SCHULICH School of Engineering

ENSF 593/594

14 – Java Graphical User Interface (Java GUI) – Part II

Event Handling In Java

Different Approaches to Event-Handling

- Two approaches to event handling
  - read-evaluation loop (client-written loop)
  - notification-based (callbacks)
- Swing uses the 2nd approach

Swing Events

- objects communicate by "firing" and "handling events (event objects)
- events are sent from a single source object to one or more registered listener objects
- different event sources produce different kinds of events
  - e.g., a *JButton* object, when clicked, generates an *ActionEvent* object, which is handled by an *ActionListener* (an object whose class implements this interface)

Overview of Event Handling

- In Java there are three type of classes that are involved in event handling:
  - **Event source classes**: such as: Buttons, TextFields, ScrollBars, etc.
  - **Class Event**: an class that encapsulates the event information.
  - **Event Listeners**: that receives an event object from an event source and provides an appropriate action or response.

What is an Event?

- A event is an object that represents some **occurrence** which we may want to **handle**.
  - Often generated by user actions: **keyboard, mouse, etc**.
  - Can also can be generated by other programs.
  - Different types of events are represented by different classes:
    - **`MouseEvent`**,
    - **`WindowEvent`**
    - **`ActionEvent`**,
    - `etc.`

What is a Listener?

- A listener is an object that **waits for** and **responds to** events.
  - There are different types of listeners represented by different listener interfaces: `MouseListener`, `ActionListener`, etc.
  - A listener class should implement **one of the listener interfaces.**

How Events are Handled?

- A listener object is an instance of class that ***implements*** an **interface** called a ***listener interface***.
- An ***event source*** must be **registered** to a ***listener object***.
- The **event source** sends out event object to **all registered listeners** when an event happens.
- The **listener objects** will use the information **events encapsulated in the event** to determine their reaction to the event.

