

The Java Series

GUI Building with AWT

The `java.awt` package

Provides a set of classes to build user interfaces.

Window, Button, Textfield, etc..

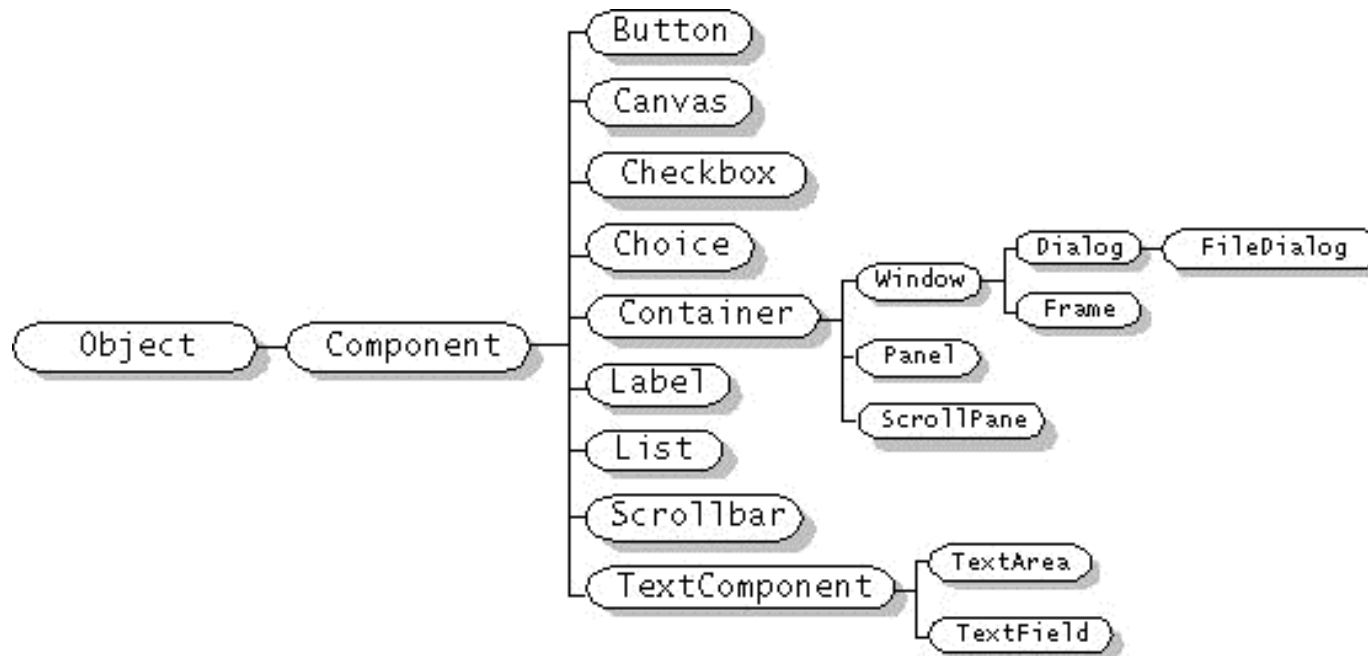
To build a UI we just instantiate objects from those classes:

We create windows.

Insert buttons into windows.

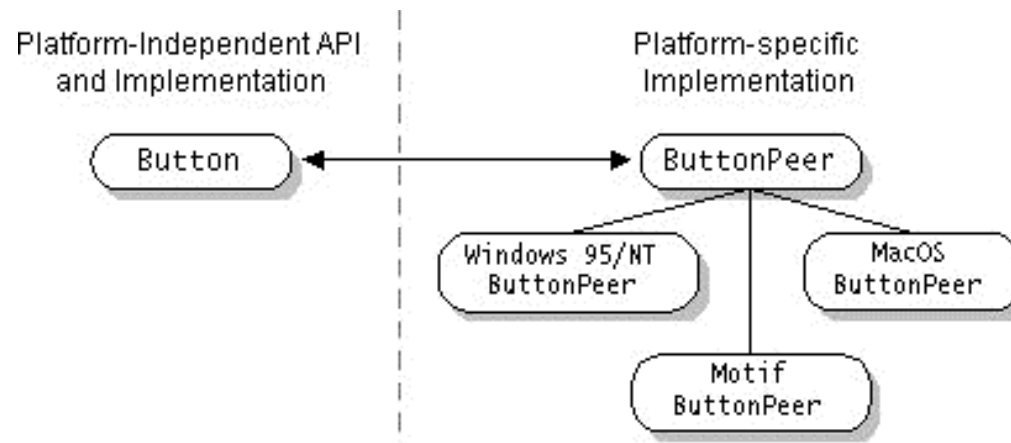
Read/set text from textfields, etc..

