1.1 Event Model

AWT GUI Components

- In Java, the GUI (Graphical User Interface) is built hierarchically in terms of Components -- one Component nested inside another starting with the smallest Buttons, including Menus, TextFields etc. and ending with full Window divided into Frames, MenuBars etc.
- The placement of components in the window is controlled in a fairly high-level way by one of several Layout Managers.
- The user can interact with the GUI on many of its components, by clicking a button, typing in text, etc. These actions cause an Event to be generated, which will be reported by the system to a class which is an Event Listener, and which will have an event handler method for that event. This method will provide the appropriate response to the user's action.

Top Levels of the Component Hierarchy



AWT vs. Swing Components

- The AWT components are designed to take on the “look and feel” of the underlying window system where they are displayed. For applets, this is wherever the browser is running.
  - AWT components have peer classes in which they have a specific window implementations.
- Swing components are designed to have a fixed “look and feel” on all platforms.
  - They are sometimes called lightweight because they are implemented in Java itself. The only peer implementation required is to put up a window and paint it. Thus, swing components are far less prone to windowing system dependency bugs.
- The default “platform look and feel”, abbreviated plaf, is Metal. Others are available such as Motif and Windows.

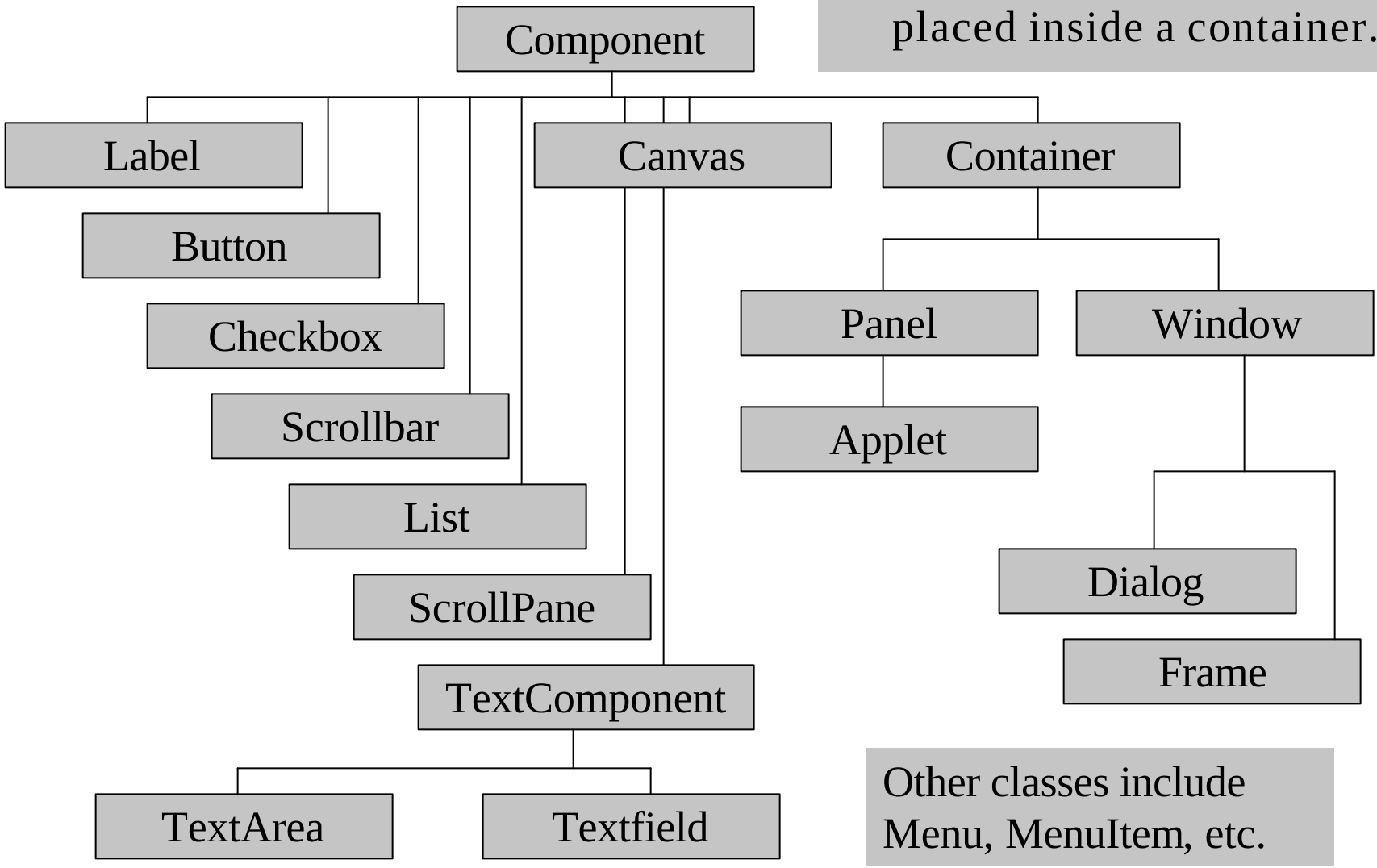
  For example: UIManager.setLookAndFeel (plaf);

AWT Components

- We first describe the basic AWT components and the event handling model.
- We next describe Swing components.
  - The event handling model is the same.
  - Each AWT component, such as Button, has a corresponding Swing component, called JButton.
  - The Swing component typically may have additional functionality. For example, a Button's appearance may have a text label and colors. In addition, a JButton may have an Icon on it.
  - Swing components have a more complex implementation - they are essentially wrapper classes for a set of classes giving a "model - view - controller" design pattern:
    - » model gives contents - such as state or text
    - » view gives visual appearance
    - » controller gives behavior - such as reaction to events

Picture of the AWT Component Class and some of its inheritance

- Other components can be placed inside a container.