Registering with the Listener Object

1. Create an ActionListener Object:

```
ActionListener mylistener = new …
```

2. Create the event source (e.g. a Button):

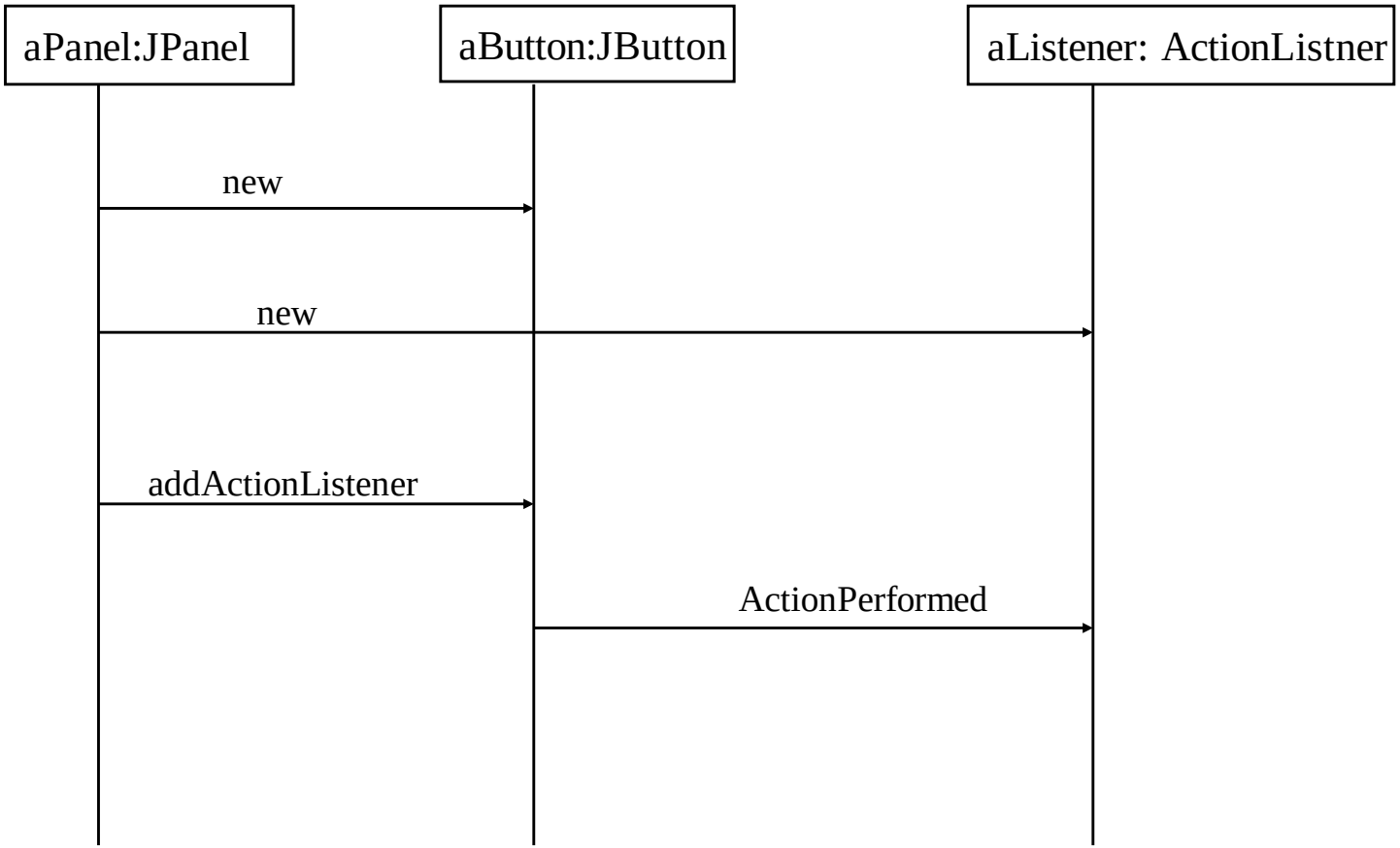
```
JButton myButton = new …
```

3. Register the event source(e.g. Button) with the listener object

```
myButton.addActionListener(mylistener);
```

Now the **listener object** will be **notified** when the button is clicked.

Event Notification



Useful Listeners and Methods

Interface	*Method(s)*
ActionListener	actionPerformed(ActionEvent e)
MouseListener	mouseEntered(MouseEvent e)
	mouseExited(MouseEvent e)
	mousePressed(MouseEvent e)
	mouseReleased(MouseEvent e)
WindowListener	windowActivated(WindowEvent e)
	windowClosed(WindowEvent e)
	windowClosing(WindowEvent e)
	windowDeactivated(WindowEvent e)
	windowDeiconified(WindowEvent e)
	windowIconified(WindowEvent e)
	windowOpened(WindowEvent e)
MouseMotionListener	mouseDragged(MouseEvent e)
	mouseMoved(MouseEvent e)
KeyListener	keyPressed(KeyEvent e)
	keyReleased(KeyEvent e)
	keyTyped(KeyEvent e)

- For Complete List of Listeners and their methods visit: http://docs.oracle.com/javase/tutorial/uiswing/events/api.html

Different Ways of Implementing Event Listeners

- Options for implementing listeners:
  1. listener class
  2. anonymous inner classes
  3. named inner classes

1. Listener Class

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

public class MyFrame extends JFrame {
        JButton b1, b2;
        MyListener linstener;

        public MyFrame(String s){
                super(s);
                linstener = new MyListener(this);
                setLayout ( new BorderLayout());
                b1 = new JButton("Submit");
                b2 = new JButton("Cancel");
                b1.addActionListener(linstener);
                b2.addActionListener(linstener);
                        add("North",b1);
                add("Center", b2);
         }
 //The main function goes here
}
```

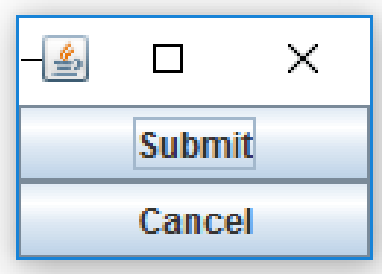
Now, Create a Class that Listens to the Actions

Create an ActionListener Class

```
import java.awt.event.*;

class MyListener implements ActionListener {
        private MyFrame frame;

        // constructor
        public MyListener(MyFrame jf) {
                frame = jf;
        }

        // performs an action in response to the event
        public void actionPerformed(ActionEvent e) {
                if (e.getSource() == frame.b1) {
                        System.out.println("Do Something");
                } else if (e.getSource() == frame.b2) {
                        System.out.println("Do Something Else");
                }
        }
}
```

Now Instantiating the Frame