The AWT class hierarchy



Physical Graphical Elements

The java interpreter automatically creates the corresponding physical element when we create objects from AWT classes.



Each java interpreter knows how to do it in the platform it is running.

Physical Graphical Elements

When creating a graphical object:

The Windows interpreter makes Win API calls.

The Unix interpreter makes Motif calls.

The MacOS interpreter does MacOS calls.



But this is done at run-time, depending on which platform we run the java application:

The source Java code is always the same.

The Java bytecodes are always the same.

Other classes

AWT Also provides a set of classes to manage the graphical objects:

Events, layout managers, etc..

The combination of both is what let us build a UI and decide how it interacts with the user.

GUI Building

When making a GUI we have to implement two aspects:

- The positioning and distribution of a set of graphical elements.

- The interaction among those elements when things happen to them.

These are referred to:

- The LAYOUT definition.

- The EVENTS handling.

The Goal

The idea of this presentation is to explain the main **mechanisms** to put in place to build GUIs.

Detailed information about each element can be found in the doc.

We will also see how to search the doc.

Scenario 1

We want to create the following window



In terms of AWT elements there are:

- 1 Window
- 2 Buttons
- 1 Textfield

Sce 1: MyApplication class

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

We declare we are going to use AWT

```
public class MyApplication {
```

```
    public static void main (String args[]) {
```

```
        Frame f = new Frame("Hello");  
        f.setLayout (new FlowLayout());
```

We create a Frame object. The interpreter creates the physical windows with local OS calls

```
        Button    b1 = new Button("This is button 1");
```

```
        Button    b2 = new Button("This is button 2");
```

```
        TextField t1 = new TextField("Some Text");
```

```
        f.add(b1);
```

```
        f.add(b2);
```

```
        f.add(t1);
```

Create a few graphical objects.

```
        f.pack();
```

```
        f.setLocation(100,100);
```

```
        f.show();
```

Insert them into the frame

Position and show the frame

```
    }  
}
```

Scenario 1

Note that:

The last thing we do is to show the frame. When the program finishes the frame is not destroyed and works as expected.

The interpreter runs the interface concurrently with our program as soon as `f.show()` is executed.

The Component class

Any Graphical object is a Component.

It s the root of the whole hierarchy: Buttons, Frames, TextAreas, etc..

FROM THE REFERENCE DOCUMENTATION

<http://wwwinfo.cern.ch/support/java/docs/api>

```
public abstract class Component
extends Object
implements ImageObserver, MenuContainer, Serializable
```

A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

The Component Class

SOME METHODS FROM THE REFERENCE DOCUMENTATION

`void paint(Graphics g)`

Paints this component.

`protected void processEvent(AWTEvent e)`

Processes events occurring on this component.

`void setLocation(int x, int y)`

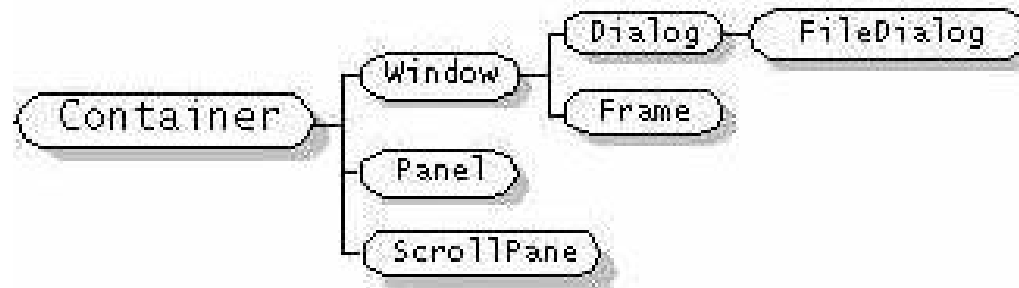
Moves this component to a new location.