Other classes include Menu, MenuItem, etc.

Basic AWT Components

- For each basic component, one can create one or more instances of the component type and then use one of the "add" methods to place it into a Container such as an applet window.
- For now, we assume that components are added to the window in order from left to right and top to bottom as they fit. (This is actually the default FlowLayout Manager).
- For each component, there will be methods:
  - some affect the properties or appearance, such as setBackground or setFont, which are inherited from the Component class.
  - others may dynamically obtain or change information about the component, such as getText for TextFields, which may return whatever String the user has typed into the TextField.

Basic AWT Component: Label

- This is an area where text can be displayed in the window.
- Create an instance of the Label class and add it to the window:
  - Label label1 = new Label ("aligned left");
  - add (label1);
- Another constructor allows a second argument which is an alignment: Label.LEFT, Label.CENTER, or Label.RIGHT
  - Label label2 = new Label ("aligned right", Label.RIGHT);
- Method setText allows you to change the String in the Label, and getText() returns the current String
  - label2.setText("another message");

Basic AWT Component: Button

- A Button is the familiar way to allow a user to cause an event by clicking with the mouse and is created with a String to label it
  - Button button1 = new Button("Click here");
  - add (button1);
- AWT Buttons are normally created to appear in the style of the user's windowing system, except that you can control the color of the button and the String
  - button1.setBackground (Color.cyan);
  - button1.setForeground (Color.black);

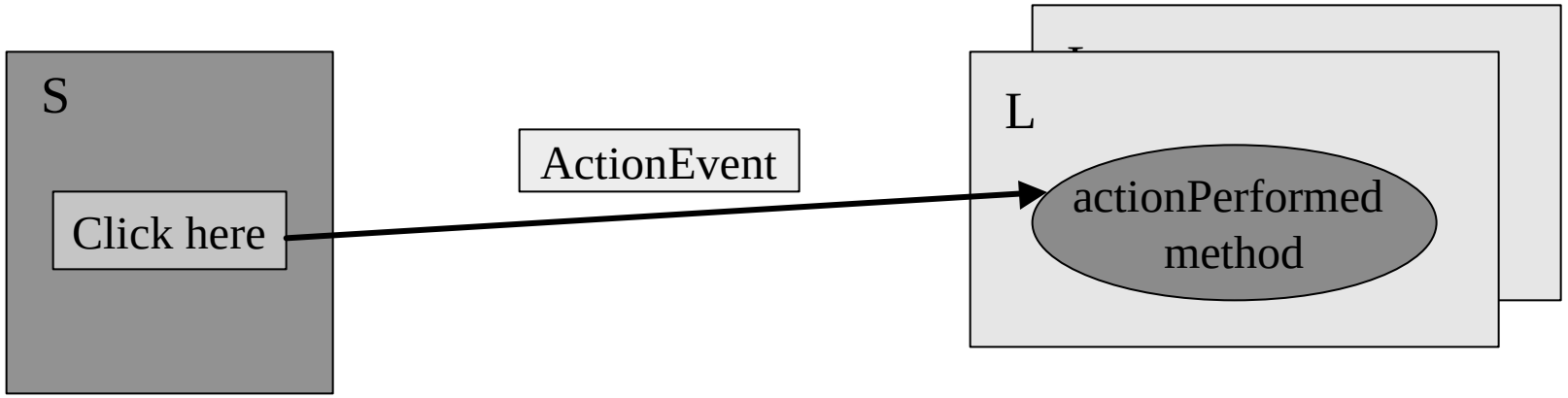
The AWT Event Model

- An Event Listener is an instance of any class that wants to receive events.
- An event source is an object that generates events. An event source will keep a list of event listeners who want to be notified for particular events. This is sometimes called event delegation.
- The event source notifies event listeners by invoking a particular method of the event listener (aka the event handler method or event procedure) and passing it an Event object, which has all the information about the event.
- For example, a component with a button is an event source, which generates an event called ActionEvent. There must be a class which implements an interface called ActionListener and which is on the list of listeners for that button. Then the Java system will provide the mechanism that passes the ActionEvent to a standard method of the ActionListener interface, namely a method called actionPerformed(). This method will receive the event object and carry out the response to the event.

Event Model illustrated with Button

Window with event source - a Button. The button puts L on its ActionListener list.

Instance of class implementing ActionListener



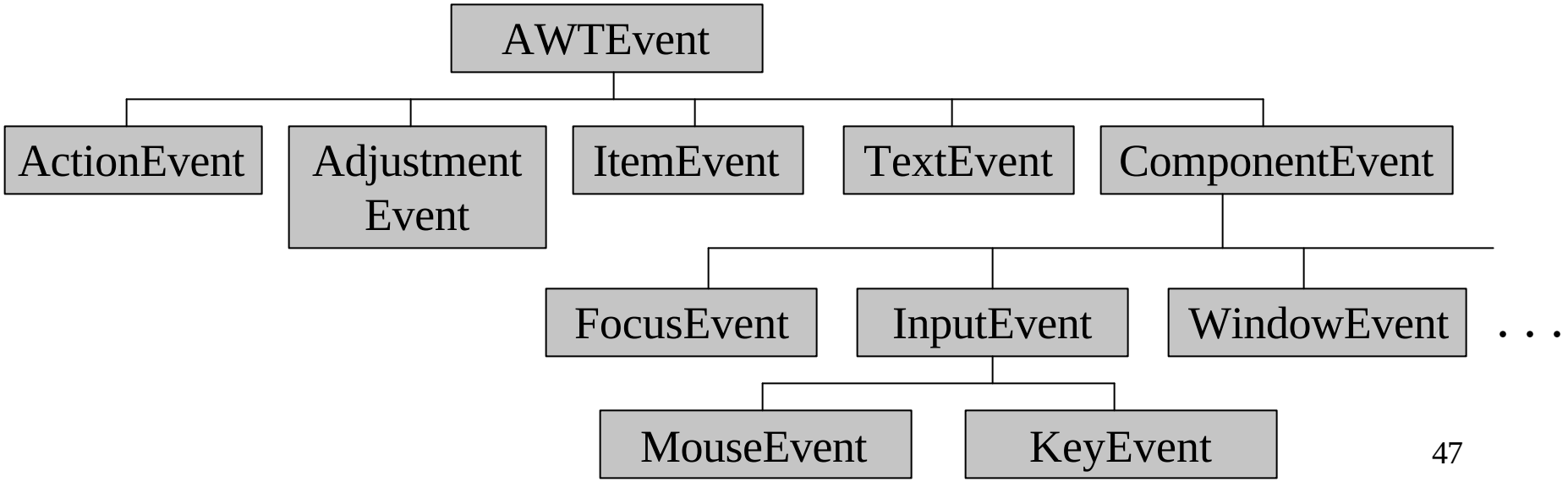
When user clicks the button, the button makes an ActionEvent object and passes it to the actionPerformed method of listeners on its list.