```
public static void main(String args[])
{
    // Create the frame
    MyFrame frame = new MyFrame("Frame 1");
    frame.pack();
    frame.setVisible(true);
}
```



- The **pack** method sizes the frame so that all its contents are at or above their preferred sizes.
  - Alternative way is to establish a frame size explicitly by calling **setSize** or **setBounds** (which also sets the frame location).
- Using pack is preferable since pack leaves the frame layout manager in charge of the frame size.
- Layout managers will adjust it based on platform

ActionListener as an Inner Class

- The problem can be also solved by having the listener as an inner class of your Frame.
- Example:

ActionListener as an Inner Class

```
public class MyFrame extends JFrame {

    class   MyActionListener  implement ActionListener{
     // constructor
      public MyActionListener (…) {
        …
      }

      // performs an action in response to the event
      public void actionPerformed (ActionEvent event) {
      …
      }
    }
}
```

ActionListener as an Inner Class

```
public class MyFrame extends JFrame{
    JButton b1, b2;
    MyListener linstener;
    class MyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(e.getSource() == b1)
                doSomething....;
            else if        (e.getSource() == b1)
                doSomethingElse....;
        }
    }      // END OF INNER CLASS

    public MyFrame(String s){
        super(s);
        linstener = new MyListener();
        // REST OF THE CLASS AS PRIVIOUS SLIDE GOES HERE

    }      // END OF MyFrame CONSTRUTOR
}          // END OF CLASS MyFrame
```

Another Way: To Create an Anonymous Listener Object

```
myFunction ( new myAnonymousClass( )
{
    myAnonymousMemberFuction ( )
    {
        …
        …
    }
}
);
```

Example

- *Next step is to add an actionListener object to your Frame:*

```
public class MyJFrame extends JFrame {
public MyJFrame () {
   …
   // the object aButton calls the method addActionListener
   aButton.addActionListener (new ActionListener()
   {
      public void actionPerformed(ActionEvent evt)
     {
       JOptionPane.showMessageDialog (null, "you pushed the button .", "My  Message",
       JOptionPane.PLAIN_MESSAGE);
     } // end of actionPerformed
    }   // end of anonymous class and instantiation of an ActionListener object
  );    // end of call to addActionListener

} // end of constructor MyJFrame

    … // other method
} // end of Frame
```

Closer Look at the Anonymous Listener

```
Line 1.     aButton.addActionListener(
Line 2.       new ActionListener() {
Line 3.         public void actionPerformed(ActionEvent evt) {

Line 4.             // DO SOMETHING

Line 5.         }

Line 6.       }
Line 7      );
```

- Line 1, is call to function addActionLinsitner that creates an object of an anonymous class as its parameter.
- Line 2 to 6 is definition of an anonymous class.
- Line 3 is definition of a function within this anonymous class. This function receives an object of class ActionEvent as its parameter.
- Line 7 is the closing brace of call to addActionLinstener

Closer Look at the ActionEvent Class

- A semantic event which indicates that a component-defined action occurred.
- Some of the useful methods of this class includes:

```
public long getWhen();
// Returns the timestamp of when this event occurred.
public Object getSource();
// The object on which the Event initially occurred.
```

Closing a Frame

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);  // set initially to do nothing
addWindowListener(new java.awt.event.WindowAdapter() {
  public void windowClosing(WindowEvent e)
  {
    JFrame frame = (JFrame)e.getSource();
    int result = JOptionPane.showConfirmDialog(frame,
                                "Are you sure you want to exit the application?",
                                        "Exit Application", JOptionPane.YES_NO_OPTION);

    if (result == JOptionPane.YES_OPTION)
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
});
```

Mouse Events

Mouse Event Handling

- Mouse events notify when the user uses the mouse to interact with a component. It happens when:
  - cursor enters or exits a component's are
  - user presses or releases one of the mouse buttons.

MouseListener Interface

- **To handle mouse events your GUI classes need to implement one or more of the following interfaces:**
  - **MouseListener**
  - **MouseMotionListener**
  - **MouseWheelListener**
- **To implement this interfaces normally you should write an inner class, or an anonymous inner class.**

MouseListener Interface

- This interface has a few methods with a single argument, of type MouseEvent:
  - mousePressed
  - mouseClicked
  - mouseEntered
  - mouseExited
  - mouseReleased

.

