ITI 1121. Introduction to Computing II [*]

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of February 7, 2013

Abstract

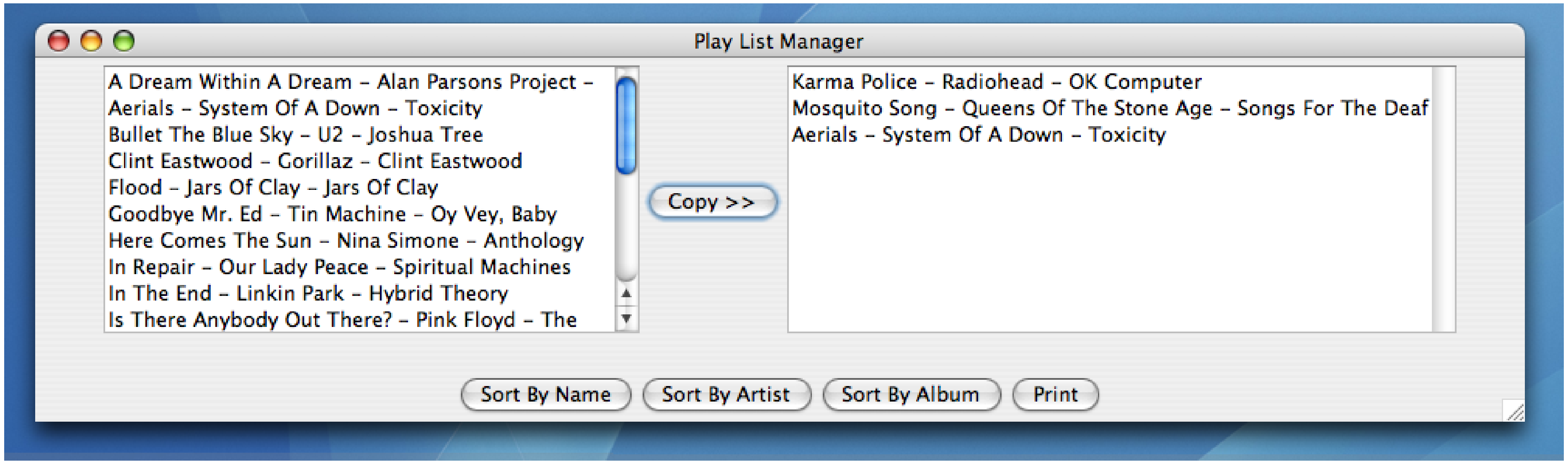
- Graphical components
- **Event-driven programming**

[*]These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

AWT

The *Abstract Window Toolkit* (AWT) is the oldest set of classes used to build Graphical User Interfaces (GUIs) in Java. It has been part of all the Java releases.

A more recent and improved toolkit is called **Swing**.



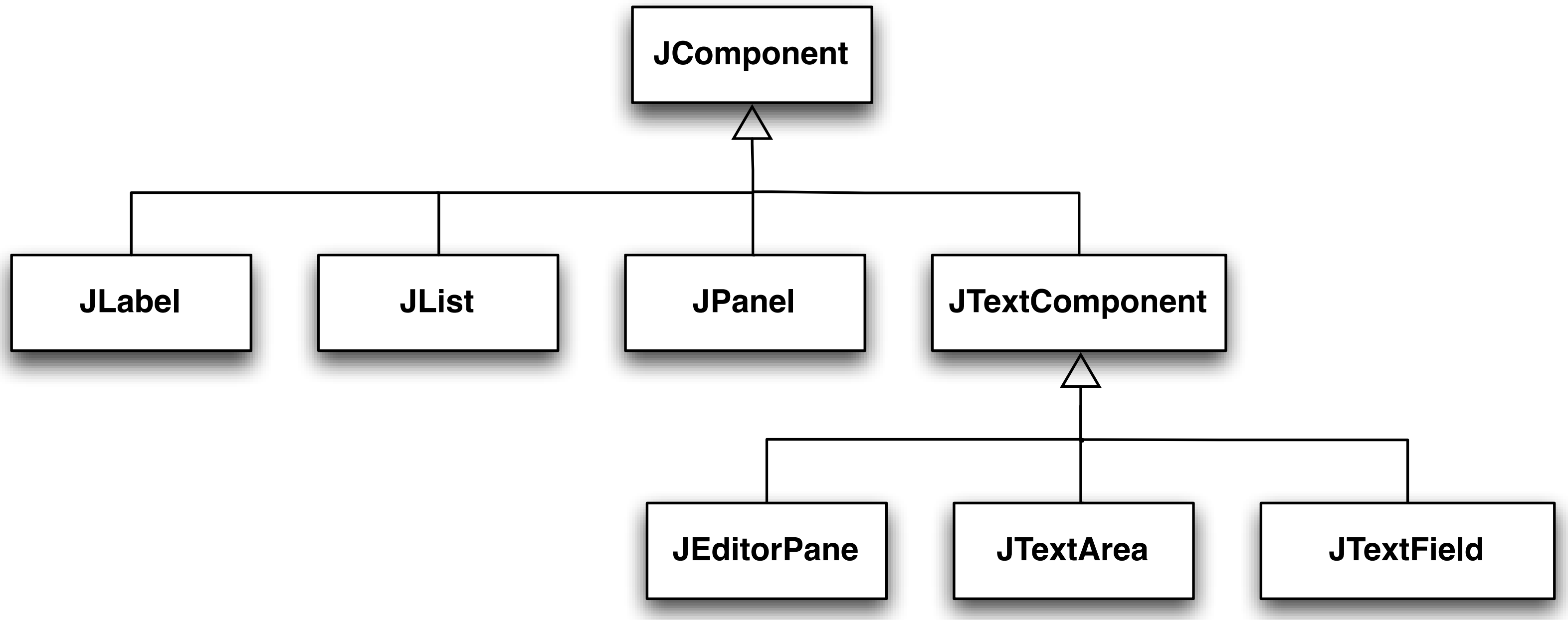
A Dream Within A Dream – Alan Parsons Project –
Aerials – System Of A Down – Toxicity
Bullet The Blue Sky – U2 – Joshua Tree
Clint Eastwood – Gorillaz – Clint Eastwood
Flood – Jars Of Clay – Jars Of Clay
Goodbye Mr. Ed – Tin Machine – Oy Vey, Baby
Here Comes The Sun – Nina Simone – Anthology
In Repair – Our Lady Peace – Spiritual Machines
In The End – Linkin Park – Hybrid Theory
Is There Anybody Out There? – Pink Floyd – The

