Object–Oriented Design with UML and Java

PART XIII: GUIs

Copyright © David Leberknight

Version 2023

Object-Oriented GUIs

- **GUI** = Graphical User Interface
- Every thing on the screen is an object
- In addition, there are some objects that are not visible
- There are classes for everything, and they fall into hierarchies
- Much of the code is “event-driven”

This chapter covers the **Java AWT** and **Swing** frameworks.

Note the rise of **JavaScript** (also **TypeScript**) for browser GUIs.

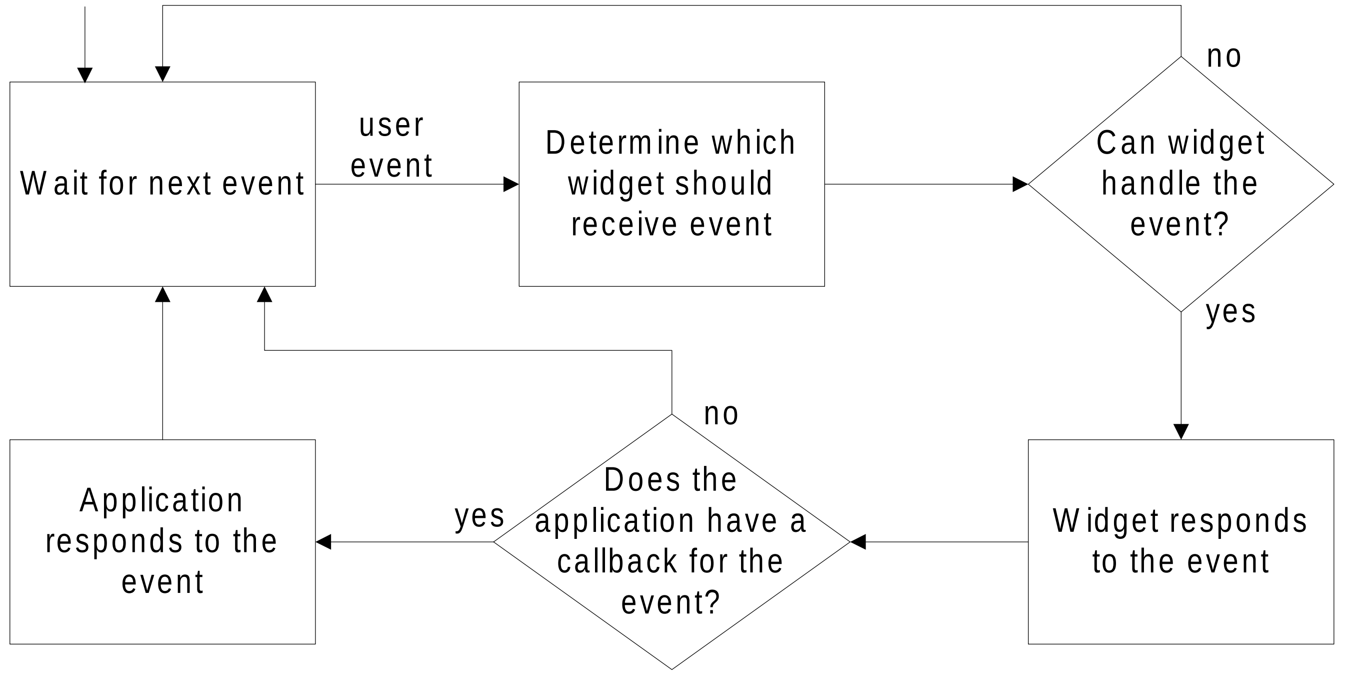
- The **JQuery** and **Angular JS** frameworks are popular.

Note also web-application frameworks such as **Java Server Faces**.

- Especially in conjunction with **Spring MVC**.

GUI Frameworks

GUI frameworks are similar in that they all have an *event loop* which receives an *event*, such as a mouse click, and *dispatches* it to the appropriate *widget*; the application also gets a chance to respond to the event, if it cares.


Wait for next event
user event
Determine which widget should receive event
Can widget handle the event?
no
yes
Widget responds to the event
Does the application have a callback for the event?
no
yes
Application responds to the event

Java’s AWT (Abstract Window Toolkit)

The AWT is reasonably well designed, but is not without gotchas.

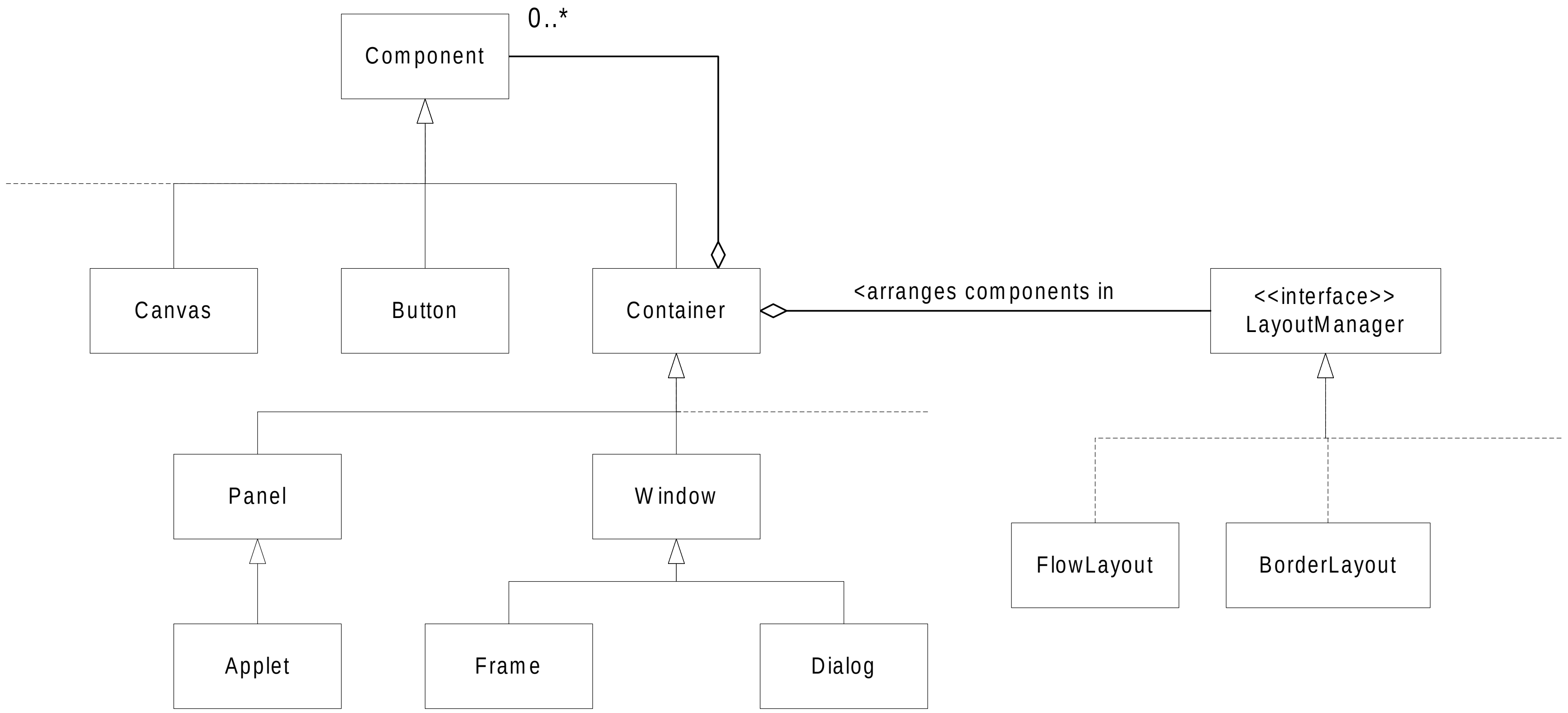
- You will need good reference materials.
- Do not rely on these notes for documentation. Look on-line…

In this class we are using Java, but you should learn JavaScript (also TypeScript) frameworks for programming user interfaces in the wild.

Basic Elements

- Components:
  - Button / List / Checkbox / Choice / TextField / Etc.
- Containers (subclass of Component):
  - Panel / Window / Dialog / Applet / Frame / Etc.
- Menu Components
  - Menu / Menu bar / Etc.
- Layout Managers
  - BorderLayout / GridLayout / Etc.
- Events
  - MouseEvent / MouseMotionEvent / ItemEvent / Etc.
- Graphics
  - Graphics / Image / Color / Font / FontMetrics / Etc.

Components, Containers, and Layout Managers



Layout Managers

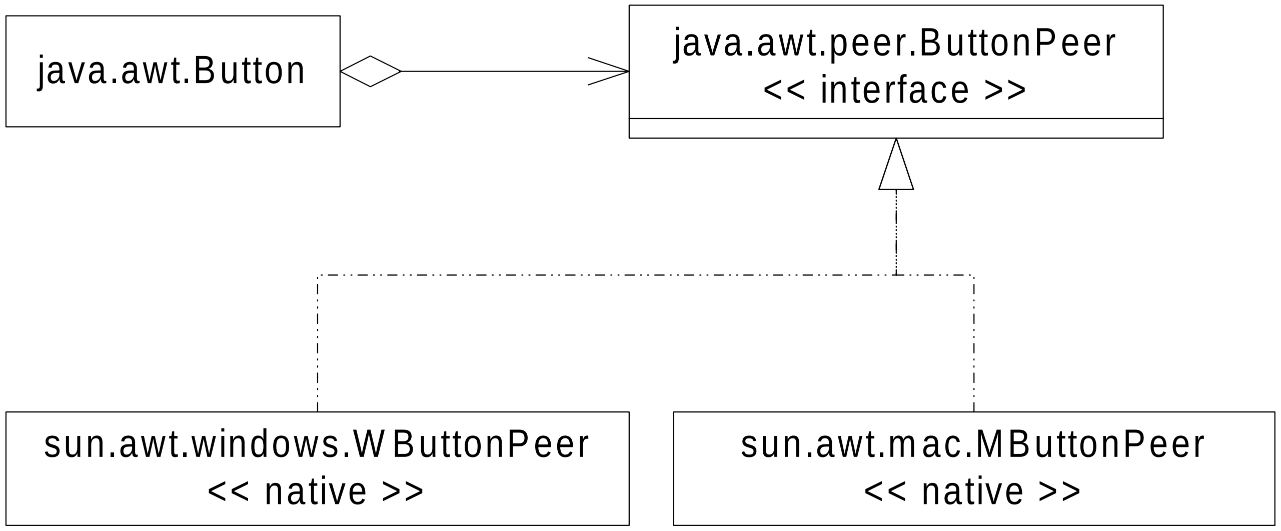
- A *LayoutManager* is responsible for arranging the *Components* inside a single *Container*. Containers are often nested hierarchically.
- LayoutManagers automatically adjust the layout of a Container whenever the Container gets resized, and just before the first call to **`paint()`**.
- Containers *do not* know how to do layout management.
- Containers *do* know how to add and remove children Components.
- A Component object can only be inside one Container.
- The Container's layout Strategy may be changed at run-time (but usually isn’t).
- Components may define size preferences, which may or may not be honored; the layout manager may choose to ignore this information. It is common to have Component subclasses override the **`getMinimumSize()`**, **`getMaximumSize()`** and **`getPreferredSize()`** methods, because unfortunately, there is no **`setPreferredSize()`** method (Swing fixes this).
- Similarly, it is sometimes necessary to override **`getInsets()`**.
- **Always** use layout managers to avoid hard-coding sizes and locations.