MouseMotionListener Interface

- You need to implement the following methods that receive a single argument of type MouseEvent:
  - mouseMoved
  - mouseDragged
- Some of the useful methods of MouseEvent Class include:
  - int getClickCount
  - Point getPoint -- returns the position of event
  - int getX --returns the x-coordinate of the position
  - int getY

MouseWheelListener Interface

- The only method to implement for this interface is:
  - mouseWheelMoved
- This method receives one argument of type MouseWheelEvent

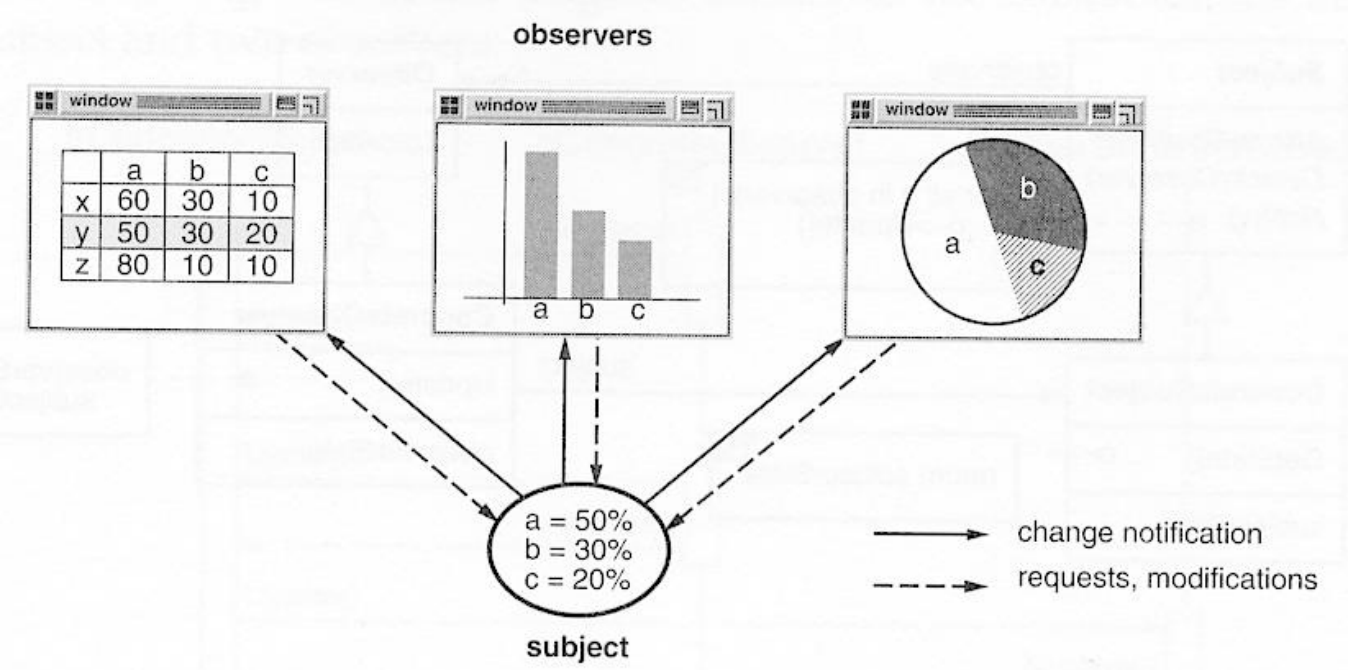
Part III

MVC Model

Model/View/Controller

- MVC roles:
  - Model
    - provides a number of services to manipulate the data e.g., recalculate, save
  - view
    - tracks what is needed for a particular perspective of the data
  - controller
    - gets input from the user, and uses appropriate information from the view to modify the model e.g., get slider value, trigger chart modify