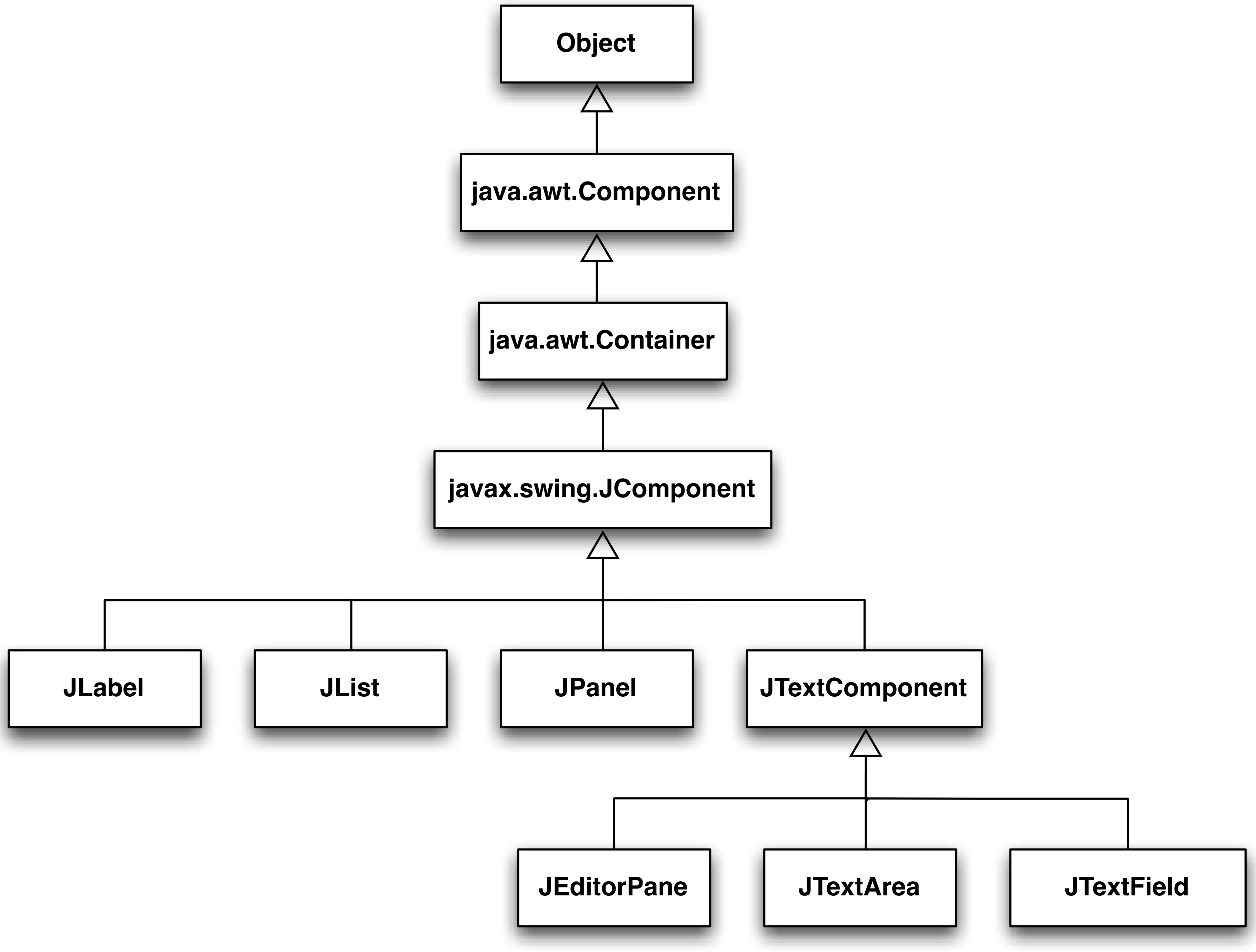
Karma Police – Radiohead – OK Computer
Mosquito Song – Queens Of The Stone Age – Songs For The Deaf
Aerials – System Of A Down – Toxicity

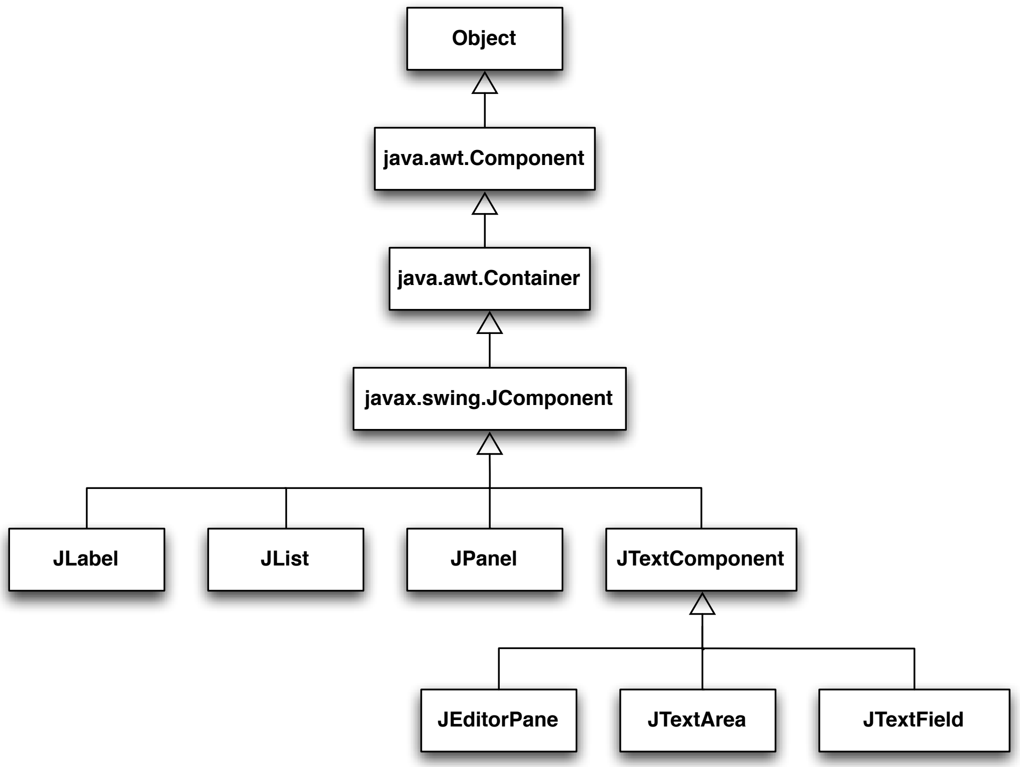
JComponent

A graphical element is called a **component**. Accordingly, there is a class called **JComponent** that defines the characteristics that are common to of all components.

Components include: JLabel, JList, JMenuBar, JPanel, JScrollBar, JTextComponent, etc.




Object
java.awt.Component
java.awt.Container
javax.swing.JComponent
JLabel
JList
JPanel
JTextComponent
JEditorPane
JTextArea
JTextField



AWT and Swing are a rich source of examples of inheritance. A **Component** defines a collection of methods that are common to all the graphical objects, such as **setBackground(Color c)** and **getX()**.

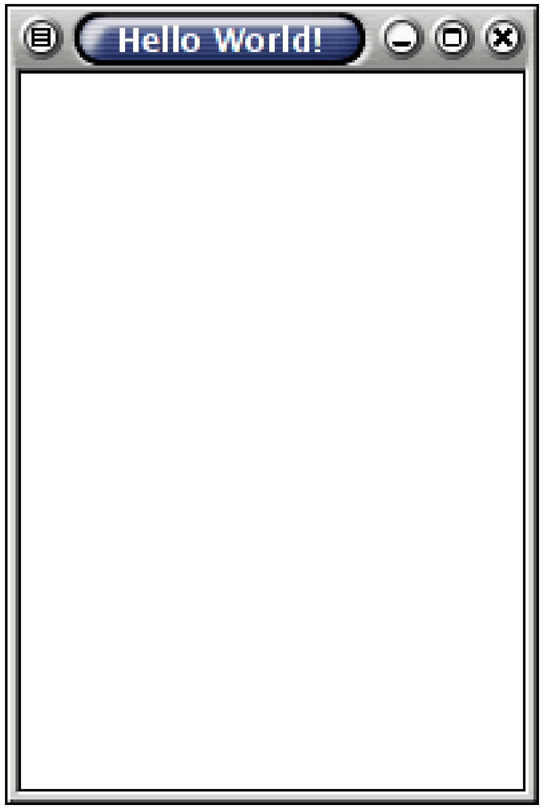
A **Container** contains other graphical components, and therefore declares a method **add(Component c)** and **setLayout(LayoutManager m)**.