`void setSize(int width, int height)`

Resizes this component so that it has width width and height.

The Container class

A Container is a Component able to hold other Components (including other Containers)



The Container class

Any container has:

- The list of Components it contains

- A Layout Manager in charge of distributing the components within.

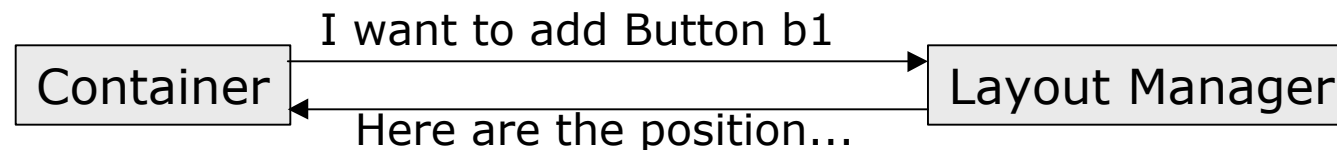
- Methods to add/remove components.

The Layout Manager class

Each Container object uses a LayoutManager object to calculate the positions and sizes of its Components.

Each LayoutManager object can only be used by one Container.

Different LayoutManagers use different algorithms to position the Components.



Layout Managers

There is defined set of LayoutManagers ready to use out of the box. As components are added they are positioned:

FlowLayout: left to right, top to bottom

BorderLayout: with respects to the borders of the container

GridLayout: Grid based positioning

GridBagLayout: Complex grid based positioning

Scenario 2: The GridLayout

```
import java.awt.*;

public class MyApplication {

    public static void main (String[] args) {
        Frame f = new Frame();
        f.setLayout (new GridLayout(2,2, 20, 20));
        f.setLocation(100,100);

        f.add (new Button("This is button 1"), new Dimension(1,1));
        f.add (new Button("This is button 2"), new Dimension(2,1));
        f.add (new TextField("This is a textfield"), new Dimension(2,2));
        f.pack();
        f.show();
    }
}
```

We create and associate a LayoutManager with the Frame we are defining. It's a 2x2 grid, with 20 pixels of spacing

Now, when adding Components, we have to specify where in the Grid they should be placed

Scenario 3: The Border Layout

```
import java.awt.*;

public class MyApplication {

    public static void main (String[] args) {
        Frame f = new Frame();
        f.setLayout (new BorderLayout());
        f.setLocation(100,100);

        f.add ("North", new Button("North"));
        f.add ("West", new Button("West"));
        f.add ("South", new Button("South"));
        f.add ("Center", new Button("Center"));
        f.pack();
        f.show();
    }
}
```

We create and associate a `LayoutManager` with the `Frame` we are defining. The `BorderLayout` manager does not require initial arguments when creating it.

Now, when adding `Components`, we have to specify to what part of the available space in the container they should occupy.

Containers and Components

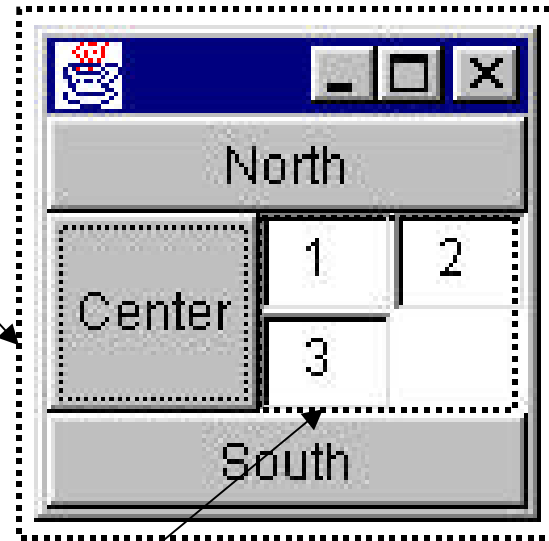
Containers are also regular Components (derived from the Component class).