Peers (JDK 1.1)

- The implementation of many AWT widgets (JDK 1.1.x) is different on different platforms. For example, an AWT Button is implemented by delegating to a (native) Windows Button on the Windows platform, and a (native) Mac Button on the Mac. These native widgets are called "peers".   Peers were designed to preserve the "native look and feel" for each platform, but they can be different sizes on different platforms (and have slightly different behaviors).
- The new "*Swing* Set" (of Components) fixes this by providing 100% pure Java Components, with various 100% pure Java “pluggable look and feels.”
- To get around some of the difficulties associated with the native peer model, and to take advantage of other nice features, *always* use the AWT's *Layout Managers* instead of trying to hard-code any x,y coordinate locations for widgets.
- Peers are implemented using the ***Bridge*** and ***Abstract Factory*** design patterns.

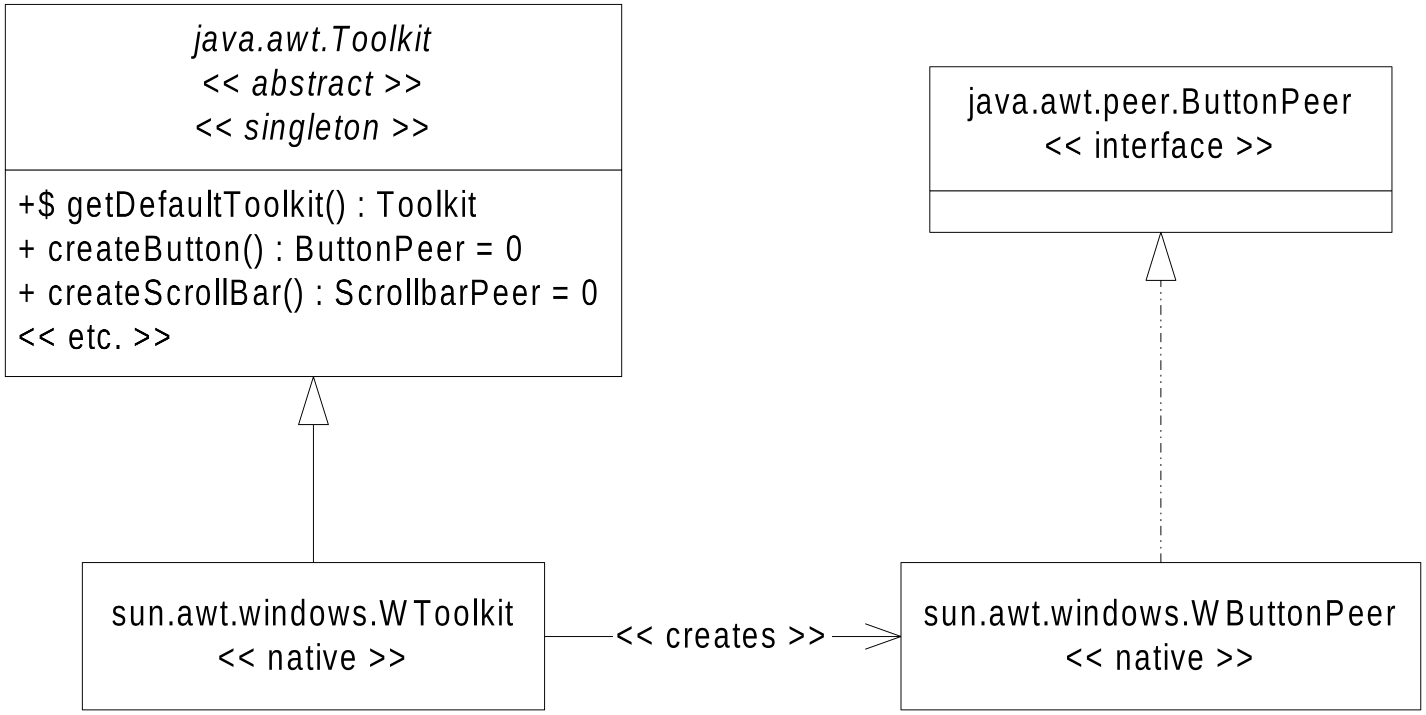
Peers (cont.)

- The Java AWT (pre-Swing) uses the *Bridge* pattern to separate the widget (component) abstractions from the platform dependent “peer” implementations.
- The **`java.awt.Button`** class is 100% pure Java, and is part of a larger hierarchy of GUI components. The **`sun.awt.windows.WButtonPeer`** class is implemented by native Windows code.



Peers (cont.)

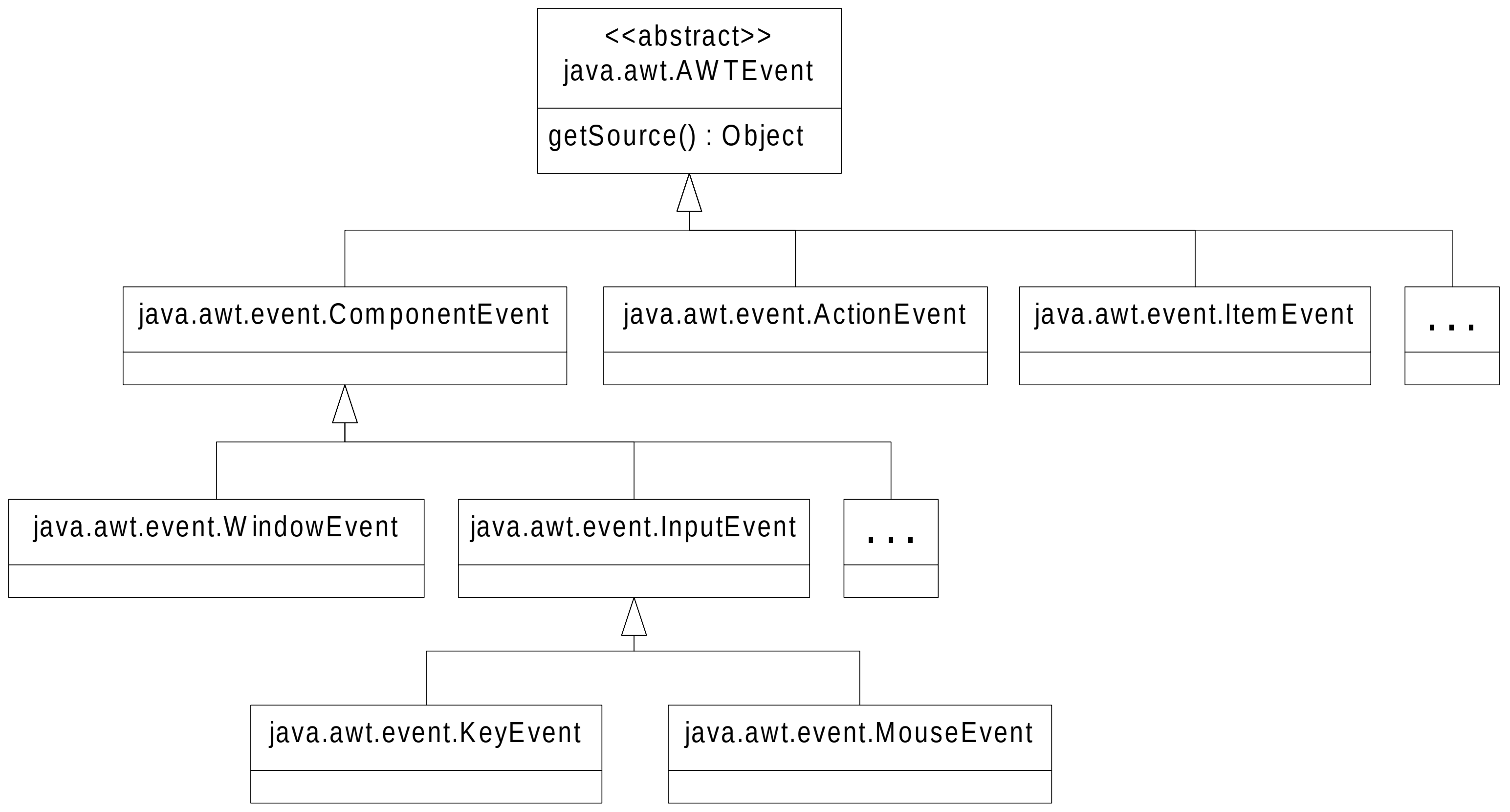
- The *Abstract Factory* pattern is used to create the correct *family* of peers.



Events

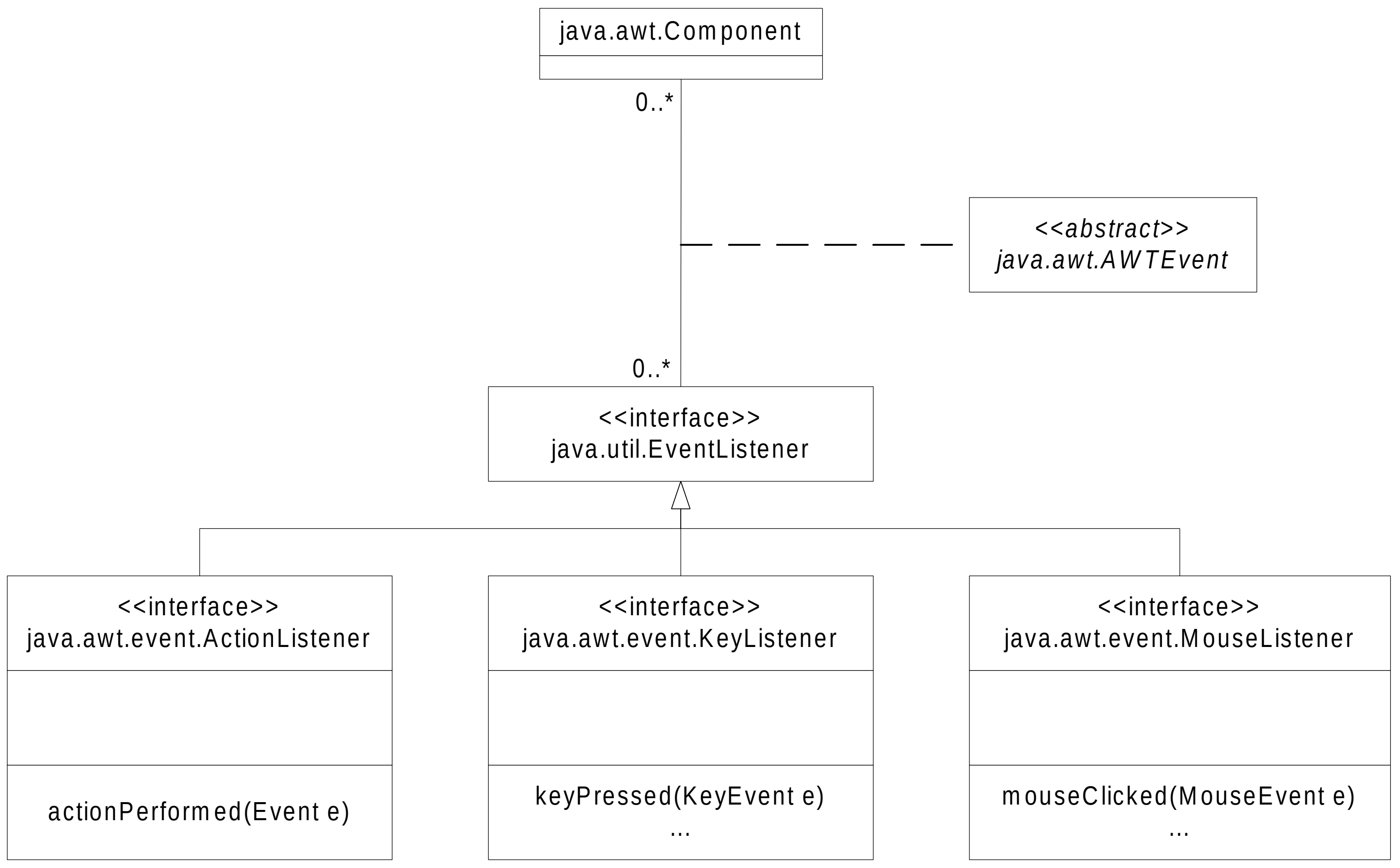
- Events originate from a source Component and are automatically sent to the registered listener object(s) ala the ***Observer*** design pattern. A Component may have more than one listener. A listener may listen to more than one Component.
- All AWT / Swing events (including paint) operate through a single thread, using a queue. Consider clicking the mouse over a button… the following events are generated:
  1. mouse down at (x,y);
  2. mouse up at (x,y);
  3. mouse click at (x,y);
  4. focus lost at whatever widget previously had the focus;
  5. focus gained at button;
  6. action at button.
- These events might be in a different order on different platforms.
- An application may put "events" onto this queue using `SwingUtilities.invokeLater( runnable )`.