Hello World -1-

A **JFrame** is a top-level window with a title and a border.

```
import javax.swing.JFrame;

public class Hello {

    public static void main(String[] args) {

        JFrame f = new JFrame("Hello World!");
        f.setSize(200,300);
        f.setVisible(true);

    }
}
```

$\Rightarrow$ A top-level component (JFrame, JDialog or JApplet) is one that is not contained within any other component.


Hello World!

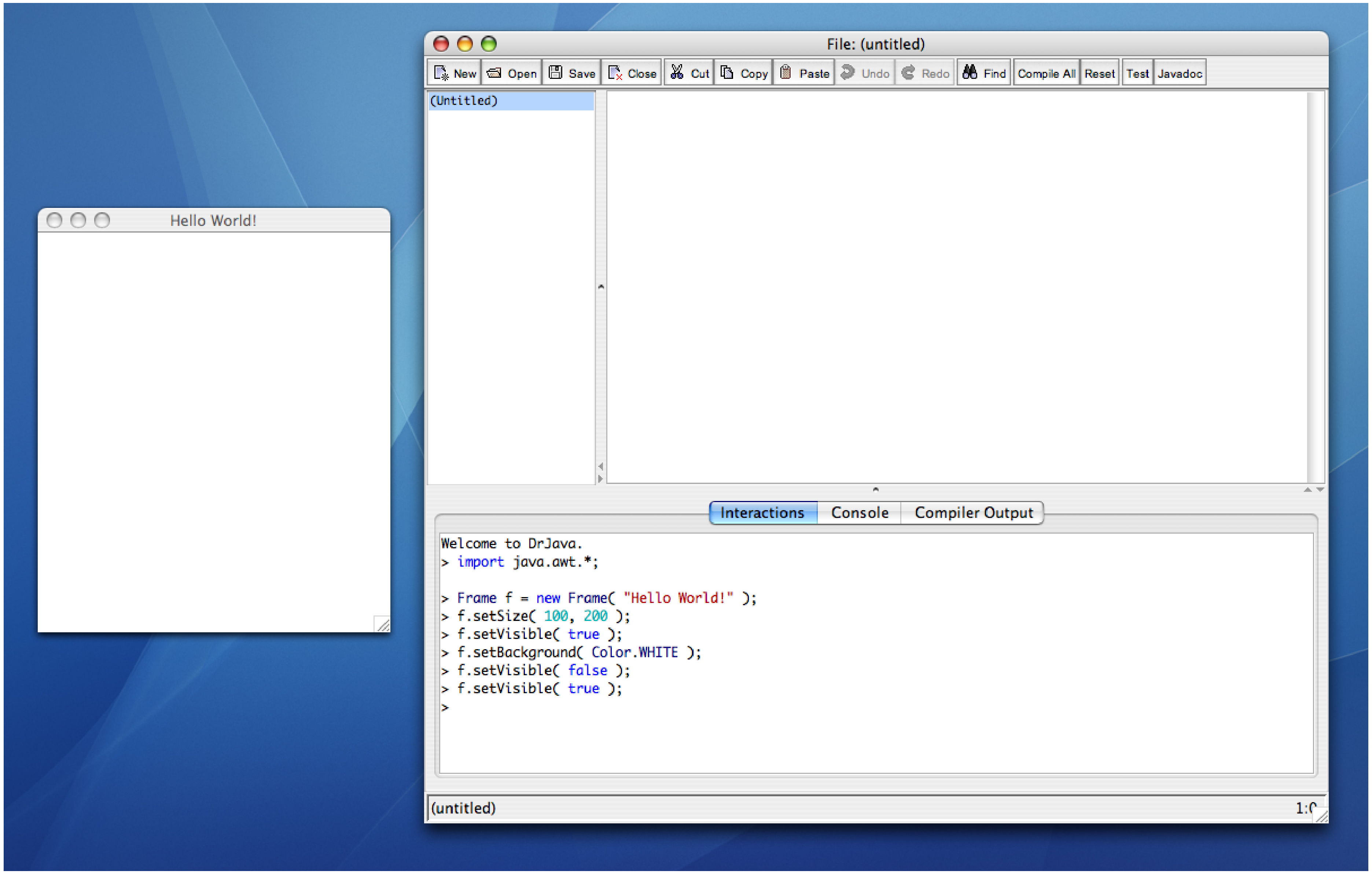
DrJava

Alternatively, use **DrJava** to create and experiment with graphical objects. Use the interactions window and type each of the following statements one by one.

```
> import javax.swing.JFrame;
> JFrame f = new JFrame("Hello World!");
> f.setSize(100,200);
> f.setVisible(true);
> f.setVisible(false);
> f.setVisible(true);
> f.setVisible(false);
```

You'll see that a **JFrame** of object is not visible unless you make it visible.

DrJava



```
Welcome to DrJava.
> import java.awt.*;

> Frame f = new Frame( "Hello World!" );
> f.setSize( 100, 200 );
> f.setVisible( true );
> f.setBackground( Color.WHITE );
> f.setVisible( false );
> f.setVisible( true );
>
```

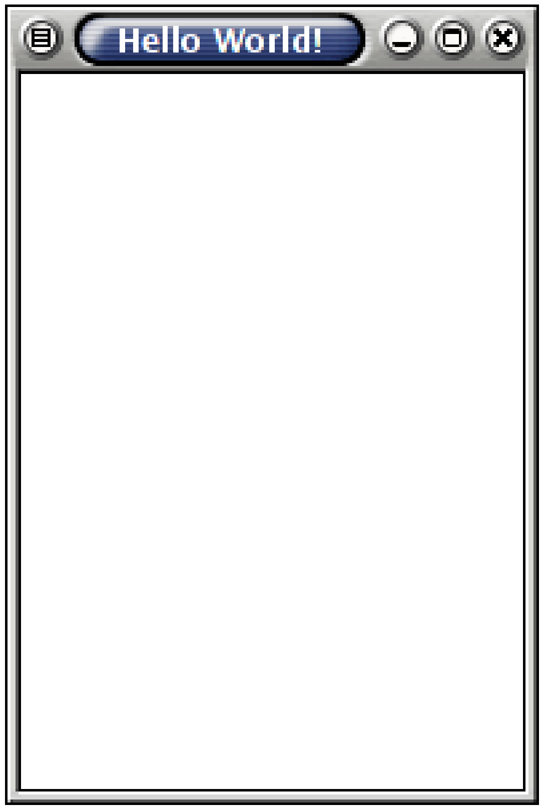
Hello World -2-

Let's create a specialized **JFrame** that has the required characteristics for this application.

```
import javax.swing.JFrame;
public class MyFrame extends JFrame {
    public MyFrame( String title ) {
        super( title );
        setSize( 200, 300 );
        setVisible( true );
    }
}
```