Since a Container contains a set of Components it can also contain other Containers.

Then we have to decide what Layout Manager we what to use with EACH container.

Scenario 4

We have a Frame using the BorderLayout



We have a Panel occupying a position within the Frame as any other Component. This Panel uses a GridLayout

Scenario 4

```
import java.awt.*;
public class MyApplication {
    public static void main (String[] args) {
        Panel p = new Panel(new GridLayout(2,2));
        p.add (new TextField("1"), new Dimension(1,1));
        p.add (new TextField("2"), new Dimension(1,2));
        p.add (new TextField("3"), new Dimension(2,2));

        Frame f = new Frame();
        f.setLayout (new BorderLayout());
        f.setLocation(100,100);

        f.add ("North", new Button("North"));
        f.add ("West", new Button("Center"));
        f.add ("South", new Button("South"));
        f.add ("Center", p);
        f.pack();
        f.show();
    }
}
```

We create a Panel, associate a GridLayout on the fly, and add some components

We add the Panel (p) in the Frame as any other component

Other Components

There are more components available in AWT:

Buttons

TextFields

Labels: Just plain text.

Choices: Drop down lists.

Lists: Lists with selectable items.

Checkboxes: Clickable radio buttons.

TextAreas: Multi line text input.

Canvases: To draw arbitrary shapes & pictures.

Each component class has its particular way to create it, give it information, etc..

Scenario 5: Other Components

```
import java.awt.*;

public class MyApplication {

    public static void main (String[] args) {

        Choice l = new Choice();
        l.addItem("Item 1");
        l.addItem("Item 2");
        l.addItem("Item 3");

        TextArea ta = new TextArea(5,20);
        TextField tf = new TextField();
        Label lb = new Label("This is an example");

        Frame f = new Frame("Some sample");
        f.setLayout(new GridLayout(2,2));
        f.setLocation(100,100);
        f.add (lb, new Dimension(1,1));
        f.add (l,  new Dimension(1,2));
        f.add (tf, new Dimension(2,1));
        f.add (ta, new Dimension(2,2));

        f.pack();
        f.show();
    }
}
```

A few components. Each component has its own methods, constructors, etc.. specific to the function they perform.

See API documentation for details on each different component

Events

Once the GUI Layout is done we have to define what to do when the user interacts.

GUI Programming is EVENT DRIVEN

Events are things happening on the program:

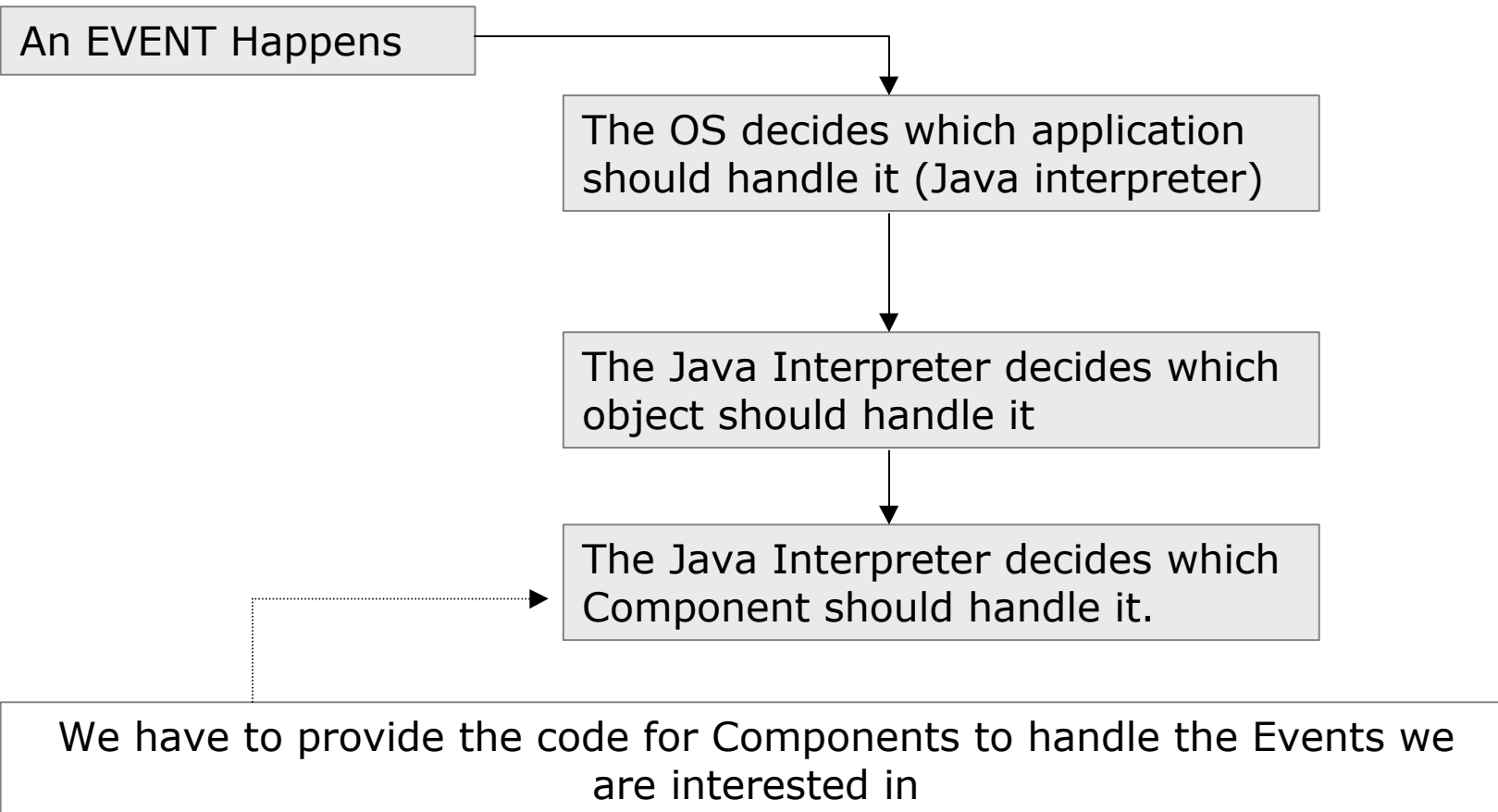
- A mouse move or click.

- A key pressed.

- A focus gained.

- A list item selected.

Events



Events

To handle Events we write objects and methods which will be invoked whenever a certain event happens.

This is very different from sequential programming.

The philosophy is that we set things up to be ready whenever an event happens.

The Java interpreter will know what method to invoke whenever an event happens.

Types of Events

There are two types of Events to handle

Primitive Events: Mouse, Key,

Semantic Events: Component-dependant
according to the function of the component:

- List Item Selected

- Text Field changed

- etc...

Components and Events

The Java interpreter passes an event to the relevant component.

The component contains a list of Listener objects.

The component notifies each registered listener object by invoking an specific method in each of them.

Each component contains a list of registered Listeners for each type of event.

Handling Events

1. Create a class defining the methods to be invoked upon receiving a certain event.
2. Instantiate an object from that class.
3. Register the object with the component you want to handle the event.
4. Wait for the event to happen.

Scenario 6

We are going to handle a semantic event:

CLICKING ON A BUTTON

We are going to be handling an Action Event.
See the API doc for the **Button** AWT class
and for the **ActionListener** interface.