Events (cont.)



<<abstract>>
java.awt.AWTEvent
getSource() : Object
java.awt.event.ComponentEvent
java.awt.event.ActionEvent
java.awt.event.ItemEvent
. . .
java.awt.event.WindowEvent
java.awt.event.InputEvent
. . .
java.awt.event.KeyEvent
java.awt.event.MouseEvent

Events (cont.)


java.awt.Component
0..*
<<abstract>>
java.awt.AWTEvent
0..*
<<interface>>
java.util.EventListener
<<interface>>
java.awt.event.ActionListener
actionPerformed(Event e)
<<interface>>
java.awt.event.KeyListener
keyPressed(KeyEvent e)
...
<<interface>>
java.awt.event.MouseListener
mouseClicked(MouseEvent e)
...

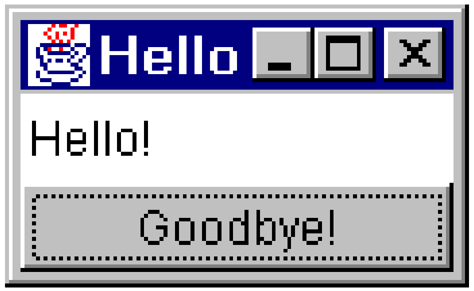
AWT 1.1 Design Patterns

- Components
  - Components and Containers (which are kinds of Components) are usually arranged in an object hierarchy, ala the ***Composite*** design pattern. The widget abstractions are separated from the native peer implementation (pre swing) using the ***Bridge*** pattern. The ***Abstract Factory*** & ***Singleton*** patterns are used to construct the correct family of native peers.
- Layout Managers
  - Layout managers control size and position of all Components within a Container. Container components may use different layout manager ***Strategies***. For example, a Frame (Container type Component) could use a GridLayout or a BorderLayout strategy.
- Events
  - Source / Listener model uses the ***Observer*** Design Pattern. Objects register themselves as listeners to events of interest. For example, a Button will want to listen to "action" events (mouse clicks) using an ActionListener.

GUI Application Structure

- Most GUI applications in AWT (pre-Swing) will have, as the “main” class, an application-specific subclasses of *Frame* or *Applet*.
- Most other GUI components are instances of AWT classes:
  - *Menu / MenuItem / Button / Panel* / Etc.
- All AWT classes can be subclassed to create custom components. Care should be taken when overriding AWT methods.
- Events get passed to registered *Listeners* - instances of application-specific classes that implement Listener interfaces.
- For a GUI Application, use a ***Frame.*** There is usually only one instance of one subclass of Frame. Only a Frame can have a menu bar.
- Your subclass of Frame can have **`main()`**, or it can be instantiated by some other class that represents the whole application, which has **`main()`**.
- The Frame should set up its Panels, LayoutManagers, Menus, Buttons, etc., when its constructor executes, or just afterwards in an **`init()`** method.

Example: Hello Goodbye



Hello Goodbye (cont.)

```
package awtExamples.helloGoodbye;
import java.awt.*;
import java.awt.event.*;
public class HelloGoodbye extends Frame {
  public HelloGoodbye() {
    super( "Hello" ); // Set’s the Frame’s Title
  }
  public void init() {
    Label helloLabel = new Label( "Hello!" );
    Button goodbyeButton = new Button( "Goodbye!" );
    setLayout( new BorderLayout() ); // for the Frame
    add( helloLabel, BorderLayout.NORTH ); // Frame is a Container
    add( goodbyeButton, BorderLayout.SOUTH );
    goodbyeButton.addActionListener( new GoodbyeButtonHandler() );
    addWindowListener( new WindowEventHandler( ) );
  }
```

Hello Goodbye (cont.)

```
 public static void main( String args[] ) {
    HelloGoodbye hg = new HelloGoodbye();
    hg.init();
    hg.pack(); // invoke the layout manager
    hg.show(); // set visible and start AWT threads
  } // main() terminates, but the program continues to run…

  // *** Inner Classes that “observe” AWT components *** //
  class GoodbyeButtonHandler implements ActionListener {
    public void actionPerformed( ActionEvent e ) {
      System.exit( 0 );
  } }
  class WindowEventHandler extends WindowAdapter  {
    public void windowClosing( WindowEvent we )  {
      System.exit( 0 );
} } }
```

Java Applets

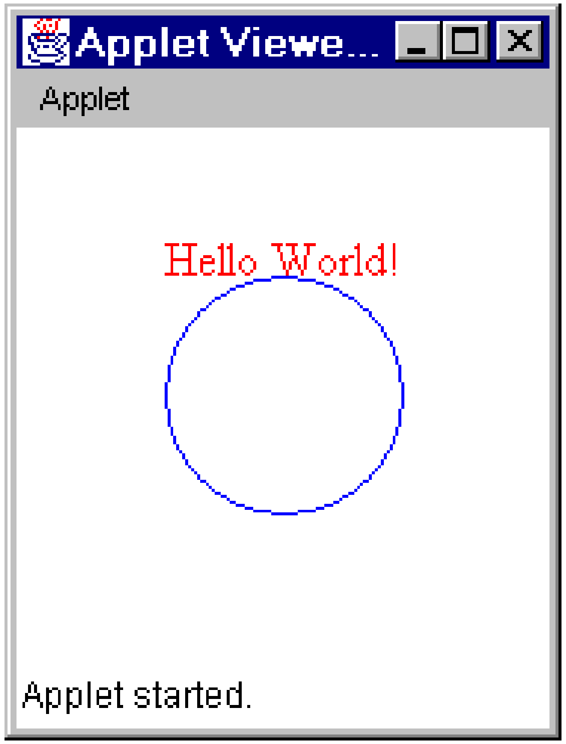
- The role of the Frame is taken over by a browser. The browser first allocates real estate based on the width and height in the HTML tags. Then the browser's *Java Virtual Machine* (JVM) verifies the *bytecode* for the applet's .class file as it gets loaded from the network; then it creates a new instance of the class. The JVM invokes the applet's "life cycle" methods as appropriate; it also notifies the applet of Events, and calls **`update()`** / **`paint()`** when appropriate.

- Special applet "life cycle" methods:
  - **`init() ...`** Called when the applet is first loaded. This is where constructor-type things are done.
  - **`destroy() ...`** Called when the applet is about to be unloaded.
  - **`start() ...`** Called when the applet becomes visible.
  - **`stop() ...`** Called when the applet becomes temporarily invisible.
- The applet has no constructor.

Java Applets (cont.)

- The JVM insulates the Java program from the underlying platform…
- Portability! But this is not perfect… "write once… debug everywhere"
- Security restrictions for "untrusted" applets, inside the "sandbox":
  - *Cannot* access the local machine environment, including the disk.
  - *Cannot* call a non-Java (native) method, etc...
  - *Can* communicate with the server from where it originated. The applet can thus access the server's disk, subject to certain rules, and it can communicate to the server via a Socket or via the Java Servlet API.
- Sun's Java WebStart ™ technology shares the primary advantage of applets (ease of web-based deployment) while bypassing the dependency on Internet browsers: **`http://java.sun.com/products/javawebstart/`**
- Rich Internet Applications (RIA) using AJAX (Asynchronous JavaScript and XML) have become the standard for serious web-based application development. This topic is beyond the scope of this course.

“Hello World” Java Applet



“Hello World” Java Applet

- A special application-specific **`HTML`** file tells the browser how to launch the applet.

```
<HTML><APPLET code=”awtExamples.helloApplet.HelloApplet.class"
              width=180 height=180>
<PARAM name=”GREETING" value=”Hello World!"></APPLET></HTML>
```

- The class **`HelloApplet`** must be an extension (subclass) of class Applet.
- Note: the **`.html`** file must be in the parent directory of the **`awtExamples`** subdirectory, whose subdirectory **`helloApplet`** contains the **`.class`** files.
- The code resides in two files: **`HelloApplet.java`** and **`HelloApplet.html`**.
- **`HelloApplet`** gets its greeting during **`init()`** from an **`HTML`** parameter.
- **`paint()`** gets called automatically by the native Window manager whenever necessary, such as when part of the Component's real estate gets re-exposed after being hidden by another window, or sometime after a call to **`repaint()`**.
- Notice that the circle and the text are drawn at the same x,y location. The difference between drawing text and drawing graphics is described later...

“Hello World” Java Applet (cont.)

```
package awtExamples.helloApplet;

import java.applet.Applet;
import java.awt.*;

public class HelloApplet extends Applet
{
  private String greeting;
  private int x = 50;
  private int y = 50;

  public void init()
  {
    greeting = getParameter( "GREETING" );
    if( greeting == null ) greeting = "Hello!";
  }
```

“Hello World” Java Applet (cont.)

```
    public void paint( Graphics g )
    {
      g.setColor( Color.blue );
      g.drawOval( x, y, 80, 80 );
      g.setColor( new Color( 255, 0, 0 ) ); // Color.red
      g.setFont( new Font( "TimesRoman", Font.PLAIN, 16 ) );
      g.drawString( greeting, x, y );
    }

    //  public void update( Graphics g ) // unnecessary
    //  {
    //    // Don’t redraw the background.
    //    paint( g );
    //  }
  }
```

Testing Applets

- Most browsers cache Java code differently from the way they cache HTML pages, and so you must be careful to be sure that you're running the right code...
- Test your code on various hardware platforms using different browsers. Remember to always clear the browser's cache before running a new test (since browsers often use cached Java code instead of downloading the new version ;-(
  - To clear the cache in Netscape… Edit… Preferences… Advanced… Cache… Clear Memory Cache / Clear Disk Cache.
  - To clear the cache in Internet Explorer… Tools…Internet Options… General… Temporary Internet Files… Delete Files.
- Sometimes you need to also delete all of your *local* .class files in order to test the *deployed* Applet since some browsers will use class files they find on your local disk!
- When developing an Applet, it is sometimes nice to wrap it in an application Frame, which behaves like a browser, so that it is easier to debug.
  - Example: **The Fractal Applet** runs as both an Applet and an Application.