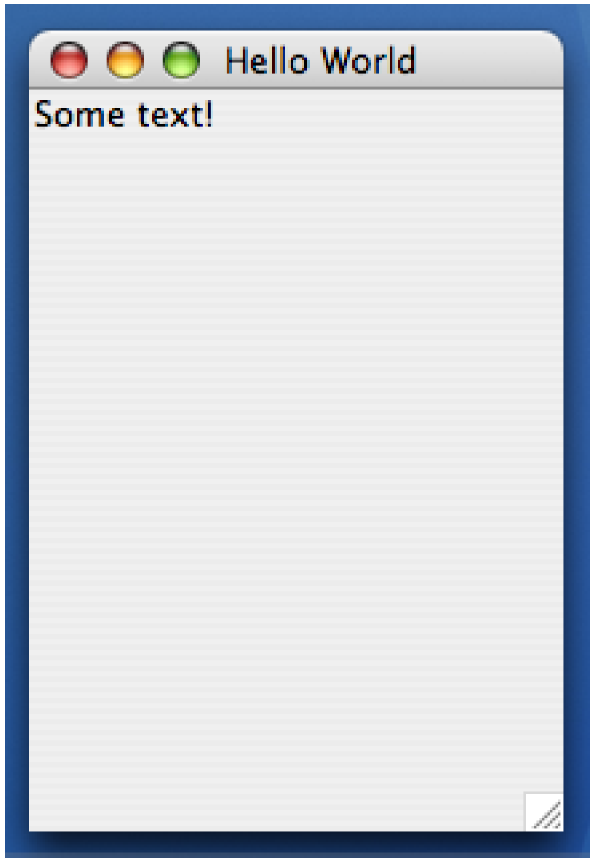
Which would be used as follows:

```
public class Run {
    public static void main( String args[] ) {
        JFrame f = new MyFrame( "Hello World" );
    }
}
```

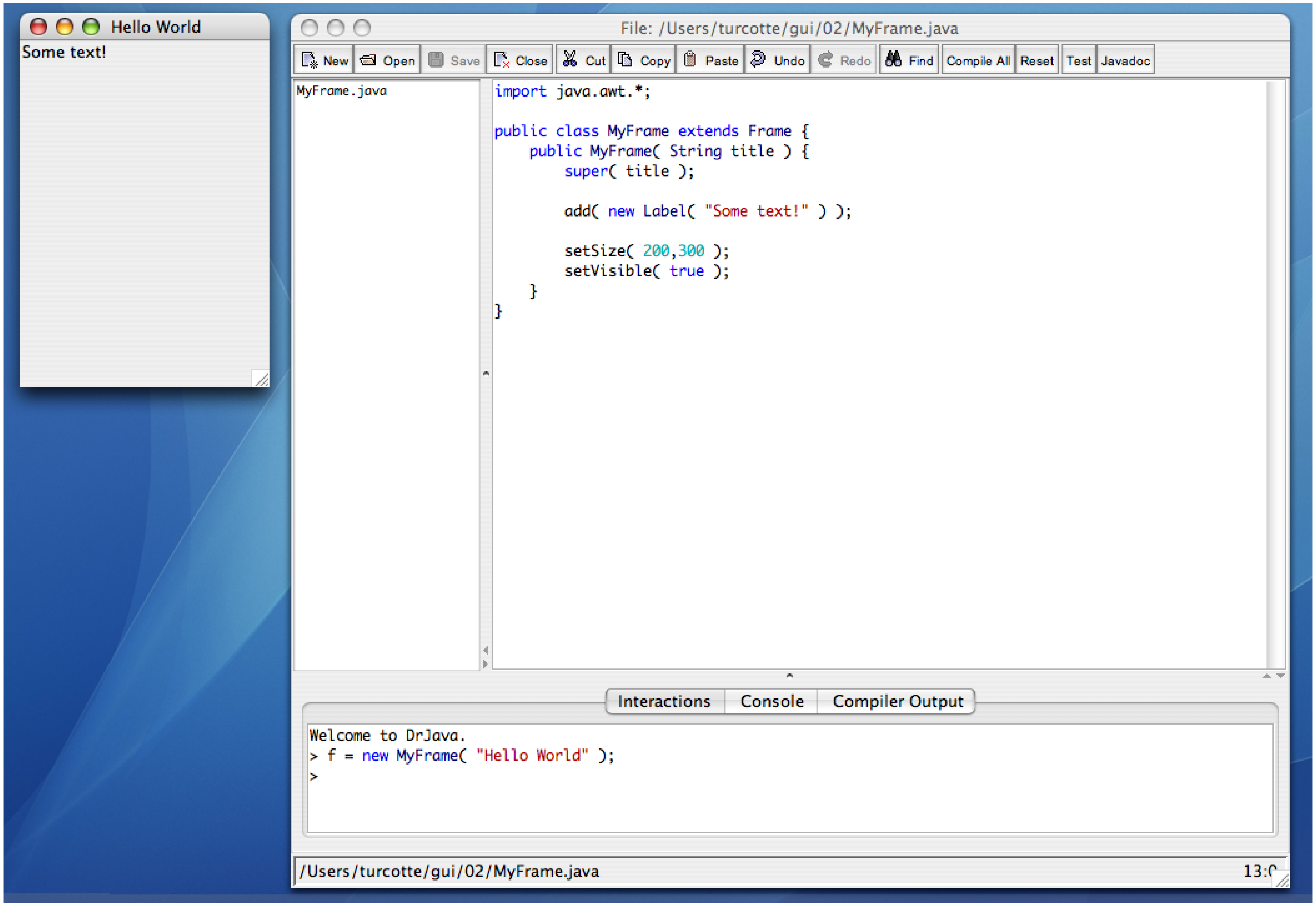

Hello World!

MyFrame is a specialized **JFrame**, which is a specialized **Container**, therefore, it has the ability to contain other components.

```
import javax.swing.*;

public class MyFrame extends JFrame {
    public MyFrame( String title ) {
        super( title );

        add( new JLabel( "Some text!" ) ); // <---

        setSize( 200,300 );
        setVisible( true );
    }
}
```



Some text!



```
import java.awt.*;

public class MyFrame extends Frame {
    public MyFrame( String title ) {
        super( title );

        add( new Label( "Some text!" ) );

        setSize( 200,300 );
        setVisible( true );
    }
}
```

```
Welcome to DrJava.
> f = new MyFrame( "Hello World" );
>
```

LayoutManager

When adding components to a container, we'd like to have control over the placement of the objects (components).

A *layout manager* is an object responsible for placing and sizing the components in a container.

LayoutManager is an interface and Java provides several implementations including **FlowLayout**, **BorderLayout** and **GridLayout**.

FlowLayout adds the components from left to right, from top to bottom, this is the default layout manager for a **JPanel**.

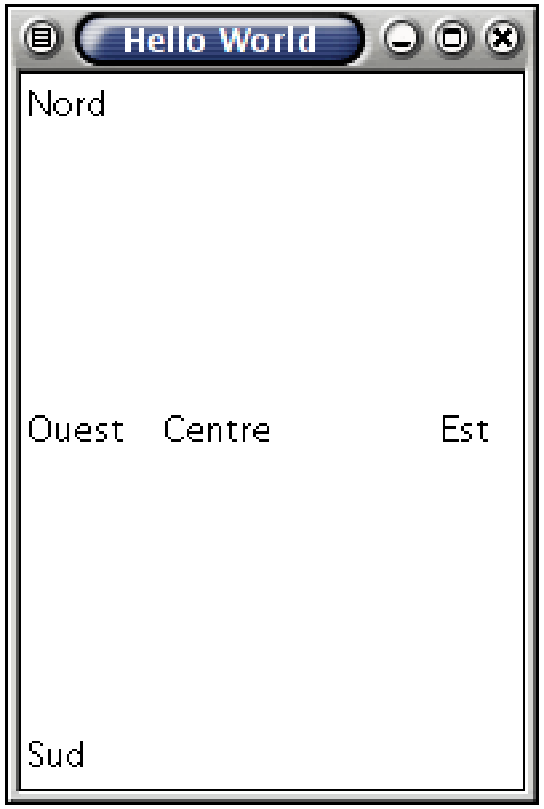
BorderLayout is a layout that divides the container into zones: north, south, est, west and center, this is the default layout manager for a **JFrame**.