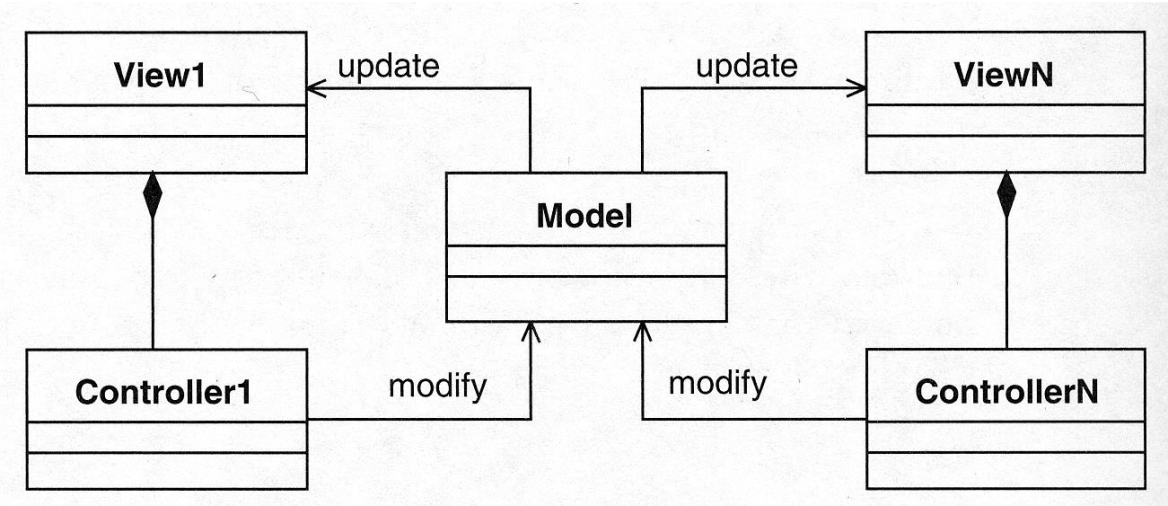
Dependencies



Dependencies

- Problems:
  - need to maintain consistency in the views (or observers)
  - need to update multiple views of the common data model (or subject)
  - need clear, separate responsibilities for presentation (look), interaction (feel), computation, persistence