Java.awt.Graphics

- Java makes it easy to do simple graphics...

```
public abstract class Graphics {
  ...
  public abstract void dispose(); // free system resources
  public abstract void drawImage( Image i, int x, int y, ... );
  public abstract void drawLine( int x1, int y1, int x2, int y2 );
  public abstract void drawOval( int x, int y, int w, int h );
  public abstract void drawString( String s, int x, int y );
  public abstract void fillRect( int x, int y, int w, int h );
  public abstract void setColor( Color c );
  public abstract void setFont( Font f );
  ...
}
```

Java.awt.Graphics (cont.)

- You will never "new" an instance of the class **`Graphics`**. A Graphics object will be provided to you by the AWT.
- The default behavior of **`update()`** is to draw the background and then call **`paint()`** to draw the foreground. People typically override **`update()`** to do nothing except call **`paint()`** to reduce visible “flickering”...
- **`repaint()`** schedules an *asynchronous* call to **`update()`** in another Thread.
- Refrain from doing long calculations from within **`paint()`**.
- More sophisticated techniques are required for faster performance.
  You may get a Graphics object from an **Image** to do painting “off screen” (using a technique called “*double buffering*” described later);
  you may get a Graphics object from a **Canvas** and paint without **`paint()`**.

```
Graphics g = canvasRef.getGraphics(); // not inside paint()
g.fillOval( ... );  // immediately draws onto canvas
```

Java.awt.Component

```
public abstract class Component ... {
  ...
  public synchronized void addKeyListener( KeyListener kl );
  public synchronized void addMouseListener( MouseListener ml );
  ...
  public Image     createImage( int w, int h );
  public Dimension getPreferredSize();
  public Insets    getInsets();
  public int       getWidth();
  public void      paint( Graphics g );
  public void      repaint();
  public void      update( Graphics g );
  public void      validate();  // for dynamic layout management
  ...
}
```

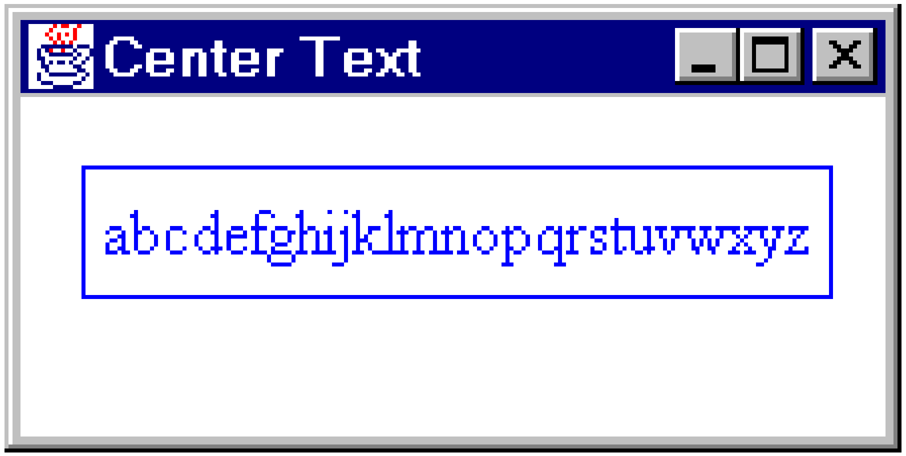
AWT Examples

- Following are some example programs that use the Java AWT…
- These programs are designed to demonstrate important features of the AWT.
- The proper techniques for doing some things with the AWT are not always obvious...
- Notice how each Application uses a slightly different coding style to implement the listener for the windowClosing event (generated by clicking the little **`x`** in the upper right corner of the Application’s Frame).

Also refer to:

`https://www.leberknight.com/fractalApplet/fractaljava.html`

Example: Center Text

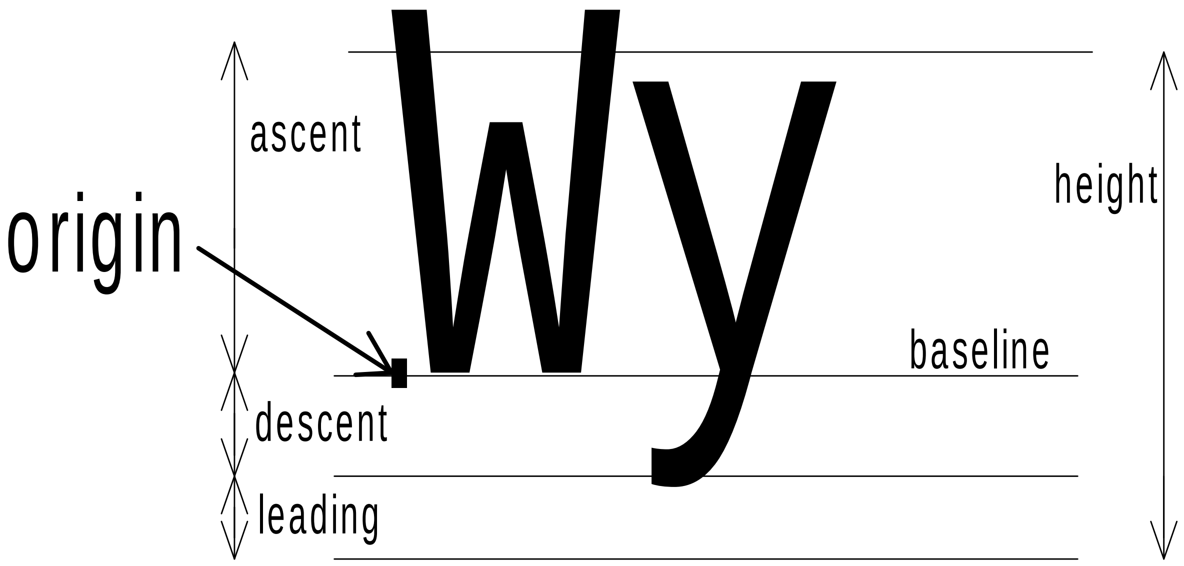


abcdefghijklmnopqrstuvwxyz

- The ***CenterText*** application paints text, centered inside a rectangle. Note that **`g.drawRect()`** uses a different origin point than **`g.drawString()`**.
- In this example, the class CenterText itself implements the WindowListener interface, to listen for the WindowClosing event.

Example: Center Text (cont.)

Drawing text is slightly different from drawing graphics; in particular, they use a different reference point (origin). For example, **`g.drawOval()`** uses the upper left corner as its origin. In contrast, the origin point for **`g.drawString()`** is shown below. The quantities *ascent*, *descent*, *leading* & *height* (plus *width*, given a String) can be obtained from a **`FontMetrics`** object.



Example: Center Text (cont.)

```
package awtExamples.centerText;
import java.awt.*;
import java.awt.event.*;
public class CenterText extends Frame implements WindowListener {
  private String textToCenter = "abcdefghijklmnopqrstuvwxyz";

  public static void main( String[] args ) {
    new CenterText( );
  }

  CenterText( ) {
    super( "Center Text" );
    addWindowListener( this );
    setSize( 222, 111 ); // works OK for Frame
    show( );
  }
```

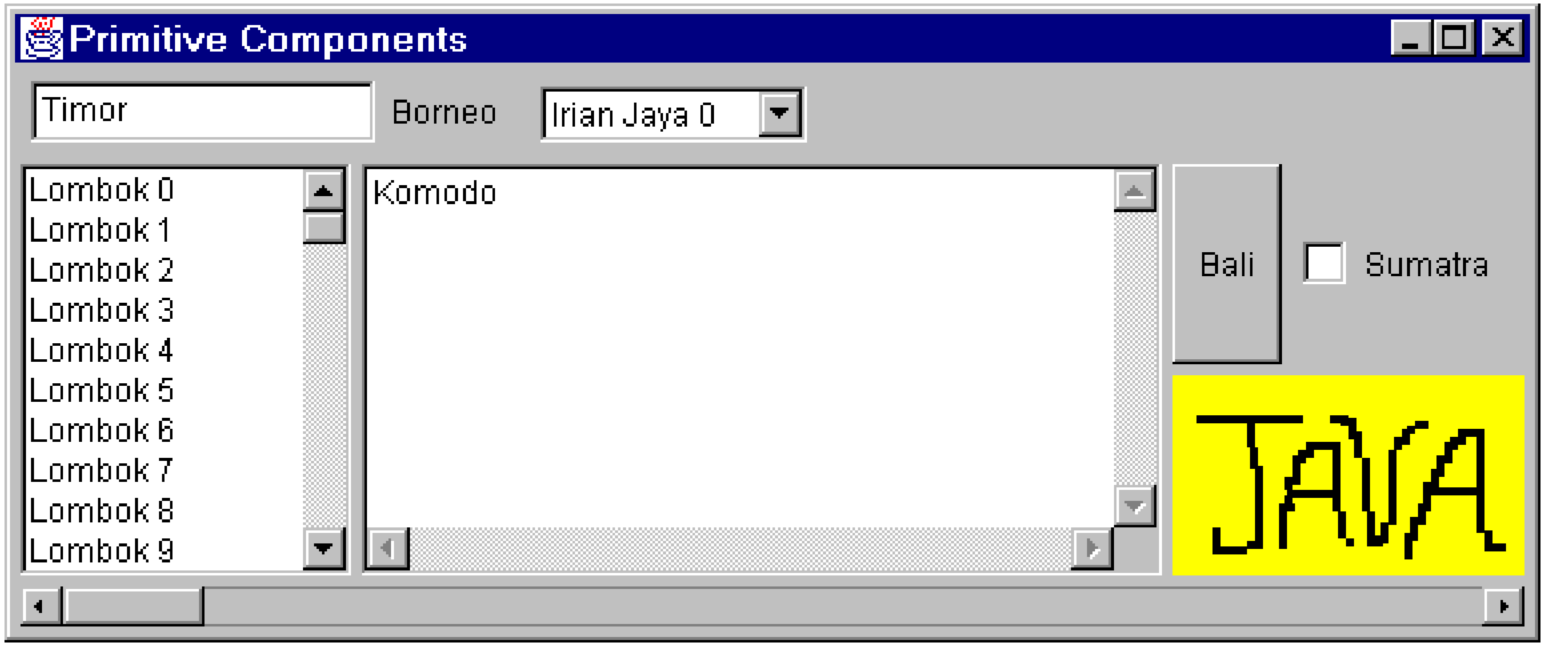
Example: Center Text (cont.)

```
void paint( Graphics g )
{
  g.setFont( new Font( "TimesRoman", Font.PLAIN, 16 ) );
  g.setColor( Color.blue );
  FontMetrics fm = g.getFontMetrics();
  int numPixBuffer = 5;
  int xTextOrigin = (getWidth()-fm.stringWidth( textToCenter ))/2;
  int yTextOrigin = (getHeight()-fm.getDescent()+fm.getAscent())/2;
  int rectHeight = fm.getHeight() + 2 * numPixBuffer;
  int rectWidth = fm.stringWidth( textToCenter ) + 2 * numPixBuffer;
  int xRect = xTextOrigin - numPixBuffer;
  int yRect = yTextOrigin - fm.getAscent() - numPixBuffer;
  g.drawRect( xRect, yRect, rectWidth, rectHeight );
  g.drawString( textToCenter, xTextOrigin, yTextOrigin );
}
```

Example: Center Text (cont.)

```
  // Implementation of the WindowListener interface:
  public void windowClosing( WindowEvent we ) {
    // The user just clicked on the little "kill" X, or equivalent.
    we.getWindow().dispose(); // garbage
    System.exit( 0 );
  }
  // We have to implement these methods, since they are part of the
  // WindowListener interface, but we don't care about them:
  public void windowActivated( WindowEvent we ) { }
  public void windowClosed( WindowEvent we ) { }
  public void windowDeactivated( WindowEvent we ) { }
  public void windowDeiconified( WindowEvent we ) { }
  public void windowIconified( WindowEvent we ) { }
  public void windowOpened( WindowEvent we ) { }
}
```

Example: Primitive Components



PrimitiveComponents shows each of the 9 primitive (“leaf”) AWT Components: TextField, Label, Choice, List, TextArea, Button, Checkbox, Scrollbar, & Canvas (with “erase-on-paint” scribble feature), plus a hierarchical nesting of Containers, and a couple of different LayoutManagers.