Setting up ActionEvents for a Button

- When the button is created, it should have at least one listener class added to its list of listeners:
  - button1.addActionListener (eventclass);
- where eventclass is an instance of the listener class.
  - every component which can cause an event called X, has methods addXListener and removeXListener.
- Then this class must implement the interface ActionListener. This interface requires only one event handler method:
  - public class EventClass implements ActionListener
  - {
  - public void actionPerformed (ActionEvent e) {... }
  - }
- If the event source class is acting as its own listener, then you just say
  - button1.addActionListener (this);

The Event Classes

- Every event has a source object, obtained by getSource(), and a type value, obtained by getID(). In the case of buttons, the ID is ACTION_PERFORMED. Other Events may have more than one type of event ID.
- Event subclasses also have methods for whatever data is needed to handle the event. For example, ActionEvent has a method getActionCommand, which for buttons, returns the string labelling the button. MouseEvent has methods getX() and getY(), which return the x and y pixel location of the mouse, and getClickCount().

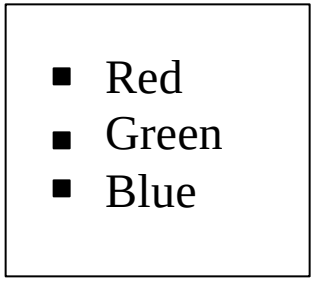


AWT Components -- Text Fields & Areas

- To add a text field for display or input one line of text (in this case, 30 characters wide):
  - TextField tf = new TextField("initial text", 30);
  - add(tf);
- The text which is displayed can be changed:
  - tf.setText("now show a new text");
- If the user types input into the text field, it can be obtained:
  - stringvar = tf.getText();
- Or you can disallow the user to type:
  - tf.setEditable(false);
- The TextArea class also has these methods, but it can display multiple lines.
- When the user types in text and presses "return" or "enter", an ActionEvent is generated, so, similarly to Buttons, an ActionListener must be provided. TextAreas generate TextEvents.

AWT Components -- Checkbox

- Checkboxes are on-off toggles implemented as
  - Checkbox red = new Checkbox("Red");
  - Checkbox green = new Checkbox("Green");
  - Checkbox blue = new Checkbox("Blue",null, true);
  - add(red); add(green); add(blue);



- The first two are initially set to "false" as the optional third argument is not given. The last one is initially set to "true".
- The state of a checkbox, i.e. whether it is checked, is given by a boolean result of the method, getState:
  - if (red.getState()) . . .;
- If a user clicks a Checkbox, an ItemEvent is generated. The listener must implement ItemListener with the one method itemStateChanged (ItemEvent e).

Some Further AWT Components -- typical subunits of panels

- Choice is a class that gives a menu where you choose from various items, also sometimes called a drop-down list. Selecting an element of the menu generates an ItemEvent.
- List is another child of Component that is similar in use to Choice but gives a fixed size list which can be scrolled and where you can select one or more entries. Lists can generate both ItemEvents if the user selects (clicks once) an item, and ActionEvents if the user double-clicks an item.
- Scrollbar is a class that defines a horizontal or vertical scrollbar. Note this is distinct from scrollbars that come with TextArea and List. It generates AdjustmentEvents. The AdjustmentListener must have a method adjustmentValueChanged.

Keyboard and Mouse Events

- More generally, any component, including containers, can generate mouse and keyboard events as the user moves or clicks the mouse in the window, or types a single key on the keyboard.
- This is quite often used in a Canvas or graphics drawing area.
- The previous events (ActionEvent, ItemEvent, AdjustmentEvent and TextEvent) are called semantic events as they express what the user is doing on a component. The remaining ones, such as KeyEvents, MouseEvents, FocusEvents and Window Events, are called low-level events.

Key Events

- Typing a single key generates KeyEvents. These events must be handled by implementing the KeyListener interface. It has three methods corresponding to the three actions that can occur on a key:
  - public void keyPressed (KeyEvent e)
  - public void keyReleased (KeyEvent e)
    - » these methods are called when a key is pressed down and released up, respectively, and report a virtual key code, which is an int encoding of the keys, such as VK_SHIFT, VK_A, . . .
  - public void keyTyped (KeyEvent e)
    - » this reports the character on the key that was pressed
- Typically, one uses methods on the key event to find the name of the key or key code:
  - String s = e.getKeyChar ( );
  - String t = e.getKeyText (e.getKeyCode());

Mouse Events

- There are seven different MouseEvents, handled by methods in both the MouseListener and the MouseMotionListener interfaces.
  - MouseListener:
    - » public void mousePressed (MouseEvent e)
      - called when the mouse button is pressed with the cursor in this component
    - » public void mouseClicked (MouseEvent e)
      - called when the mouse button is pressed and released without moving the mouse
    - » public void mouseReleased (MouseEvent e)
      - called when the mouse button is let up after dragging