Model/View/Controller



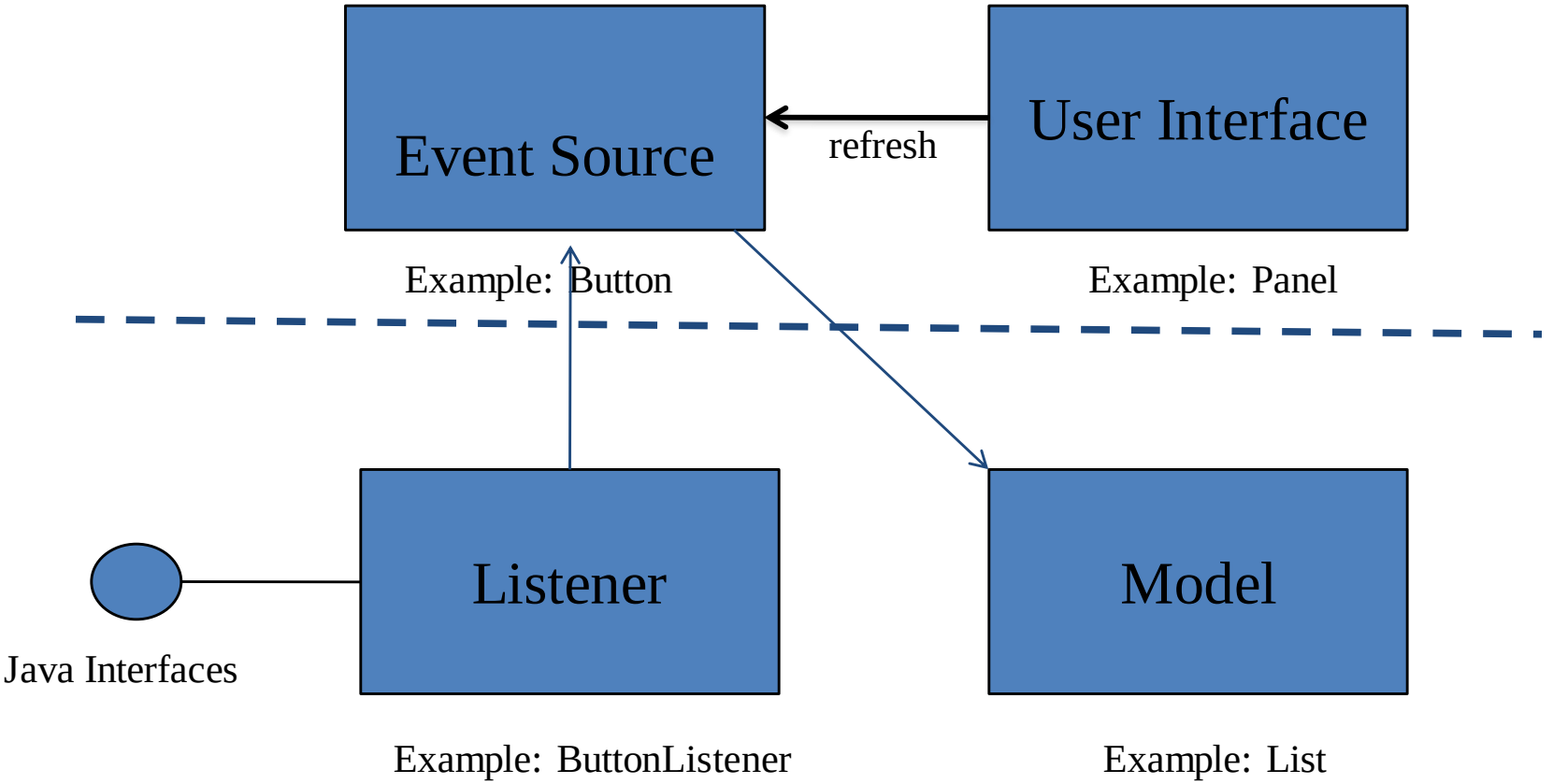
Model/View/Controller

- Separation:
  - you can modify or create views without affecting the underlying model
  - the model should not need to know about all the kinds of views and interaction styles available for it

Model/View/Controller

- In Swing:
  - in practice, views and controllers are implemented with Swing components and listeners
  - both views and controllers will be dependent on Swing APIs

Swing View/Model Architecture



Working with Shapes

Working with Shapes

- Starting Java 1.0 class Graphics was introduced.
- Java SE 1.2 introduced java 2D library
  - Enables you to :
    - Draw lines, rectangles and any other geometric shape.
    - Fill those shapes with solid colors or gradients and textures.
    - Draw text with options for fine control over the font and rendering process.
    - Draw images, optionally applying filtering operations.

Two kinds of painting

- **System-triggered Painting**
  - System requests a component to render its contents, usually for:
    - The component is first made visible on the screen.
    - The component is resize.
    - The component has damage that needs to be repaired.
      - Something that previously obscured the component has moved
      - Previously obscured portion of the component has become exposed).
- **App-triggered Painting**
  - the component decides it needs to update its contents
    - its internal state has changed.
    - button detects that a mouse button has been pressed.

The ‘paint’ Method

You can override the paint method to do simple drawing

- Overriding the paint method of the Frame is not advised
- Better way is to use the paintComponent of swing component, such as JComponent, Canvas, or JPanel.

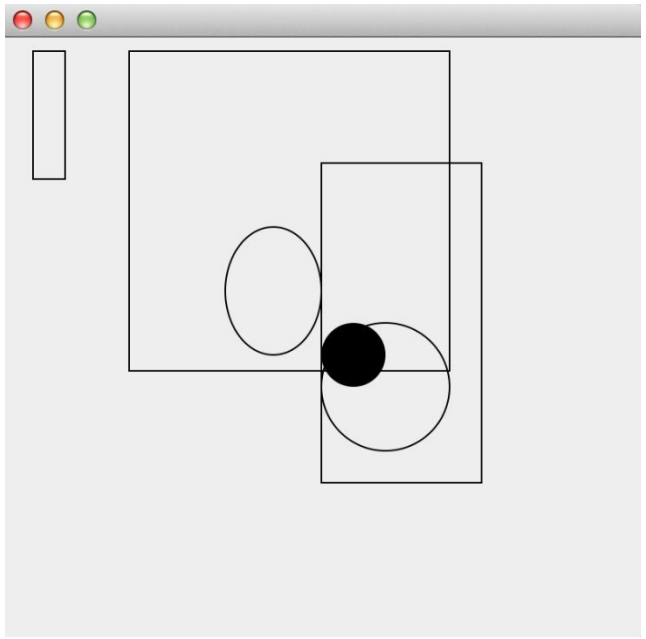
•When AWT invokes this method, the Graphics object parameter is pre-set to the properties of the component.

•It is not recommended that programs invoke paint() directly.