Example: Primitive Components (cont.)

```
Frame with BorderLayout
   NORTH: Panel with FlowLayout
      TextField / Label / Choice
   SOUTH: Scrollbar
   WEST: List
   CENTER: TextArea
   EAST: Panel with GridLayout
      Panel with BorderLayout
         WEST: Button
         EAST: Checkbox
      Canvas
```

- There are listeners for most Events, which dump interesting info to The Java Console (found by navigating through the browser’s menus).
- In this example, a separate class, WindowHandler, implements the WindowListener interface, to listen for the WindowClosing event.

GUI Design - Layout

Sketch the desired appearance on a piece of paper first.

See if any hierarchical nesting of simple LayoutManagers will achieve the desired effect:

- BorderLayout (north, south, east,west, center)
- GridLayout (regular, evenly-spaced grid)
- FlowLayout (uses preferred sizes, might “wrap” around if space too small)

If your design is too complex for simple LayoutManagers, consider:

- GridBagLayout (more complex, but also more powerful)
- CardLayout (to create tabs)

Note: Some LayoutManagers respect preferred sizes, while others “stretch” the components to “fill” the available space.

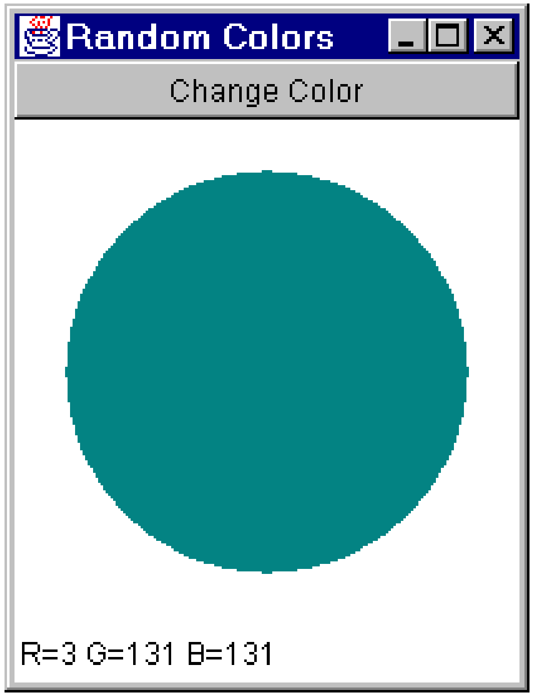
Sometimes it is necessary to experiment, since LayoutMangers do not always do exactly what you would want them to do...

Example: Random Colors

This code defines a subclass of Canvas called ColorCanvas to display the graphic; override getPreferredSize() for the Application version.

Note the use of ExitOnClose, a class that extends WindowAdapter, which is an AWT "convenience class" (used instead of WindowListener).

Also note the call to repaint().

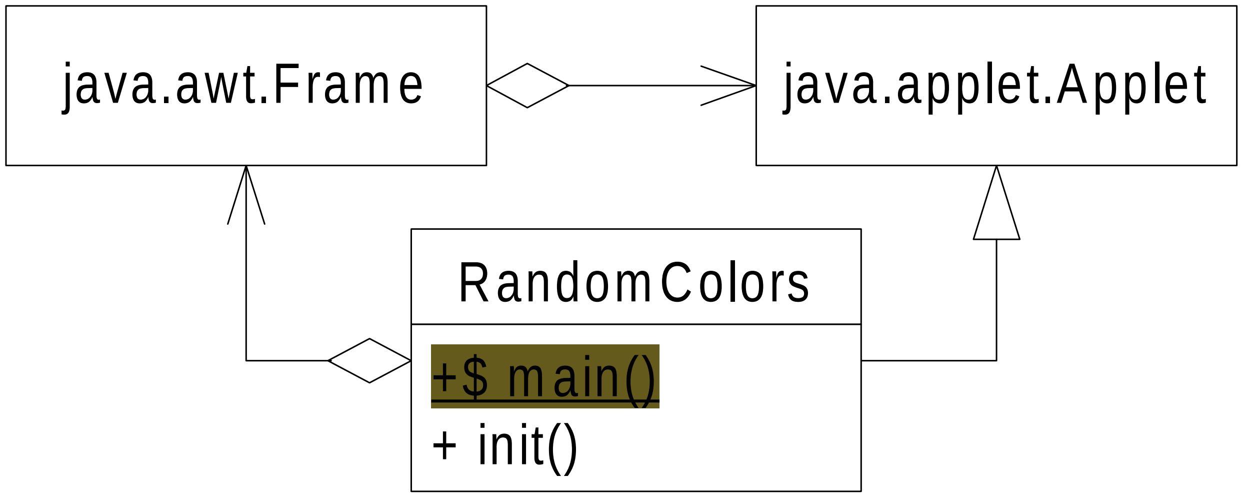


R=3 G=131 B=131

```
<HTML>
<HEAD><TITLE> RandomColors - Applet Version
</TITLE></HEAD>
<APPLET code =
"awtExamples.randomColors.RandomColors.class"
 width=300 height=300>
</APPLET>
</HTML>
```

Example: Random Colors (cont.)

- If main() is invoked, then this code runs as an Application, and it creates a Frame, into which it puts an instance of itself; in this case, the class Applet substitutes for its superclass, Panel. Invoking show() on the Frame starts up the AWT threads.
- If the RandomColors class is referred to from an HTML file, then it is an Applet, and main() is never invoked. The browser is used as a container instead of a Frame.



Example: Random Colors (cont.)

```
 public static void main( String[] args ) {
    isApplet = false;
    try {
      Applet randomColorsApplet = new RandomColors();
      Frame applicationFrame = new Frame( "Random Colors" );
      randomColorsApplet.init();
      applicationFrame.setName( "RandomColors Frame" );
      applicationFrame.setLayout( new BorderLayout() );
      applicationFrame.addWindowListener( new ExitOnClose() );
      applicationFrame.add( randomColorsApplet, BorderLayout.CENTER );
      applicationFrame.pack();
      applicationFrame.show();
    }
    catch( Throwable t ) {
      System.out.println( "Random Colors ERROR !!! " + t );
}  }
```

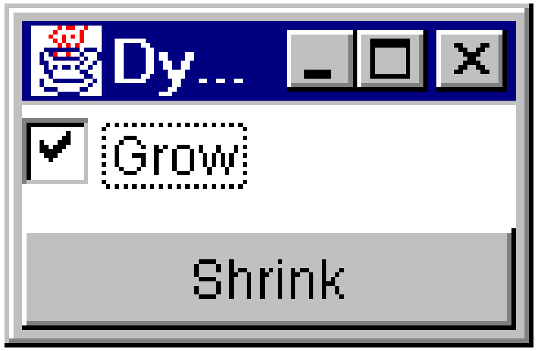
Adapters vs. Listeners (ExitOnClose)

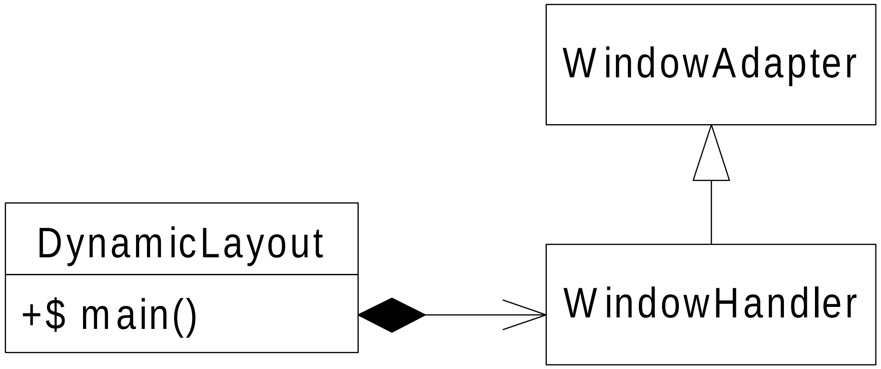
```
// The AWT provides handy "adapter" classes as a convenience...
// The class WindowAdapter provides do-nothing defaults
// for all of the methods defined by the WindowListener interface.

class ExitOnClose extends WindowAdapter
{
  public void windowClosing( WindowEvent we )
  {
    System.out.println( "GOODBYE !!! " + we );
    we.getWindow().dispose(); // garbage
    System.gc();
    System.exit( 0 );
  }
}
```

Example: Dynamic Layout

- **DynamicLayout** plays around with the Frame’s layout at run time.
- A LayoutManager can be told to recalculate the layout by telling the Container to **`validate()`** itself. Frame's method’s **`show()`** and **`pack()`** call **`validate()`**.
- Note: in order to do this in a platform-portable way, it is ***necessary*** to call **`validate()`** on the ***outermost*** Container in the containment hierarchy (the Frame, Dialog, or Applet). In this case, the Frame is already the outermost Container, so it is easy.
- The Fractal Applet’s Julia Set controls do the same thing in a more sophisticated way. Look at the code in Fractal’s **`recalculateLayout()`** method!
- Note the *inner classes*, particularly: **`DynamicLayout$WindowHandler.class`**

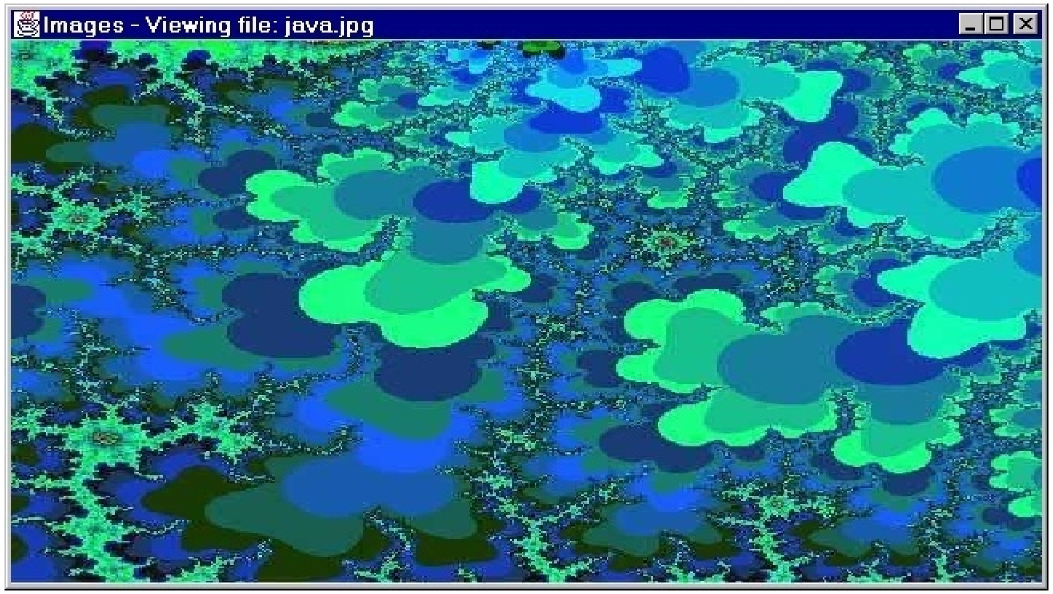




Example: Images

Images runs as an Applet and an Application, which is a bit tricky because the Applet version must load Image files from across the network.