GridLayout divides the container into $m \times n$ zones (2 dimensional grid).

$\Rightarrow$ The Java library has approximately 20 layout manager's implementations.

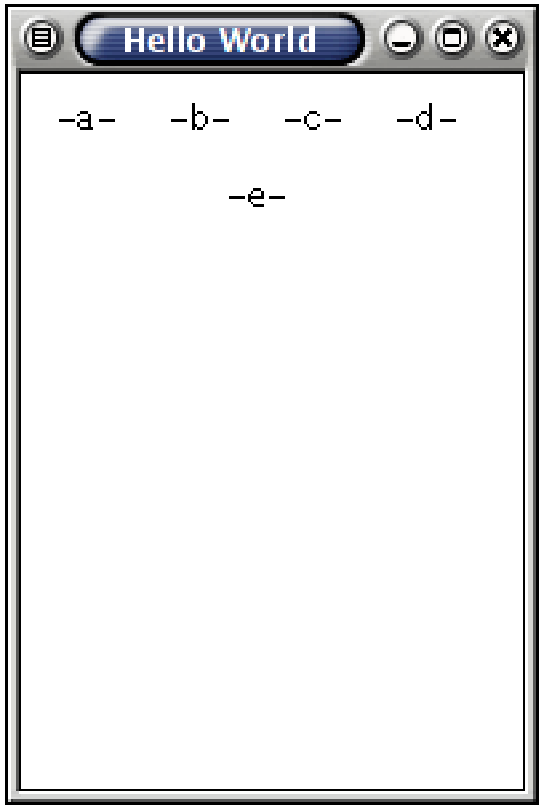
BorderLayout

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

public class MyFrame extends JFrame {

    public MyFrame( String title ) {
        super( title );

        add( new JLabel( "Nord" ), BorderLayout.NORTH );
        add( new JLabel( "Sud" ), BorderLayout.SOUTH );
        add( new JLabel( "Est" ), BorderLayout.EAST );
        add( new JLabel( "Ouest" ), BorderLayout.WEST );
        add( new JLabel( "Centre" ), BorderLayout.CENTER );

        setSize( 200,300 );
        setVisible( true );
    }
}
```


Hello World
Nord
Ouest
Centre
Est
Sud

FlowLayout

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

public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);
        setLayout(new FlowLayout());
        add( new JLabel( "-a-" ) );
        add( new JLabel( "-b-" ) );
        add( new JLabel( "-c-" ) );
        add( new JLabel( "-d-" ) );
        add( new JLabel( "-e-" ) );
        setSize( 200,300 );
        setVisible( true );
    }
}
```

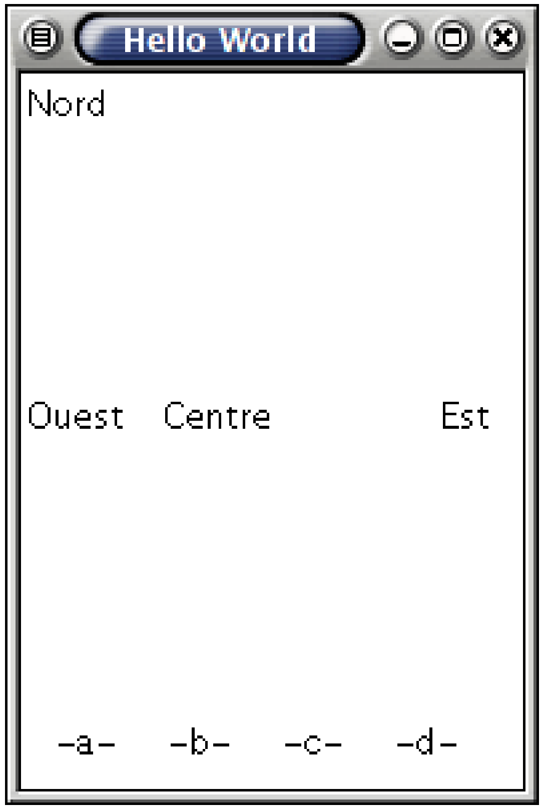

Hello World
-a-
-b-
-c-
-d-
-e-

JPanel

A **JPanel** is the simplest **Container**.

It is used to regroup several components and typically has a different layout manager than the container that it is part of.

```
import java.awt.*;
import javax.swing.*;
public class MyFrame extends JFrame {
    public MyFrame(String title) {
        super(title);
        setLayout(new BorderLayout());
        add(new JLabel("Nord"), BorderLayout.NORTH);
        add(new JLabel("Est"), BorderLayout.EAST);
        add(new JLabel("Ouest"), BorderLayout.WEST);
        add(new JLabel("Centre"), BorderLayout.CENTER);
        JPanel p = new JPanel();           // <----
        p.setLayout(new FlowLayout());
        p.add(new JLabel("-a-"));
        p.add(new JLabel("-b-"));
        p.add(new JLabel("-c-"));
        p.add(new JLabel("-d-"));
        p.add(new JLabel("-e-"));
        add(p, BorderLayout.SOUTH);     // <----
        setSize(200,300);
        setVisible(true);
    }
}
```


Hello World
Nord
Ouest
Centre
Est
-a-
-b-
-c-
-d-

Event-driven programming

Graphical user interfaces are programmed differently than most applications.

In an **event-driven** application, the program waits for something to occur, the user clicks a button or presses a key.

An *event* is an object that represents the action of the user.

In Java, the components are the source of the events.

A component generates an event or is the source of an event.

> When a button is pressed and released, AWT sends an instance of **ActionEvent** to the button, by calling **processEvent** on the button.

Callback

How to associate an action with a graphical element?

Imagine, for a moment, that you are responsible for the implementation of the class **JButton** of Java.

When the button is pressed and released, this object receives, via a call to its method **processEvent(ActionEvent e)**, an instance of the class **ActionEvent** representing this event.

What should be done?

The button should be calling a user-defined method. This method will perform some task, e.g. printing the content of a list of items, sorting elements, etc.

As the programmer of the class JButton, what concept can you use to force the programmer of an application to implement a method with a well defined signature?

No. Not an abstract class. Yes! An interface! Bravo!

Indeed, an interface can be used to force the implementation a method, here **actionPerformed**.

```
public interface ActionListener extends EventListener {

    /**
     * Invoked when an action occurs.
     */

    public void actionPerformed( ActionEvent e );

}
```

The answering machine analogy

You are still playing the role of the programmer responsible for the implementation of the class **JButton** of Java.

Our strategy will be as follows: let's ask the user (programmer of the application) to leave "a message with his/her coordinates" (using **addListener**) so that we can call him/her back (using its method **actionPerformed**) whenever the button is clicked[1].