Sce 6: ActionListener class

This is our own class. We don't derive from anything. Listener classes are not graphical elements themselves

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

public class MyActionListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        System.out.println("A button has been pressed");
    }
}
```

We just implement the ActionListener interface

Following the definition of the ActionListener interface the actionPerformed method is invoked whenever an instance of this class is properly registered

Sce 6: MyApplication

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

```
public class MyApplication {
```

```
    public static void main (String[] args) {
```

```
        Button b = new Button ("Press me");  
        MyActionListener alistener = new MyActionListener();  
        b.addActionListener(alistener);
```

```
        Frame f = new Frame("Some sample");  
        f.setLayout(new FlowLayout());  
        f.setLocation(100,100);  
        f.add (b);
```

```
        f.pack();  
        f.show();
```

```
    }  
}
```

We create a regular Button

We create the Listener object

We register the Listener with the Button. Now AWT takes care of invoking its method whenever the event happens.

Scenario 7

Handling another semantic event:

SELECTING ITEMS FROM LISTS

The mechanism is the same:

- Create the appropriate class

- Instantiate an object

- Register it with the List component we want

See the API doc for AWT List

Sc 7: ItemListener

Our own class implementing an Interface

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

public class MyItemListener implements ItemListener {

    public void itemStateChanged(ItemEvent e) {
        System.out.println("An Item has been (de)selected");
    }
}
```

The method required by the interface

Sce 7: MyApplication

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

```
public class MyApplication {
```

```
    public static void main (String[] args) {
```

```
        List l = new List();
```

```
        l.addItem("Select here");
```

```
        l.addItem("...or here");
```

```
        l.addItem("...or even here");
```

```
        MyItemListener alistener = new MyItemListener();
```

```
        l.addItemListener(alistener);
```

```
        Frame f = new Frame("Some sample");
```

```
        f.setLayout(new FlowLayout());
```

```
        f.setLocation(100,100);
```

```
        f.add (l);
```

```
        f.pack();
```

```
        f.show();
```

```
    }  
}
```

The List component



The Listener object

The registration

Scenario 8

Retrieving information from an Event.

Whenever a Listener method is invoked, an Event object is passed as parameter.

The Event object contains information concerning the specific event:

- The X,Y of mouse if a mouse event.

- The item selected if an item event.

- Etc (see API doc)

Sc 8: ItemListener

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

```
public class MyItemListener implements ItemListener {
```

```
    public void itemStateChanged(ItemEvent e) {  
        Integer item = (Integer)(e.getItem());  
        System.out.println( "Item "+item.intValue()+  
            " has been (de)selected");
```

```
        System.out.println();
```

```
        List l = (List)(e.getItemSelectable());  
        System.out.println ("Selected Item has text "+  
            l.getItem(item.intValue()));
```

```
    }  
}
```

The Event

We can retrieve the index of the selected item

Or the list component generating the event

Once we have the list, we can retrieve the text of the items

Scenario 9

We are going to add listeners for other types of events to Scenario 6.

Now we want to handle Primitive events on the button:

Mouse Enter.

See the API doc for AWT Button and then Component

Sc 9: MouseListener


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

We implement this interface



```
public class MyMouseListener implements MouseListener {  
  
    public void mouseClicked(MouseEvent e) {}  
    public void mouseEntered(MouseEvent e) {  
        System.out.println("Mouse has entered this object");  
    }  
    public void mousePressed(MouseEvent e) {}  
    public void mouseReleased(MouseEvent e) {}  
    public void mouseExited(MouseEvent e) {}  
  
}
```

As required by the interface we **HAVE** to implement all these methods even though we are only interested in one



Sc 9: MyApplication

```
import java.awt.*;

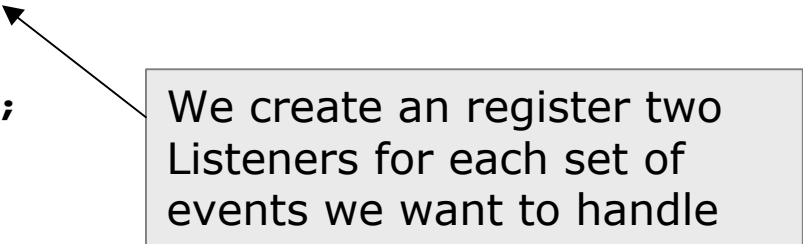
public class MyApplication {

    public static void main (String[] args) {

        Button b = new Button ("Press me");
        MyActionListener alistener = new MyActionListener();
        MyMouseListener mlistener = new MyMouseListener();
        b.addActionListener(alistener);
        b.addMouseListener(mlistener);

        Frame f = new Frame("Some sample");
        f.setLayout(new FlowLayout());
        f.setLocation(100,100);
        f.add (b);

        f.pack();
        f.show();
    }
}
```