Additional Mouse Event handler methods

- More MouseListener methods
  - » public void mouseEntered (MouseEvent e)
    - ◆ called when the mouse cursor enters the bounds of the component
  - » public void mouseExited (MouseEvent e)
    - ◆ called when the mouse cursor leaves the component
- MouseMotionListener
  - » public void mouseDragged (MouseEvent e)
    - ◆ called when the mouse is moved while the button is held down
  - » public void mouseMoved (MouseEvent e)
    - ◆ called when the mouse cursor moves

Methods for Mouse Events

- All mouse events report the x and y location of the mouse in pixels:
  - event.getX ( ) ;
  - event.getY ( ) ;
- You can also obtain the click count for double or event triple clicks:
  - event.getClickCount ( ) ;
- You can distinguish between different mouse buttons:
  - (event.getModifiers ( ) & InputEvent.BUTTON3_MASK) != 0 tests for a right click of the mouse
- Note that one response to mouse motion can be to change the appearance of the cursor. There are 14 different built-in cursors as well as a Toolkit method to define your own.
  - if (b) setCursor (Cursor.getDefaultCursor ( ))
    else setCursor (Cursor.getPredefinedCursor (Cursor.HAND_CURSOR));

Using Mouse Events for User Interaction

- We set up a test program that creates three movable objects, a rectangle, circle and triangle, as in the earlier example. In this program, we start with them all cyan. Whenever the mouse is detected to be over one of the objects, its color is changed to red. If the mouse button is used to drag the object, we move the object to the mouse location.
- Note that it is not necessary to introduce a thread for this applet since it is not running continuously - it is mostly waiting for mouse events.

Adapter Classes

- For every Event Listener interface with more than one method, there is a corresponding Event Adapter class. For example, there is an MouseAdapter class to go with the MouseListener interface.
- The adapter class implements its corresponding listener class by providing all of the required methods, but which have bodies that do nothing.
- For interfaces like MouseListener and MouseMotionListener, this can be handy because there are several methods in each interface. Typically, you don't want to implement all of the methods. So it is more convenient to make a class which extends the adapter class than to directly implement the listener class.
    - » class MouseHandler extends MouseAdapter
    - » { ... / / override only the methods that you want to implement
    - » public void mousePressed(MouseEvent e) { . . . }
    - » }

Separating GUI and Application Code

- In large applications, some Java experts recommend that it is a better design to separate the responsibilities of getting user input and executing commands because it is common to have multiple ways to activate a command.
  - Make an object for every command
  - Each command object is a listener for the events that trigger it
- The Swing package provides the Action interface to encapsulate commands and attach them to multiple event sources. The Action interface implements the ActionListener interface and has additional methods for properties of the command:
  - void actionPerformed (ActionEvent e)
  - void setEnabled (boolean b), boolean isEnabled ( )
  - void putValue (String key, Object val), Object getValue (String key)
  - addPropertyChangeListener, removePropertyChangeListener

Abstract Windowing Toolkit (AWT): Layouts

Layout of Components in a Panel

- The various panels in a container are laid out separately in terms of their subcomponents
- One can lay components out "by hand" with positioning in pixel space
- However this is very difficult to make machine independent. Thus one tends to use general strategies which are embodied in 5 LayoutMangers which all implement the LayoutManager Interface. One can expect further custom LayoutManager's to become available on the Web
- To create a layout, such as FlowLayout, in your panel:
  - setLayout(new FlowLayout());
  - This particular Layout is the default.

Brief Description of LayoutManagers

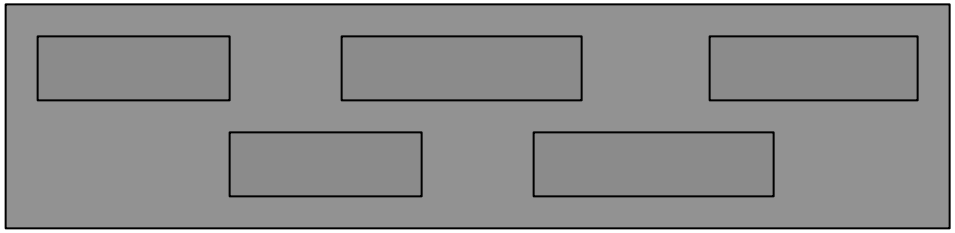
- FlowLayout is a one dimensional layout where components are "flowed" into panel in order they were defined. When a row is full up, it is wrapped onto next row
- BorderLayout arranges the components into five areas called North, South, East, West and Center.
- GridLayout is a two dimensional layout where you define a N by M set of cells and again the components are assigned sequentially to cells starting at top left hand corner -- one component is in each cell. The cells are the same size.
- In Java2, BoxLayout is a one dimensional layout where you have more control over the spacing and sizing of the components.
- CardLayout lays out in time not space and each card (Displayed at one time) can be laid out with one of spatial layout schemes above
- GridBagLayout uses a class GridBagConstraints to customize positioning of individual components in one or more cells

Description and Example of BorderLayout

- BorderLayout has five cells called North South East West Center and components are assigned to these cells with the add method. Unlike other add methods, here the order is not important:
  - » add(new TextField("Title",50), BorderLayout.NORTH);
  - » add(new TextField("Usually_status_message",50), BorderLayout.SOUTH);
- Remember this is default for a Frame Container
- The constructor "new BorderLayout()" can have no arguments or "new BorderLayout(hgap, vgap)" can specify numbers of pixels inbetween components.