The method `addListener( ... )` of the button allows an object to register as a listener:

". . . whenever someone clicks the button call me back . . . "

What will be the type of the parameter of the method **addListener(...)**?

How will the button interact with the listener?

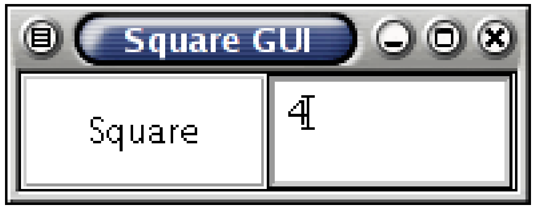
The button interacts with the listener by calling its method **actionPerformed(ActionEvent e)**!

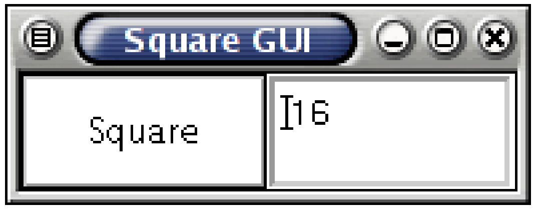
[1]this is called a callback

The parameter must an **ActionListener**!

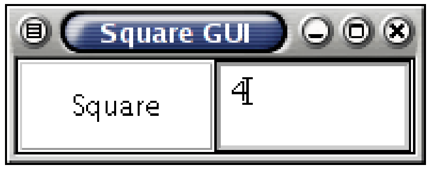
Application: Square

To better understand these concepts, let's create a small application computing the square of a number!



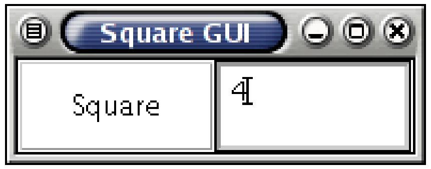


Here are the necessary declarations to create the graphical aspects of the application.



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Square extends JFrame {
    protected JButton button = new JButton( "Square" );
    protected JTextField input = new JTextField();
    public Square() {
        super("Square GUI");
        setLayout(new GridLayout(1,2));
        add(input);
        add(button);
        pack();
        setVisible(true);
    }
}
```

The user will be entering the information using the graphical object **TextField**.



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Square extends JFrame {
    protected JButton button = new JButton( "Square" );
    protected JTextField input = new JTextField();
    public Square() {
        super("Square GUI");
        setLayout(new GridLayout(1,2));
        add(input);
        add(button);
        pack();
        setVisible(true);
    }
}
```

The class **JTextField** has a method **String getText**, which we will be using to retrieve the string entered by the user, as well as a method **setText(String)**, which we will be using to substitute the string entered by entered by user with the square of its integer value. Hence the method **square**:

```
protected void square() {
    int v = Integer.parseInt( input.getText() );
    input.setText( Integer.toString( v*v ) );
}
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Square extends JFrame {
    protected JButton button = new JButton( "Square" );
    protected JTextField input = new JTextField();
    public Square() {
        super("Square GUI");
        setLayout(new GridLayout(1,2));
        add(input);
        add(button);
        pack();
        setVisible(true);
    }
    protected void square() {
        int v = Integer.parseInt(input.getText());
        input.setText(Integer.toString( v*v ));
    }
}
```

What is missing? Registering a listener with the button.

The interface **ActionListener** has only one method, **actionPerformed(ActionEvent e)**.

A **SquareActionListener** will be calling the method **square** of the GUI (**Square**). Therefore, it needs a reference to the GUI (**Square**).

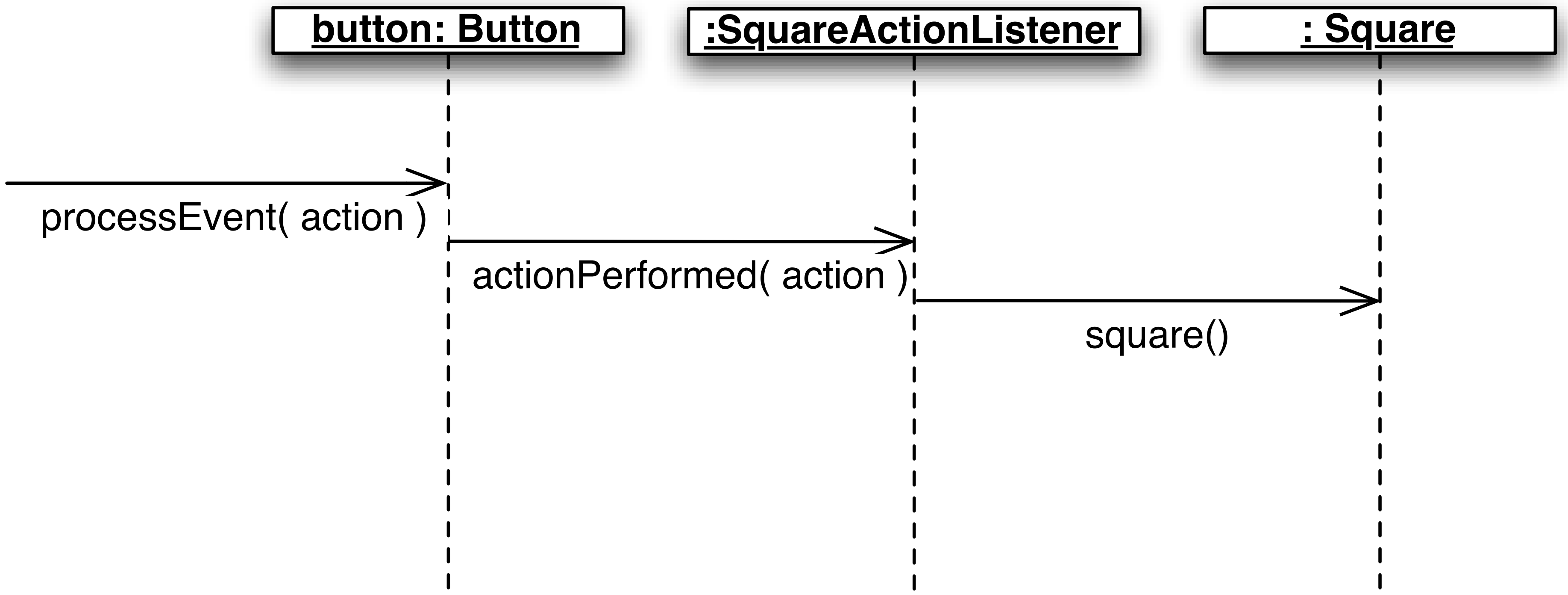
```
import java.awt.event.*; // <--

class SquareActionListener implements ActionListener {

    private Square appl;

    SquareActionListener( Square appl ) {
        this.appl = appl;
    }

    public void actionPerformed( ActionEvent e ) {
        appl.square();
    }
}
```

When a **SquareActionListener** object is created, you are telling it "here is the GUI that you are responsible for" (reference variable **appl**). When the method **actionPerformed** is called, you must call the method **square** of the object designated by **appl**.