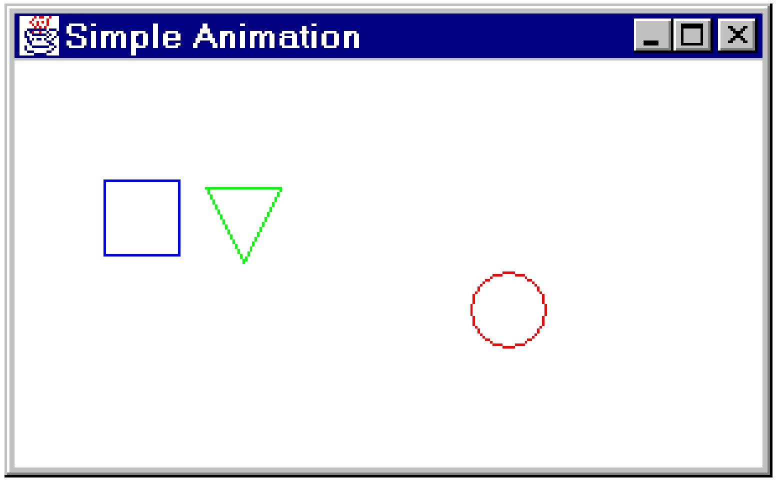
- Both versions load and display a **`.jpg`** file from the *Document Base*, using a **`URL`** to locate the file, and a **`MediaTracker`** to block until the Image loads.
- Note the ***anonymous*** *inner class* that extends WindowAdapter: **`Images$1.class`**
- Note the various differences between the Applet & Application versions.

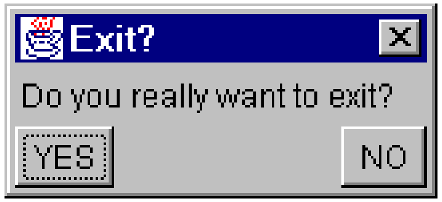


Example: Simple Animation

SimpleAnimation does relatively unsophisticated animation using 3 Threads, one for each *Sprite*. Note: this is a toy program, designed for playing around with Threads.

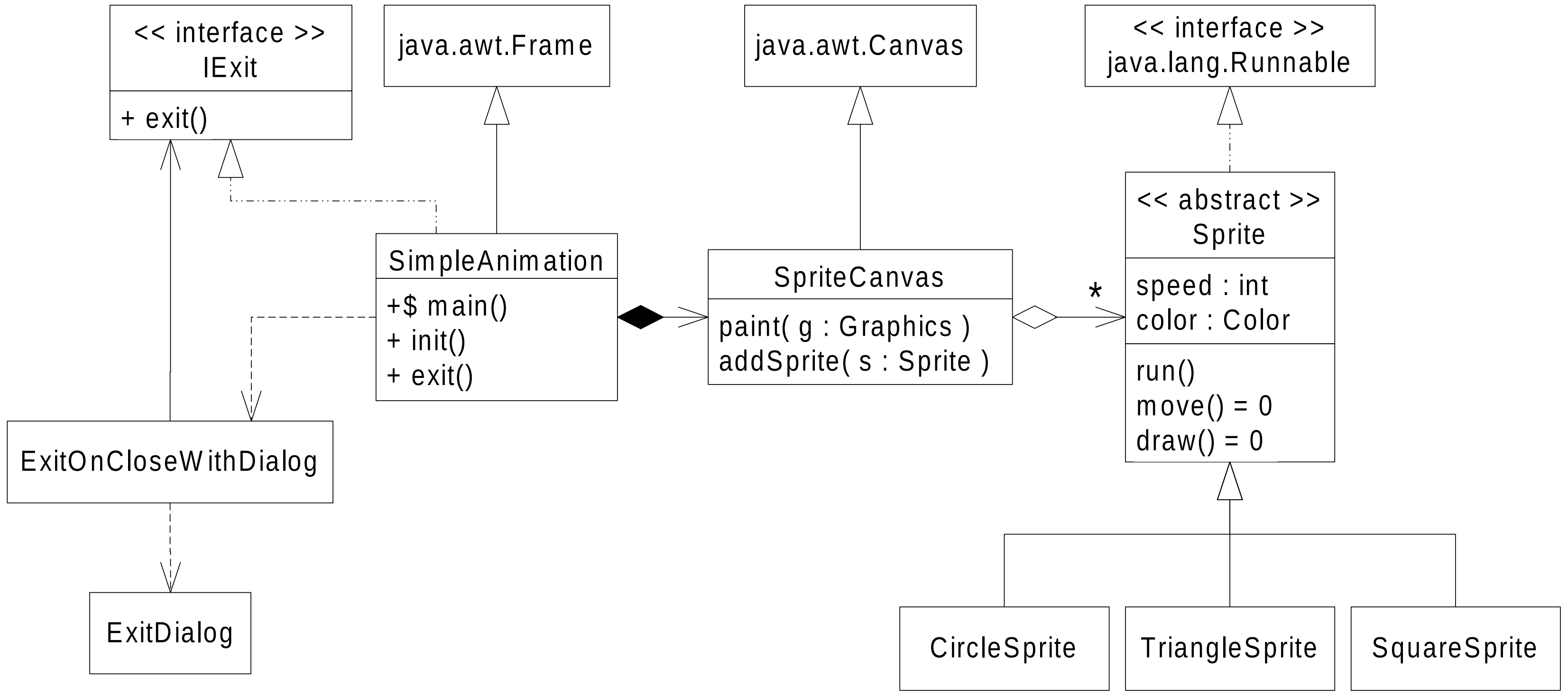
- Each Sprite sleeps for a specified time (a function of its speed), and then wakes up, moves, and *inefficiently* tells the entire SpriteCanvas to **`repaint()`**.
- Sprite's **`run()`** method is a *Template Method*.
- Notice the new helper class: **ExitOnCloseWithDialog**.





Do you really want to exit?

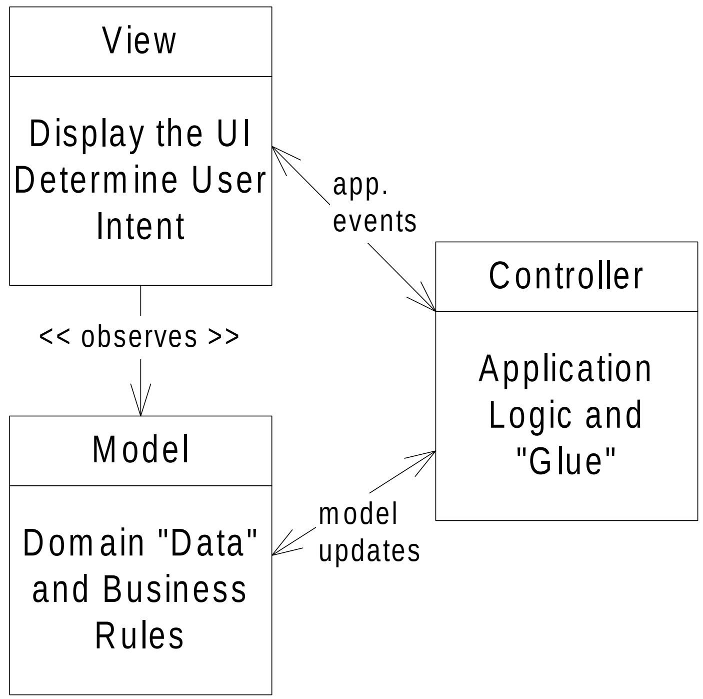
Example: Simple Animation (cont.)



Model-View-Controller

MVC is often used in conjunction with the Observer design pattern:

- Views might be *Observers* of Models ...



Model-View-Controller (cont.)

- The larger the design, the larger the gains achieved by ***separating the concerns*** of the application layers ala MVC.
- It might seem like it is duplicating effort to have controllers that do little more than delegate, but it is worth the effort!
- Views typically have layout code and event listeners, only. The listeners do nothing more than delegate to the controller. The view might also *observe* the model.
- More complicated GUIs might have a separate Model + View + Controller class per screen (Frame / Window / Tab / Page), and might have a hierarchical structure.
- The Controller and Model should be ignorant of view *technology* issues; in other words, the View *encapsulates* the display technology.
- The View should NOT have business logic (except perhaps field validation, which should also exist as a pre-condition to the Model).
- In distributed architectures, the Controller *encapsulates* the distributed middleware / messaging technology. Refer to Swing MVC.

MVC Example: The Sticks Game

- The Sticks Game **GUI** design can be simplified by having a View that knows nothing about how to play Sticks, and a Model that knows nothing about widgets.
- Furthermore, neither the HumanPlayer nor the ComputerPlayer classes (part of the *Model*) should have to listen for Events (*View* classes listen to Events).
- The Controller class is the *Mediator* (in this case, the Referee) collaborating with the (game + players) Model and (graphical) View.
- It is not necessary to name your classes Model, View & Controller, but it might be a good idea, nevertheless, to rename Referee to be SticksController, and to create a class named SticksView.

The Sticks Game (cont.)