FlowLayouts in detail

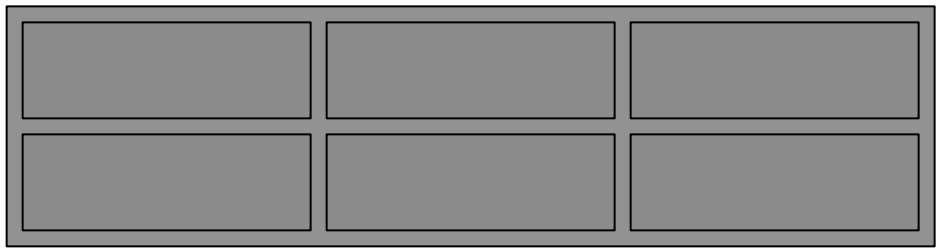
- This simple layout manager "flows" components into the window. The components can be aligned, and space between them specified by arguments hgap and vgap:
  - setLayout(new FlowLayout(FlowLayout.LEFT, 5, 2));
    setLayout(new FlowLayout(FlowLayout.CENTER));



  - setLayout(new FlowLayout(FlowLayout.RIGHT));
- The FlowLayout Manager's strategy includes making each component its default "preferred size".

GridLayouts

- The first two arguments of the GridLayout constructor specify the number of rows of cells (i.e. number of cells in the y direction) and the number of columns of cells (in the x direction)
  - setLayout(new GridLayout (2, 3));
- Additional arguments hgap and vgap specify the number of pixels inbetween the columns and rows:
  - setLayout(new GridLayout (2, 3, 10, 15));
- The GridLayout Manager's strategy is to make each cell exactly the same size so that rows and columns line up in a regular grid.



BoxLayouts in Java 2

- Like the other layout managers, you can setLayout of a panel to be a BoxLayout, but in addition, there is a special container called Box whose default layout is a vertical or horizontal box layout:

  Box b = Box.createHorizontalBox ( );

  or Box b = Box. createVericalBox ( );
- You can add components in the usual way and they are flowed into the one dimension, where the size strategy is to make them fit into one row or column, using the preferred size if possible (and alignment), but growing them to maximum size if necessary.
- There are invisible fillers available to space the components.

BoxLayout fillers in Java2

- There are three kinds of fillers:
  - A strut adds some space between components:

    ```
    b.add ( button1 );
    b.add ( Box.createHorizontalStrut ( 8 ));
    b.add ( button2 );
    ```

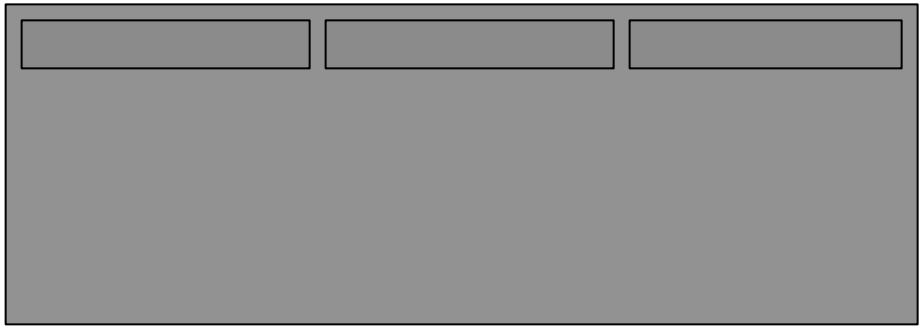
    adds 8 pixels between these buttons in a horizontal box.
  - A rigid area also adds a fixed amount of space between components, but may also specify a second dimension which may affect the height of a horizontal box and the width of a vertical box.

    ```
    b.add ( Box.createRigidArea ( new Dimension ( 10, 20 ));
    ```
  - Adding glue separates the components as much as possible.

    ```
    b.add ( button1 );
    b.add ( Box.createGlue ( ));
    b.add ( button2 );
    ```