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

class SquareActionListener implements ActionListener {

    private Square appl;

    SquareActionListener( Square appl ) {
        this.appl = appl;
    }

    public void actionPerformed( ActionEvent e ) {
        appl.square();
    }
}
```

To handle the events that will be generated by the button, one needs to add (sometimes we say register) an object that implements the interface **ActionListener**.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Square extends JFrame {
    protected JButton button = new JButton( "Square" );
    protected JTextField input = new JTextField();
    public Square() {
        super("Square GUI");
        setLayout(new GridLayout(1,2));
        add(button);
        add(input);
        button.addActionListener(new SquareActionListener(this)); // <--
        pack();
        setVisible( true );
    }
    protected void square() {
        int v = Integer.parseInt(input.getText());
        input.setText(Integer.toString(v*v));
    }
}
```


button: Button
:SquareActionListener
: Square
processEvent(action)
actionPerformed(action)
square()

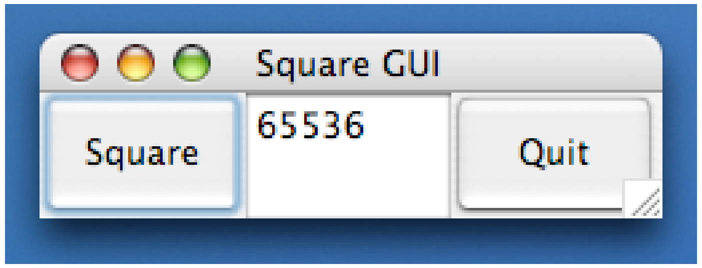
ActionListener (take 2)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Square extends JFrame implements ActionListener {
    private JButton button = new JButton( "Square" );
    private JTextField input = new JTextField();
    public Square() {
        super("Square GUI");
        setLayout(new GridLayout(1,2));
        add(button);
        add(input);
        button.addActionListener(this); // <--
        pack();
        setVisible( true );
    }
    public void actionPerformed(ActionEvent e) {
        int v = Integer.parseInt(input.getText());
        input.setText(Integer.toString(v*v));
    }
}
```

JFrame.EXIT_ON_CLOSE

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Square extends JFrame implements ActionListener {
    private JButton button = new JButton( "Square" );
    private JTextField input = new JTextField();
    public Square() {
        super("Square GUI");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // <--
        setLayout(new GridLayout(1,2));
        add(button);
        add(input);
        button.addActionListener(this);
        pack();
        setVisible( true );
    }
    public void actionPerformed(ActionEvent e) {
        int v = Integer.parseInt(input.getText());
        input.setText(Integer.toString(v*v));
    }
}
```

Let's add a button to quit the application.



The class **Square** will be the event-handler for both buttons.

Therefore, the method **actionPerformed** must be able to distinguish between an event that originated from pressing the button square and one that originated from pressing the button quit!

Fortunately, an **Event** encapsulates this information, see method **getSource()**.

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

public class Square extends Frame implements ActionListener {

    Button bSquare = new Button( "Square" );
    Button bQuit = new Button( "Quit" ); // <--
    IntField input = new IntField();

    public Square() {
        super( "Square GUI" );
        setLayout( new GridLayout( 1,3 ) );
        add( bSquare );
        bSquare.addActionListener( this );

        add( input );
        input.setValue( 2 );
        add( bQuit );
        bQuit.addActionListener( this ); // <--
        addWindowListener( new SquareWindowAdapter() );
        pack();
        setVisible( true );
    }
```

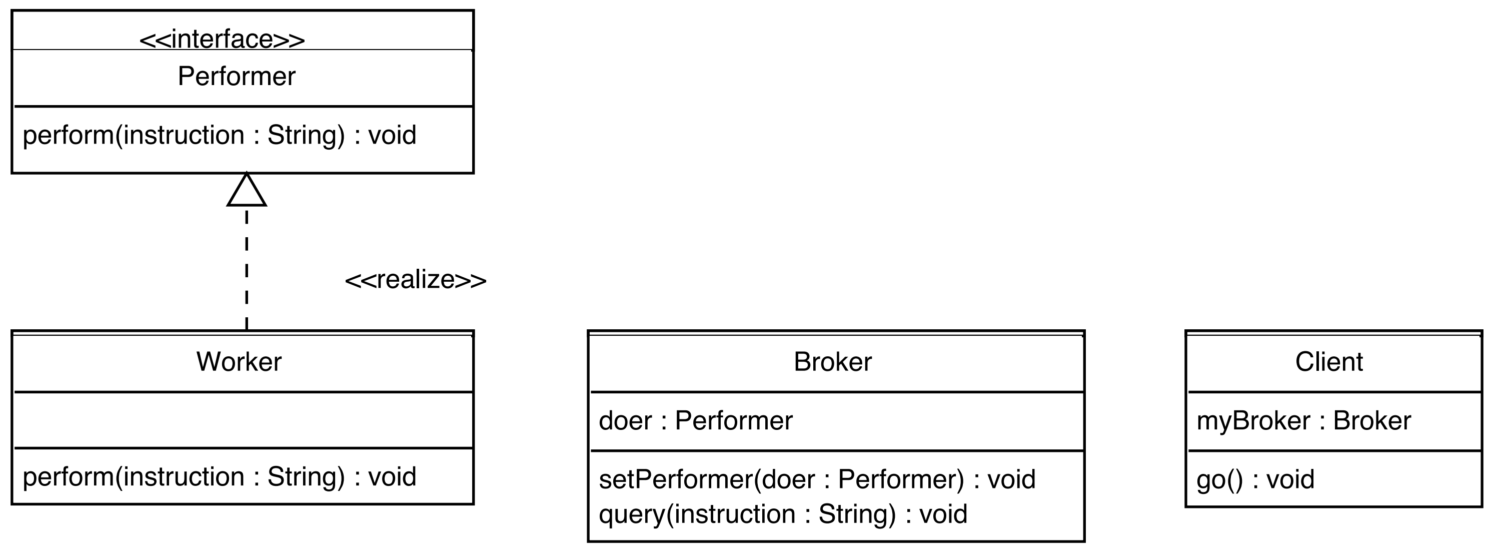
```
    public void actionPerformed( ActionEvent e ) {

        if ( e.getSource() == bSquare ) {
            square();
        } else if ( e.getSource() == bQuit ) {
            System.exit(0);
        }
    }

    protected void square() {
        int v = input.getValue();
        input.setValue( v*v );
    }
}
```

```
class IntField extends TextField {
    public int getValue() {
        return Integer.parseInt( getText() );
    }
    public void setValue( int v ) {
        setText( Integer.toString(v) );
    }
}
```

Summary

- **AWT** and **Swing** make extensive use of inheritance
- **Event-driven** programming is at the heart of graphical user interfaces
- **ActionListener** is an interface that declares a method called **actionPerformed(ActionEvent e)**
- Components, such as **JButton**, have a method **addActionListener** allowing objects implementing **AcctionListener** to become a handler for future events.

Callback: simple example


<<interface>>
Performer
perform(instruction : String) : void
<<realize>>
Worker
perform(instruction : String) : void
Broker
doer : Performer
setPerformer(doer : Performer) : void
query(instruction : String) : void
Client
myBroker : Broker
go() : void

Callback: simple example

```
public interface Performer {
    public abstract void perform( String instruction );
}
```

Callback: simple example

```
public class Worker implements Performer {

    public void perform( String instruction ) {
        System.out.println( "performing: " + instruction );
    }