•A more powerful graphics object can be created by using ‘Grpahics2D’

Paint Method - Example

```
public void paint(Graphics g){
    Graphics2D g2 = (Graphics2D) g;
     // Dynamically calculate size information
     Dimension size = getSize();
     int d = Math.min(size.width, size.height);
     int x = d/2;
     int y = d/2;
     g2.setColor(Color.black);
     g2.fillOval(x, y, d/10, d/10);
     g2.drawOval(x, y, d/5, d/5);
     g2.drawOval(140, 140, 60, 80);
     g2.drawRect(80, 30, 200, 200);
     g2.drawRect(200, 100, 100, 200);
     Rectangle2D rect = new Rectangle2D.Double(20, 30, 20, 80);
     g2.draw(rect);
 }
```



"repaint" Method

- For app-triggered painting, you should use one of the possible form of repaint method:

  public void repaint()

  public void repaint(long tm)

  public void repaint(int x, int y, int width, int height)

  public void repaint(long tm, int x, int y, int width, int height)

  Where:

  - tm is the maximum time in millisecond to be used by the RepaintManager before update.
  - x and y are x and y coordinates of the area to be repainted
- A common mistake is to invoke the no-arg version when only a portion of the component needs repaint.

repaint Method - Example

```
MouseListener ml = new MouseAdapter()
{
      public void mousePressed(MouseEvent e) {
         JButton b = (JButton)e.getSource();
         b.setSelected(true);
         b.repaint();
      }

      public void mouseReleased(MouseEvent e) {
         JButton b = (JButton)e.getSource();
         b.setSelected(false);
         b.repaint();
          }
 };   // END OF ANONYMOUS INNER CLASS
```

Brief Introduction to Applet

What is an Applet?

- An applet is a program or application that is started within a web browser
- An applet program doesn't have a main function.
- An Applet must be derived from class java.Applet.
- An applet inherits several functions form its parent class (Applet or JApplet). Three of these functions are:
  - intit
  - stop
  - paint
- It may or may not override these functions.

Security Restrictions

- Not allowed to read from, or write to, the file system of the computer viewing the applets.
- Not allowed to run any programs on the browser's computer.
- Not allowed to establish connections between the user's computer and another computer except with the server where the applets are stored.

Example

```
import javax.Applet;
import java.awt.*;
public class HelloWorld extends JApplet  {
    public void init() {
    }

    public void stop() {
    }

    public void paint(Graphics g) {
     g.drawString("Hello World", 100, 100);
    }
}
```

Applet Methods

- **Method init**: This method is used to initialize the applet and its components. It will be automatically called when the applet is started.

- In the previous example we have an applet with no other components, and we have used the default layout of an applet. Therefore, the body of the method init is empty.

Applet Methods

- **Method stop**: This method will be called automatically when the user leaves the web page in a browser, or when the applet terminates.

Applet Methods

- **Method paint**: This method that uses an object of class Graphics as it's argument will be called every time that an applet is created or the web page is revisited.
- In the previous example the method darwString is used to display "Hello World" on the **applet panel**. Its second and third integer parameters indicate the x, and y coordinates of the starting point to display the string.

Applet Method

- Method darwString that belongs to class Graphics draws a string, “Hello World”, in this example on the applet panel. Its second and third integer parameters indicate the x, and y coordinates of the point to start displaying the string.

Applet Methods

- As you have noticed our applet class does not have a main function. In fact, init is the starting function of an applet program. If you don't override the init function, the inherited init from class Applet will be automatically called.

Running Applets

- You can run your applet from a browser or by using the command appletviewer, from command line.
- An applet must be called from an html file (Hyper Text Mark-up Language).

```
<HTML>
<APPLET code="HelloWorld.class" width=800 height=500>
</APPLET>
</HTML>
```

What is an HTML file?

- As you have noticed this HTML file contains html tags like: <HTML> , and each tag has another tag with a forward slash that indicates the end block of a tag. For example the tag <HTML> is confined between the <HTML> and </HTML>
- To create a HTML file you can use any text editor like emacs or windows notepad, and should be saved with the html extension. In this example the file name will be HelloWorld.html.

What is JApplet

- JApplet inherits from Applet
- Inherits methods init, stop and paint but cannot override them
- Like other classes can use a constructor to initialize its components...etc.
  - The Components like buttons, textarea, textbox goes in the "content pane".
  - To access content pane you should call getContentPane
  - To modify the content pane attributes you should call setContentPane.
  - Default layout manager is BorderLayout (not FlowLayout)
  - You get Java (Metal) look by default, so you have to explicitly switch if you want native look.
  - Do drawing in paintComponent, not paint.