Hierarchical Use of Layouts

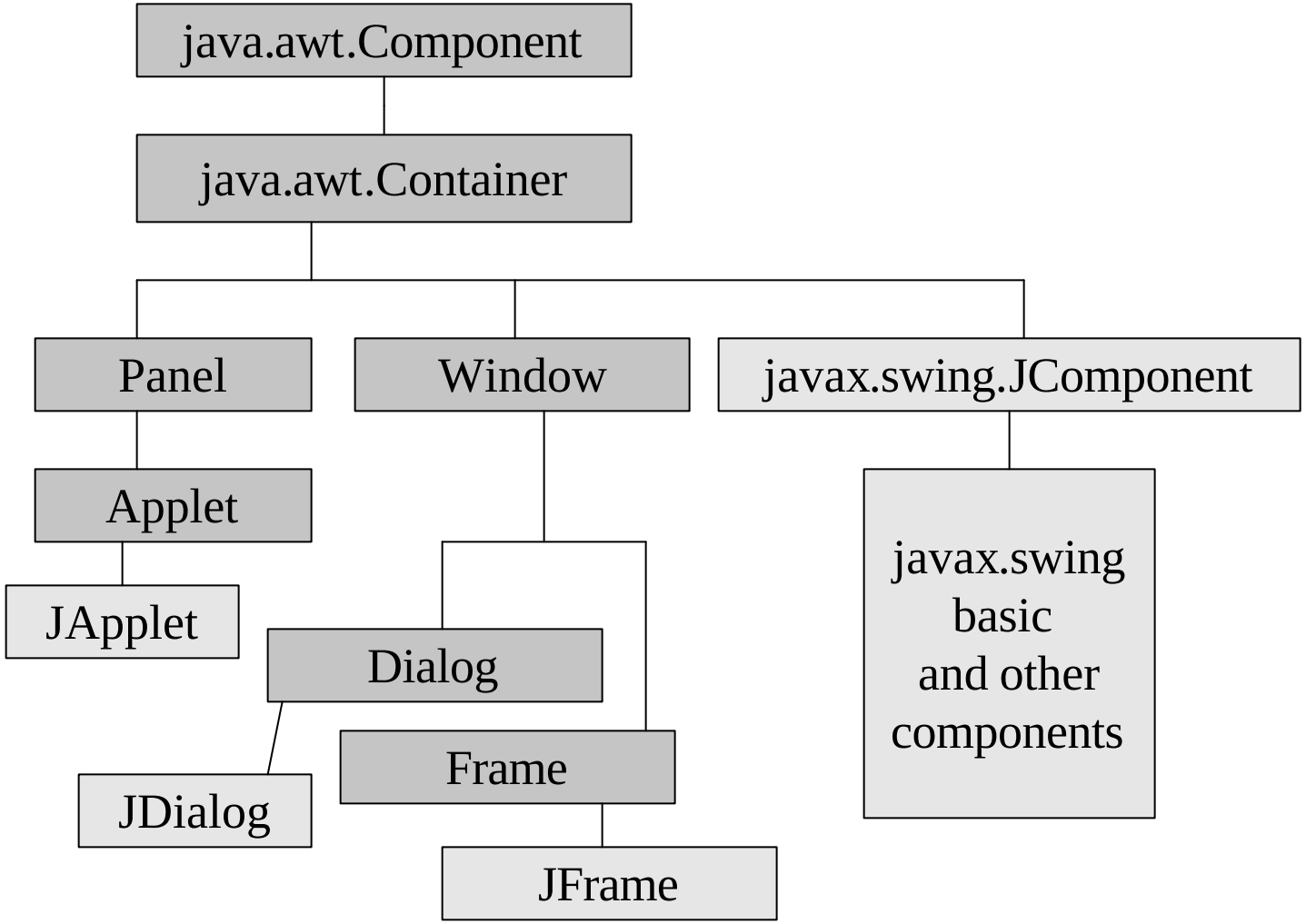
- Each component in a Layout may itself be a Panel with another Layout Manager, thus subdividing areas of the user interface. Using this hierarchy one can achieve complex GUI's.
- A simple example of using hierarchical Layout divides the main applet space into two components in a BorderLayout. The Center component is a Canvas for drawing graphics or images; the North component is itself a Panel which has three buttons in a GridLayout. This example is a very simple example of a standard paradigm for drawing or displaying data.



  - Also note other examples showing CardLayouts and GridBagLayouts.

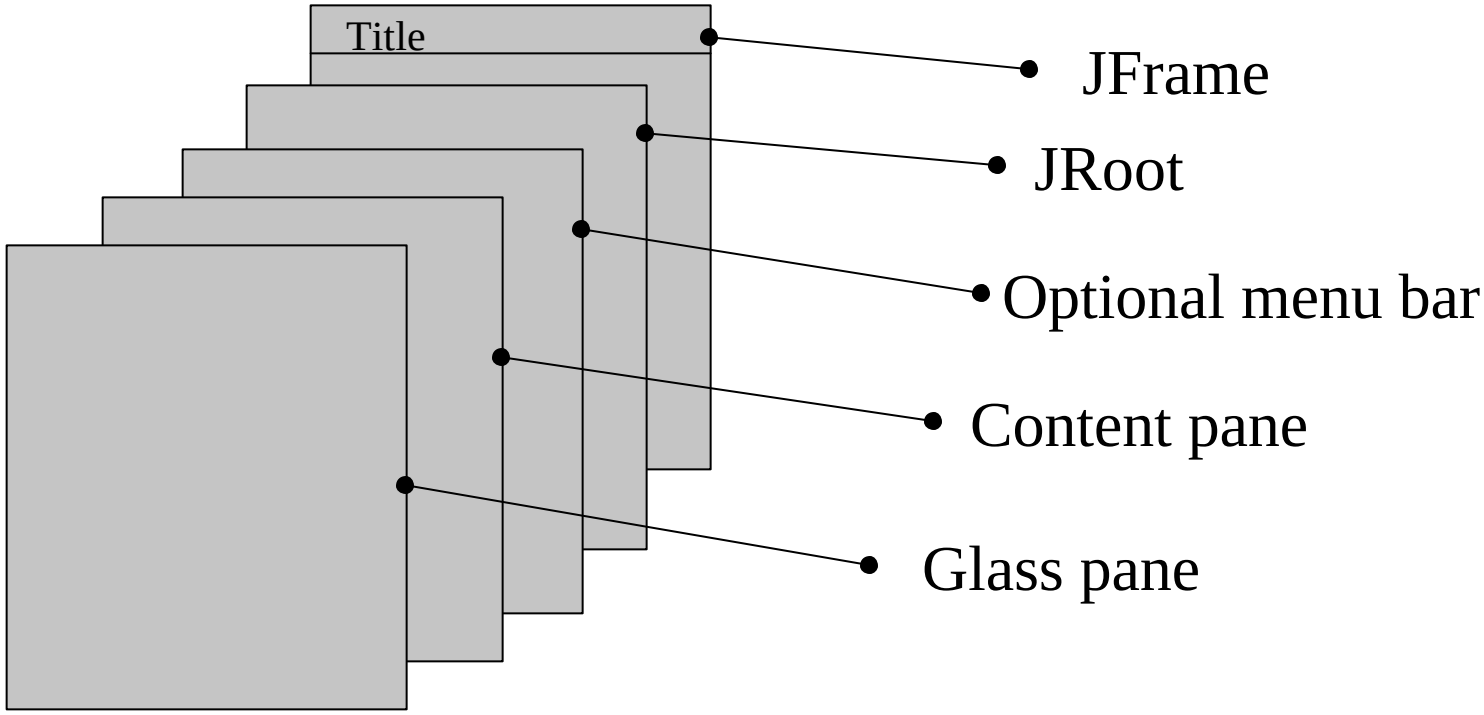
Abstract Windowing Toolkit (AWT): Swing Components and More Components of the AWT