}
```

Callback: simple example

```
public class Broker {

    private Performer doer;

    public void setPerformer( Performer doer ) {
        this.doer = doer;
    }

    public void query( String instruction ) {

        if ( doer == null ) {
            throw new IllegalStateException( "no performer" );
        }

        doer.perform( instruction );
    }
}
```

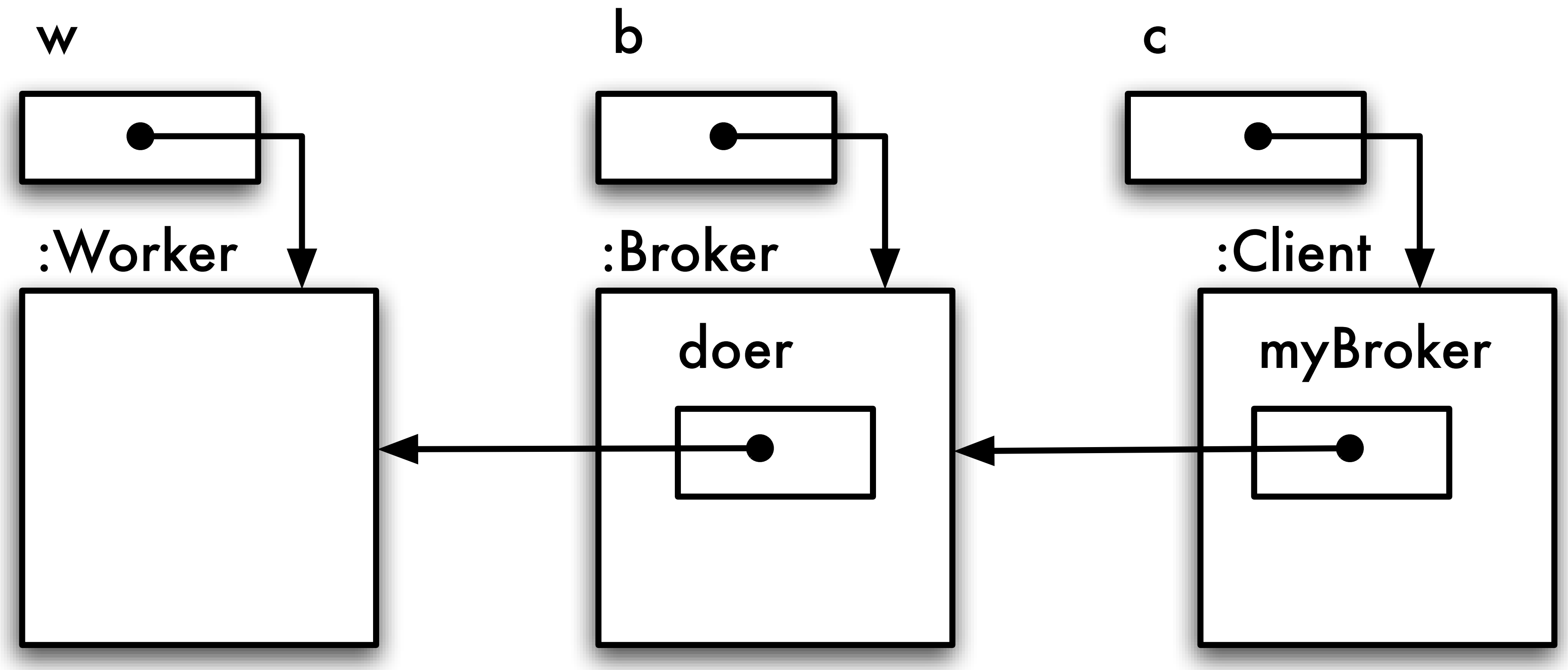
Callback: simple example

```
public class Client {

    private Broker myBroker;

    public Client( Broker myBroker ) {
        this.myBroker = myBroker;
    }

    public void go() {
        myBroker.query( "some action" );
    }

}
```

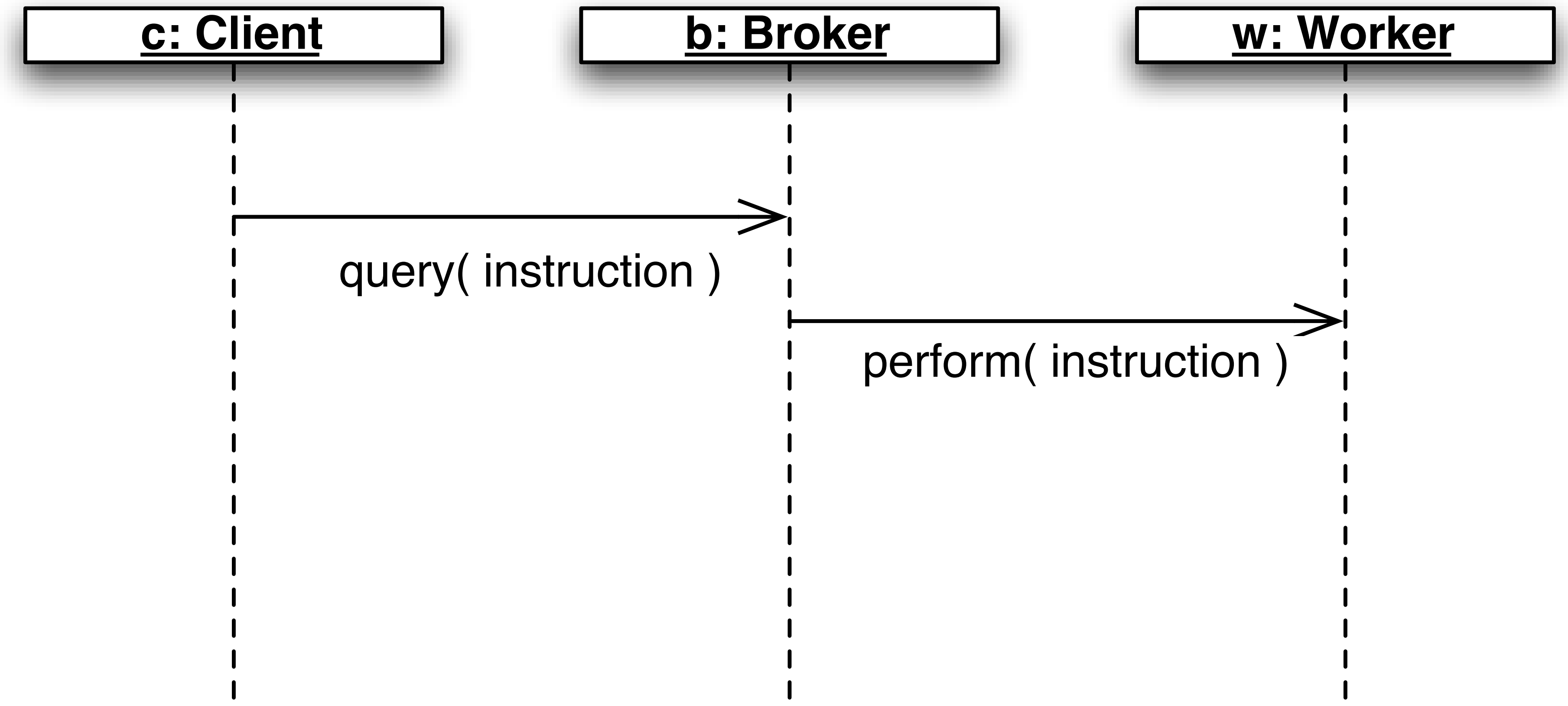
Callback: simple example

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

        Broker b;
        b = new Broker();

        Worker w;
        w = new Worker();

        b.setPerformer( w );

        Client c;
        c = new Client( b );

        c.go();

    }
}
```

Callback: simple example



Callback: simple example


c: Client
b: Broker
w: Worker
query(instruction)
perform(instruction)