Describe in detail the design changes that will be required to convert the text-based Java “console” Sticks Game application to an event-driven GUI application…

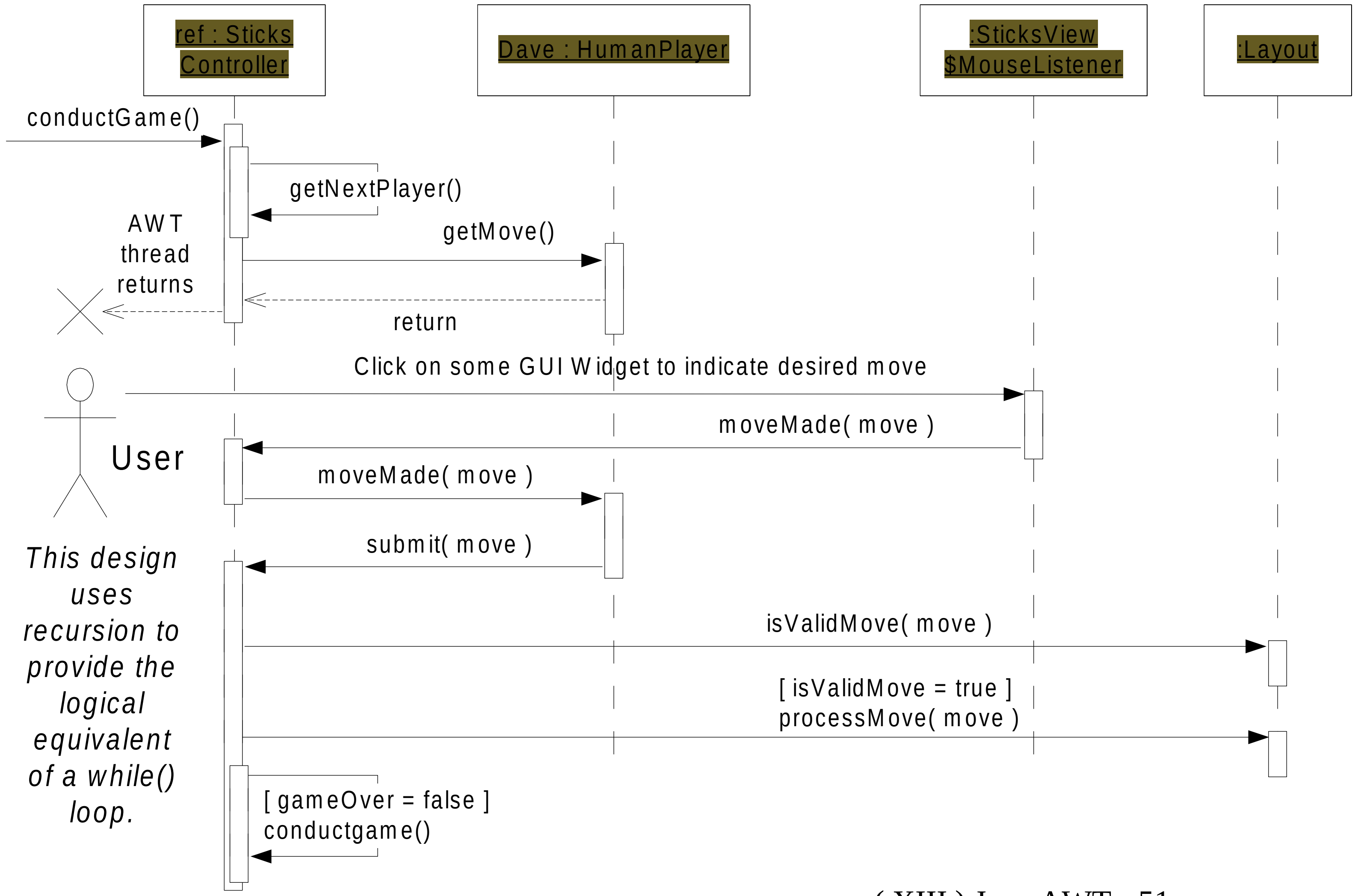
- Obviously, the dependency on the text-based Java Console must be refactored. A new graphical user interface must be designed. There must be a text entry widget for the human players’ names, and a numeric choice widget for the computer players’ search depth. There also must be a whole new “view” for displaying the board. Also a mouse click must be able to make a move. This implies the need for object(s) that listen for mouse click events.
- The biggest conceptual change is the new control flow, since the old main’s while(! gameOver()) control loop must be refactored. To detect human player mouse clicks, it is necessary to use the ***asynchronous*** Java AWT event / paint dispatch thread. Therefore, the old ***synchronous*** control loop must be radically redesigned. The rest of the “model” code may be reused unchanged.

The Sticks Game (cont.)

More on the control loop redesign:

- The Referee (SticksController) is supposed to treat the HumanPlayer and the ComputerPlayer identically, but the two classes are fundamentally different in an important respect: now that we have an event-driven GUI, the HumanPlayer makes its moves ***asynchronously***, whereas the ComputerPlayer remains ***synchronous***. When the ComputerPlayer is told to make a move, it uses minimax to search for a move, and then submits the Move object to the Referee when done; however when the HumanPlayer is told to make a move, nothing happens until the human clicks a mouse.
- The problem has to do with Threads; the Java AWT uses a single thread to process events from a single event queue. Now, if the Referee were to be implemented with a simple while loop, until game-over, as with the console application, then that while loop would require a second, separate thread. This forces our GUI redesign to understand and work with the AWT event dispatching thread. The good news is that in this case, there is an easy-to-code (but a little tricky to understand) design solution for the new control loop that uses just the AWT thread (using recursion in place of the while loop)...

The Sticks Game (cont.)



This design uses recursion to provide the logical equivalent of a while() loop.

The Sticks Game (cont.)

```
class SticksController { //  Pseudo-Code . . .

  public void conductGame() {
    currentPlayer = getNextPlayer();
    currentPlayer.getMove(); // no longer returns a Move
  }

  protected void submitMove( Move m ) { // called by getMove() !
    if( layout.isValidMove( m ) ) {
      layout.processMove( m );
      playerNumber++; // use other player next time
    }
    // else invalid move user error, will retry . . .
    if( ! layout.gameOver() ) {
      conductGame(); // recurse
    }
  }
}
```

The Sticks Game (cont.)

Another example from the <u>Sticks Game</u>…

Requirement: Use a Choice widget for selecting the ComputerPlayer's difficulty level… If the user clicks the selection "Very Difficult" then somehow we must tell the ComputerPlayer to use the DifficultStrategy…

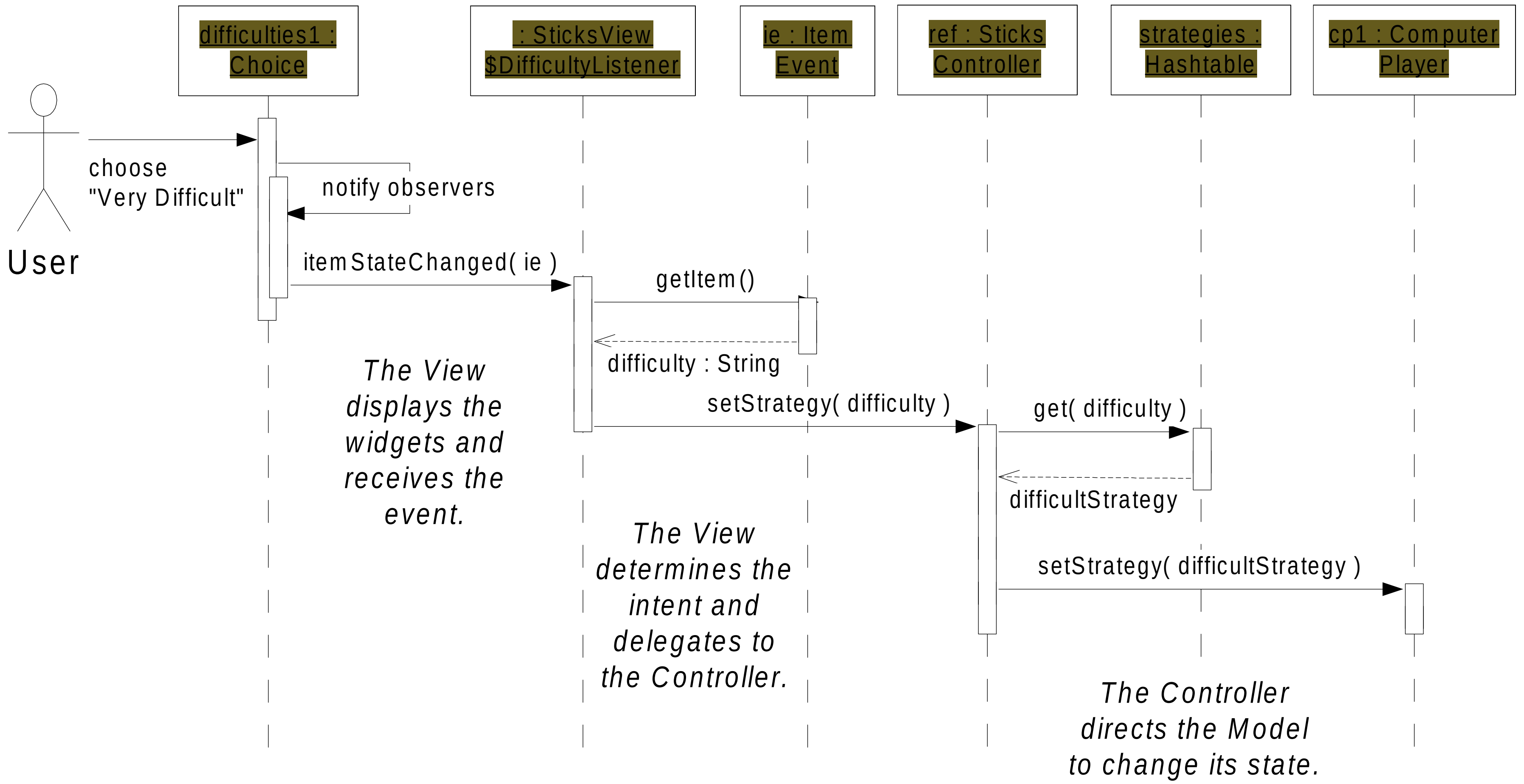
Use a HashMap to simplify the design, to map from Strings to Strategies:

- "Easy" **-->** EasyStrategy
- "Very Difficult" **-->** DifficultStrategy

Which class should be responsible for listening for the ItemStateChanged event?

- ComputerPlayer?
- Referee?
- SticksView$DifficultyListener?

The Sticks Game (cont.)


difficulties1 : Choice
: SticksView $DifficultyListener
ie : Item Event
ref : Sticks Controller
strategies : Hashtable
cp1 : Computer Player
User
choose "Very Difficult"
notify observers
itemStateChanged(ie)
getItem()
difficulty : String
setStrategy(difficulty)
get(difficulty)
difficultStrategy
setStrategy(difficultStrategy)
The View displays the widgets and receives the event.
The View determines the intent and delegates to the Controller.
The Controller directs the Model to change its state.