The AWT and Swing Hierarchy

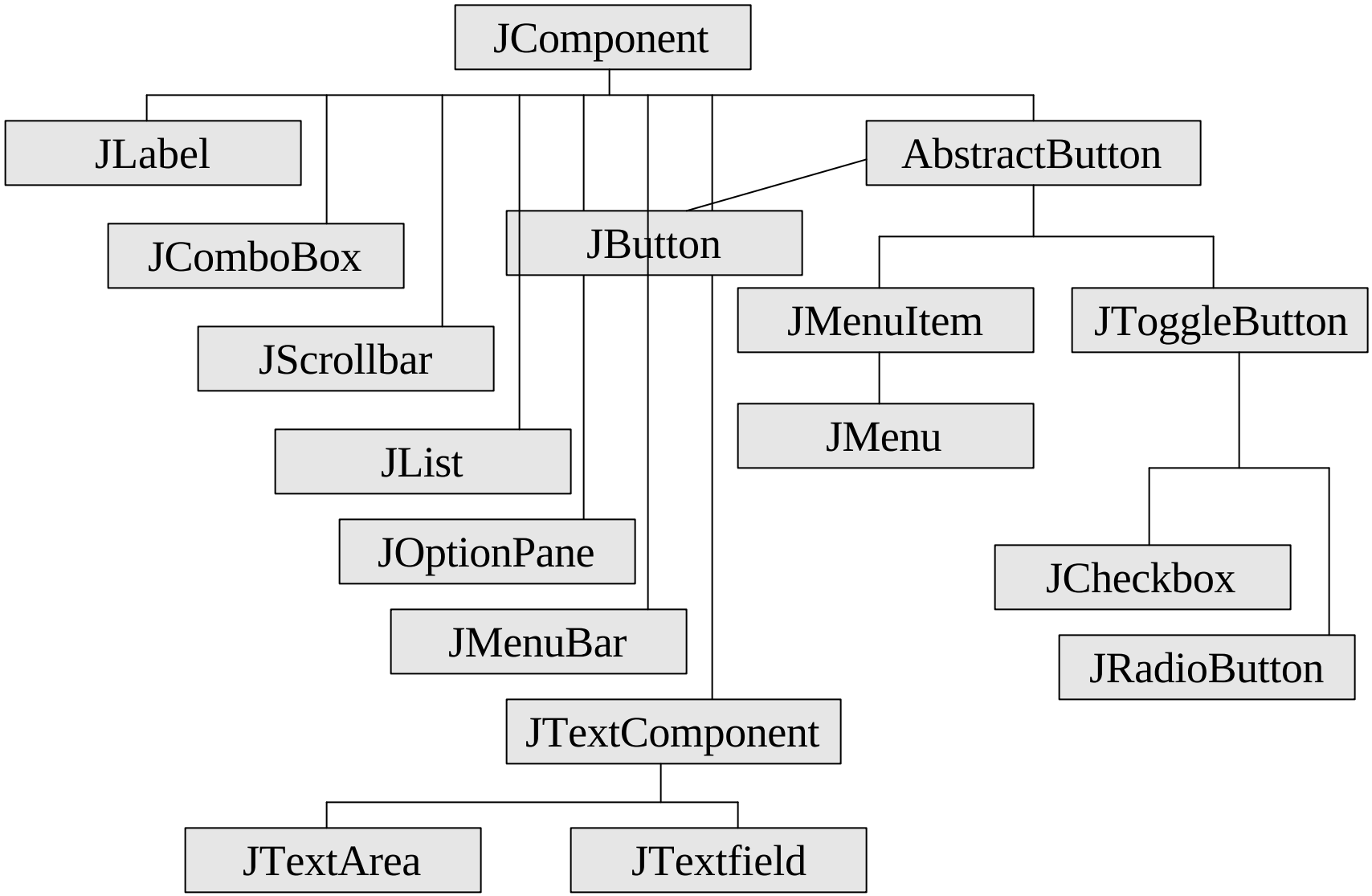


Internal Structure of a JFrame

- Applets actually reside inside Frames and JApplets inside JFrames. While applets, panels and other components paint and add directly, in the Swing set, things are painted and added to the ContentPane of the JFrame that they are in.



Swing Component Hierarchy



JLabel

- In addition to having a line of text and an alignment, Jlabels may have an icon, which is an image placed beside or above the text:

```
JLabel label =
    new JLabel ( “Text”, icon, SwingConstants.CENTER );
```

- The icon argument is anything that implements the interface Icon, such as ImageIcon.
- Note that the label alignment constants come from an interface called SwingConstants, which include LEFT, RIGHT, CENTER, NORTH, EAST, and so on.
- Where the icon is placed with respect to the text can be specified and fine-tuned with various horizontal and vertical alignment methods.

JButtons

- JButtons may also have an icon or text or both. The alignment methods are inherited from the abstract button class.
- In Swing, many components are implemented in terms of the model-view-controller design pattern. That is, there are separate classes for the different parts of the components:
  - contents, such as the state of the button, or the text of the textfield
  - visual appearance (color, size, and so on)
  - behavior (reaction to events)
- Note that the Abstract Button class allows the use of other models.
- JButtons, JCheckboxes, and JRadioButtons may have the same model, but different view and controller.

JTextField and JTextArea

- For compatibility with TextField, when the user hits “enter”, the JTextField fires an ActionEvent which can use the methods getText and setText to access the JTextField.
- However, the JTextComponents have additional DocumentEvents. When text has changed, one of the following three methods from the Document Interface is called:

```
void insertUpdate ( DocumentEvent e )
void removeUpdate ( DocumentEvent e )
void changedUpdate (DocumentEvent e )
```

- The JTextComponent class has methods to select text by highlighting it and to get selected text.
- Note that JTextArea must be put in a scroll pane to have scroll bars.