We create an register two Listeners for each set of events we want to handle

Scenario 10

It would be desirable to define only the methods we are interested in.

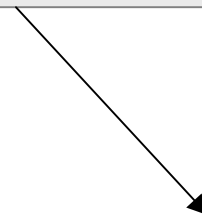
AWT contains adapters which provide empty implementations for event handler.

Our Listeners can be derived from them redefining only the methods we are interested in.

THIS IS OBJECT ORIENTATION !!!

Sce10: MouseListener

We extend the MouseAdapter AWT class which already provides empty implementations for a MouseListener



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

public class MyMouseListener extends MouseAdapter {

    public void mouseEntered(MouseEvent e) {
        System.out.println("Mouse has entered this object");
    }
}
```

The rest of the code (MyApplication) remains unchanged

Scenario 11

This is OO!! We can extend the AWT class hierarchy redefining methods as desired.

For instance: We want to create an OKButton class which always has the OK text in it, and we can optionally pass the action listener directly when we create OKButtons.

Sce11: OKButton

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

We inherit from the AWT Button class



```
public class OKButton extends Button {
```

```
    // A constructor which automatically provides the text
```

```
    public OKButton() {  
        super("OK");  
    }
```

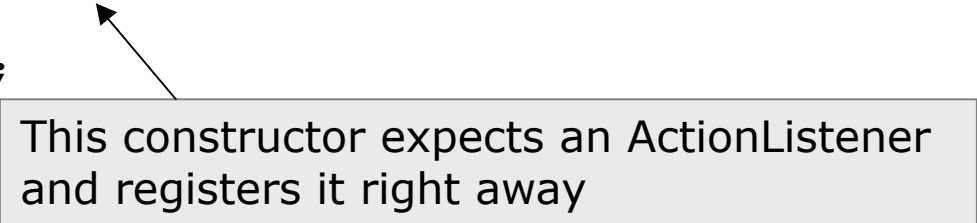
And define two constructors



```
    // A constructor to register the action listener  
    // at instantiation time
```

```
    public OKButton(ActionListener al) {  
        this();  
        addActionListener(al);  
    }
```

This constructor expects an ActionListener and registers it right away



```
}
```

Sce 11: MyApplication

```
import java.awt.*;

public class MyApplication {

    public static void main (String[] args) {

        OKButton b1 = new OKButton ();
        OKButton b2 = new OKButton(new MyActionListener());

        Frame f = new Frame("Some sample");
        f.setLayout(new FlowLayout());
        f.setLocation(100,100);
        f.add (b1);
        f.add (b2);

        f.pack();
        f.show();
    }
}
```

We create two OKButtons



When creating the second OK Button we also:

- Create an ActionListener
- The constructor takes care of registering it

Conclusions

We use an already made hierarchy of classes defined in the `java.awt.*` packages.

GUI Building is a two fold task:

Components Layout + Event handling.

There are a few kinds of objects we have to combine:

Graphical Components

Containers + Layout Managers

Listeners + Events

And remember this is OO. You can create your own graphical classes, redefine the ones provided by AWT, etc

Java at CERN

JDK is available through ASIS and NICE

Not all platforms have the same versions available.

To see the versions of your platform: `java -listversions`

To work with an specific version: `setenv JDKVERSION 1.2`

wwwinfo.cern.ch/support/java

JAVA SUPPORT at CERN

Any question about java should be posted to the **cern.java** newsgroup, jdk maintainers and other java users exchange problems and ideas there.

There is an associated mailing list for passive reading (cern-java@listbox.cern.ch)