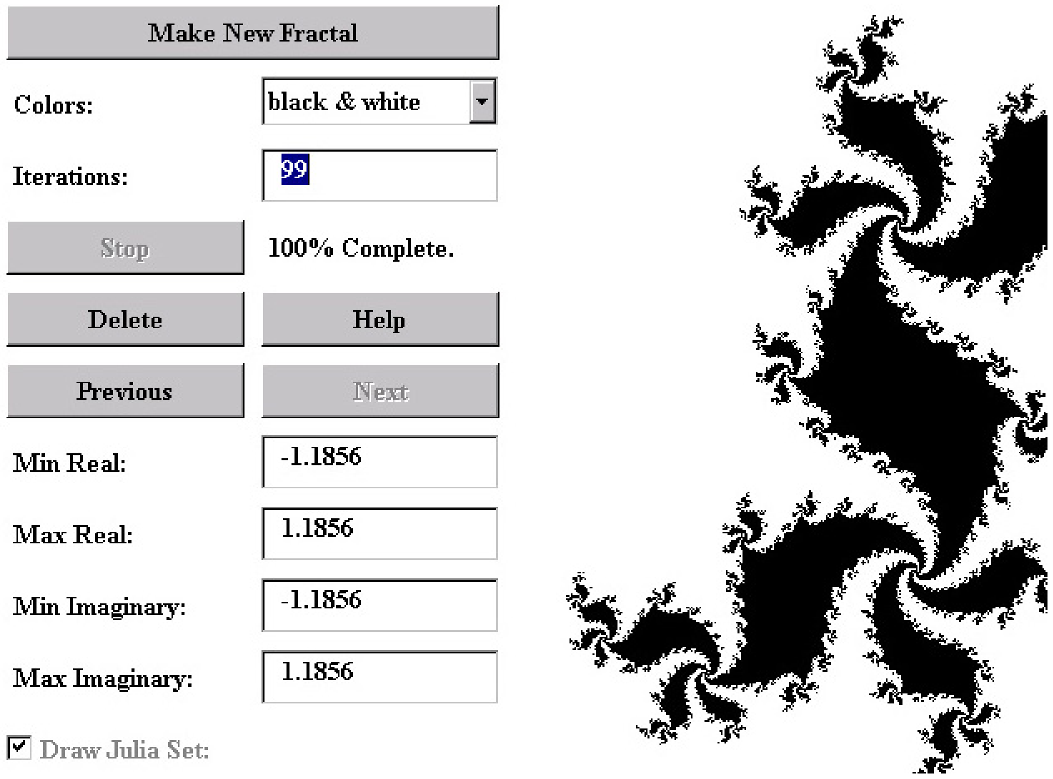
The Sticks Game (cont.)

Now imagine extending this design to be a ***Game Framework*** supporting "pluggable" games, configured with an XML file. In this case, there need only be one Controller (referee) shared by all games, since the application flow is the same for all games. There might also be shared view (all games might have the same basic layout and controls).

Consider having a FrameworkModel, and a FrameworkView, plus an abstract GameModel and a GameView that each game must implement.

This is an example of a ***hierarchical MVC*** design.

Example: The Fractal Applet



Colors: black & white

Iterations: 99

100% Complete.

Min Real: -1.1856

Max Real: 1.1856

Min Imaginary: -1.1856

Max Imaginary: 1.1856

[✓] Draw Julia Set:

The Fractal Applet (cont.)

- UML models and ***complete Java source code*** can be found on-line:

`https://www.leberknight.com/fractal.html`

- The Fractal Explorer online is now written in JavaScript.
- The original Java source code is still up there too.
- The Fractal Applet uses the following Design Patterns:

Mediator - The Fractal class acts as the *Mediator* for the entire Fractal program.

Model View Controller – The Fractal class is the *Controller*; the Calculators and Drawings are the *Model*; the ControlPanel and the DrawingCanvas make up the *View*. This is not a "pure" MVC design (there is no class called "model"), but nevertheless, the essence (intent) of MVC is respected by the separation of concerns across the implementing classes.

Strategy - There are various FractalCalculators.

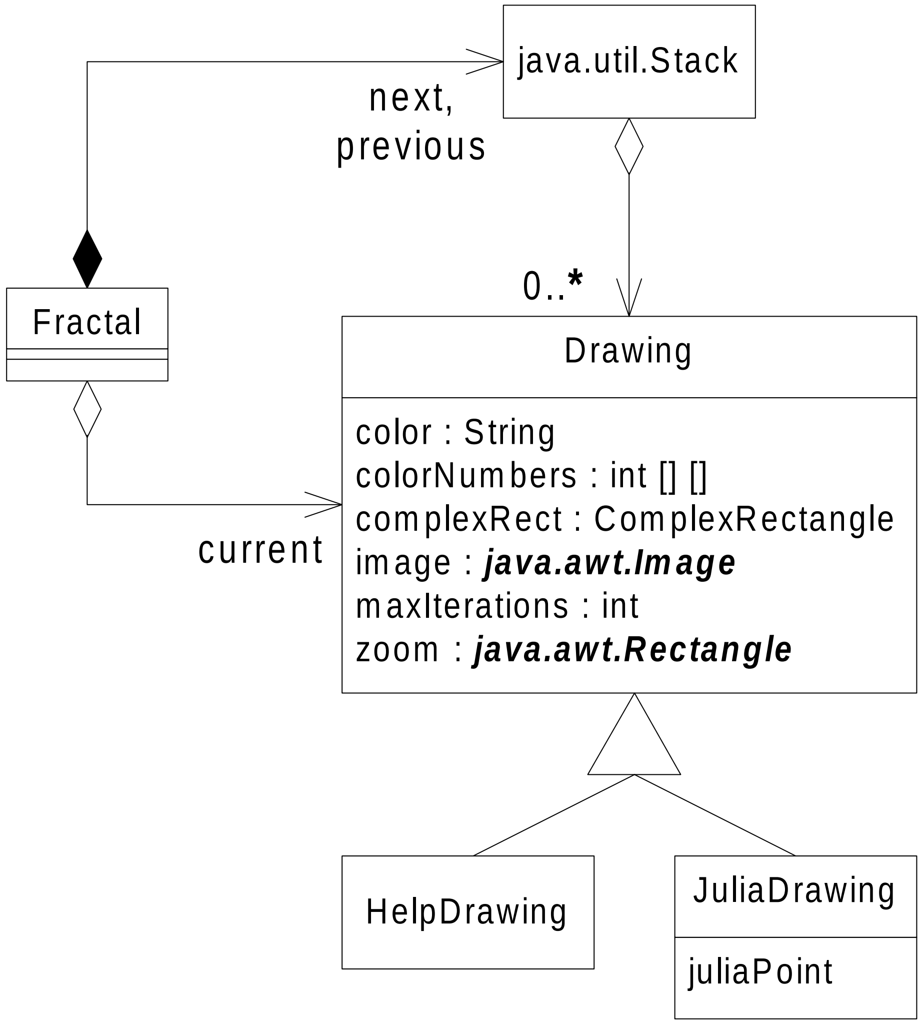
Template Method - FractalCalculator.getColor() calls the abstract method testPoint(), which is implemented by MandelbrotCalculator and JuliaCalculator.

The Fractal Applet (cont.)

More interesting details about the Java AWT - from the Fractal Applet:

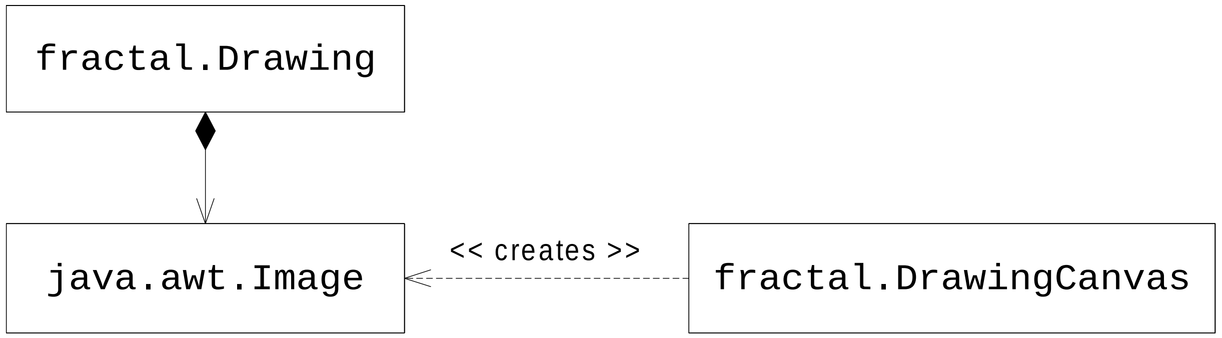
- The program may be *configured* with HTML or main() parameters.
- The Draw Julia Set `Checkbox` adds/removes `Components` from the `ControlPanel` layout *dynamically*. Refer to `ControlPanel.showJuliaControls()`, which calls `fractal.recalculateLayout()`.
- Sometimes an `Image` cannot be created by a `Canvas` until just before the first `paint()`is called from the AWT framework, in which case the AWT will return a ***`null`*** `Image`. This is why some initialization work is deferred until `Fractal.firstPaint()`.
- `DrawingCanvas` must override `getPreferredSize()`, `getMaximumSize()`, and `getMinimumSize()` because there is no `SetSize()` method for Canvas.
- `Drawing.finalize()` calls `image.flush()`.
- Thread synchronization is done at class Fractal (the controller).

Fractal Drawings



Double Buffering (in Fractal)

- The fractal applet uses the ***Double Buffering*** technique so that no long calculations ever need to be performed inside `paint()`, and also for the on-the-fly zoom rectangle graphics. Notice that `DrawingCanvas.paint()` does NOT calculate a fractal.
- The DrawingCanvas creates instances of `Image`, which then become part of a Drawing; this `Image` is the *Double Buffer*.
- Refer to `DrawingCanvas.mouseDragged()`... A `Graphics` object is obtained from the current `Image`, and used to draw zoom Rectangles (in `XOR` mode) “off screen.” Then the Canvas’ current `Graphics` object is obtained (the primary buffer), and used to `drawImage()` instantly. It is not necessary to call `repaint()`.



Swing

- The Swing Components are 100% pure Java, whereas the AWT components delegate to native “peers” which vary from platform to platform.
- In Java 1.1.x, the Swing components are available in the javax.swing package, but this code does not come built in to most Internet browsers. Thus, Swing is not generally available to Applets for the Internet at large (without requiring that users download the appropriate plug-in to upgrade their browsers). However, when running a Swing-based application, it is easy to put the javax.swing package in your classpath (often released in a jar file: **`rt.jar`**).
- In Java 1.2.x ++, the Swing components are generally bundled with the JDK.
- All of the Swing classes are subclasses of javax.swing.JComponent, which is a subclass of java.awt.Component.
- Swing components support various “Pluggable Look And Feels” (PLAF) such as Metal, Windows, Motif, and Macintosh, using a *Strategy* design.
- Swing widgets are NOT thread safe. All code that interacts with Swing should use the event queue. Use: `SwingUtilities.invokeLater( runnable )`.

Swing (cont.)

- Many Swing components behave just like AWT “replaced” components, so if you know the AWT, it is easy to get up to speed with Swing.
- Swing provides some 40 different components (4 times more than the AWT), with literally hundreds of other classes, so it takes a while to fully master.

java.awt.Button	… javax.swing.JButton
java.awt.Canvas	… javax.swing.JPanel
java.awt.Checkbox	… javax.swing.JCheckBox
java.awt.Label	… javax.swing.JLabel
java.awt.Choice	… javax.swing.JList
java.awt.Panel	… javax.swing.JPanel
java.awt.TextField	… javax.swing.JTextField

...

Swing (cont.)

The internal design of Swing uses a variation of MVC.

- The separation of the view from the model is essential for the "pluggable look and feel" design.
- Swing widgets are independent of any application, and so the controller has been combined with the view into what is called a "UI delegate." The base class for such UI delegates is javax.swing.plaf.ComponentUI.
- For example, the class javax.swing.JButton has a reference to a javax.swing.ButtonModel and a javax.swing.plaf.ButtonUI.
- ButtonModel is an interface; Swing provides a DefaultButtonModel.
- ButtonUI is an abstract class; an example concrete implementation is javax.swing.plaf.metal.ButtonUI.

Google the keywords: **javax.swing** *and* **javadoc**

And again, you will need to learn JavaScript to code GUIs in the wild.