JCheckBox and JRadioButton

- The JCheckBox class is very similar to the Checkbox class (note the lower case b), except that
  - it generates ActionEvents instead of ItemEvents
  - and there is no CheckboxGroup
- Instead, there is the class JRadioButton. Individual radio buttons are like Checkboxes - they generate ActionEvents and have methods like isSelected().
- JRadioButtons should also be added to an object of class ButtonGroup. This enforces the rule that only one radio button can be selected at a time.

JComboBox

- The class JComboBox replaces Choice from the AWT. It combines a drop-down menu with a TextField. The user can select an item from the menu not only by clicking on it, but also (if setEditable(true) has been called) by typing an item into the TextField.
- The JComboBox can be constructed by giving the constructor an array of Strings (or icons), or by using addItem to add them to the set.
- Otherwise, the class has similar methods to Choice, such as getSelectedItem().

JLists

- The class Jlist can be constructed with an array or vector of items for the list.
- The property of single or multiple selection is set by
  list.setSelectionMode
  (ListSelectionModel.SINGLE_SELECTION);
- The number or rows to show in the list is
  list.setVisibleRowCount (8);
- The Jlist does not provide its own scrollbars but should be added to a JScrollPane.
- Adding and removing elements is handled by calling
  list.setListData (names);

  where names is the updated array or vector of items.
- Items can easily be Strings or Icons, but may also be any other object if you provide an implementation of the interface ListCellRenderer.

AWT Component: Canvas

- Canvas is a simple class that is used to draw on as in artist's canvas. They cannot contain other components. This is the class on which you use the graphics methods.
- Canvases can be included as parts of other Containers. In this case, you may need to obtain a Graphics object for drawing:
  - Canvas drawcanvas;
  - Graphics drawgraphics = drawcanvas.getGraphics();
    - » drawgraphics.drawString("Graph Label");

Some Further AWT Components: Window (Frame and Dialog)

- Up to now, we have described how to build typical Applet panels inside browser window. There are classes that allow one to generate complete windows separately from the browser window
- Window Class has subclasses Frame and Dialog
- Create a window with a given title:
  - Frame f = new Frame("TitleofWindow");
  - Frames appear and disappear by using the method setVisible:
    - » f.setVisible(true); / / makes the window appear on the screen
    - » f.setVisible(false); / / makes the window disappear from the screen
  - Note Frame is a Container and can thereof be defined hierarchically with components that are laid out by LayoutManagers
  - Note a Frame in Java is NOT THE SAME as a frame in HTML

Further methods and events for Windows

- Important properties to set for frames:
  - f.setTitle (“This string will be on the title bar of the window”);
    - » (This method inherited from the Window class.)
  - f.setSize (200, 500);
  - f.setLocation (100, 100);
    - » (These methods inherited from the class Component.)
- Window methods also include toBack(), toFront(), show(), which makes the window visible and in front, and hide(), which makes it not visible.
- Any window can have a WindowListener for WindowEvents, including:
  - windowClosing, windowOpened, windowClosed, windowIconified, windowDeiconified, windowActivated, windowDeactivated

Frames can have MenuBars

- A Frame has a set of classes to define MenuBars and Menus:
  - » Menubar mbar = new MenuBar(); / / defines a menubar which can be used in a frame
- Several Menus, which each have menu items, can be added to the menubar:
  - Menu typeMenu = new Menu("Type");
  - typeMenu.add(new MenuItem("Black/ White"));
  - typeMenu.add(new MenuItem("Color"));
  - typeMenu.addSeparator();
  - MenuItem q = new MenuItem("Quit"); q.addActionListener;
  - typeMenu.add (q);
  - mbar.add(typeMenu);
  - When the user selects an item on a menu, an ActionEvent is reported to the ActionListener, where it can be handled
    - » public void actionPerformed(ActionEvent e)
    - » { if (e.getSource() == q)
    - » {q.setVisible(false); } }

More on Menus

- MenuItems can be disabled or enabled by calling
  menuitem.setEnabled (false);
  (or true for the enabled case)
- There are CheckboxMenuItems, submenus, and MenuShortcuts.
- In JMenus, there are also RadioButtonMenuItems, keyboard shortcuts and accelerators. JMenuItems may have icons as well.

Dialog Boxes

- Another type of separate window is the Dialog box, which is not so elaborate as a frame.
  - » Dialog(Frame, String Title, boolean mustbeansweredatonceornot);
  - » / / defines a dialog box
- Dialog boxes are used for transient data
  - – Issue warning to user or require (third argument true) user to verify some action etc. - e.g. this dialog box has warning text and an ok button:
    - » class WarningDialog extends Dialog
    - » {public WarningDialog(Frame parent)
    - » {super(parent, "Dialog Box Label", true); / / modal requires immediate
    - » add(new Label("Read this stern Warning!"));
    - » Button b = new Button("OK"));
    - » b.addActionListener(this); add(b);
    - » setSize(200,100); }
    - » public void actionPerformed(ActionEvent e)
    - » {if (e.getActionCommand().equals("OK")) dispose();
    - » }
    - » }

New Components: ScrollPanes and PopupMenus

- A ScrollPane provides a scrollable area within an existing window. It is a container which can have one other component added to it. This component may have a larger area that will be visible in the ScrollPane. For example, suppose that you have a Canvas drawing area that is 1000 by 1000 pixels wide. You can define a ScrollPane that will view part of it at one time.
  - Canvas draw = new Canvas();
  - draw.setSize (1000, 1000);
  - ScrollPane drawscroller = new ScrollPane();
  - drawscroller.add (draw);
  - drawscroller.setSize (250, 250);
  - add(drawscroller);
- A PopupMenu is a sometimes called a context-sensitive menu. On most systems, the right mouse button cause a popup trigger event, and you can create a PopupMenu to appear.