ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École de science informatique et de génie électrique
Université d'Ottawa

Version du 4 février 2012

Résumé

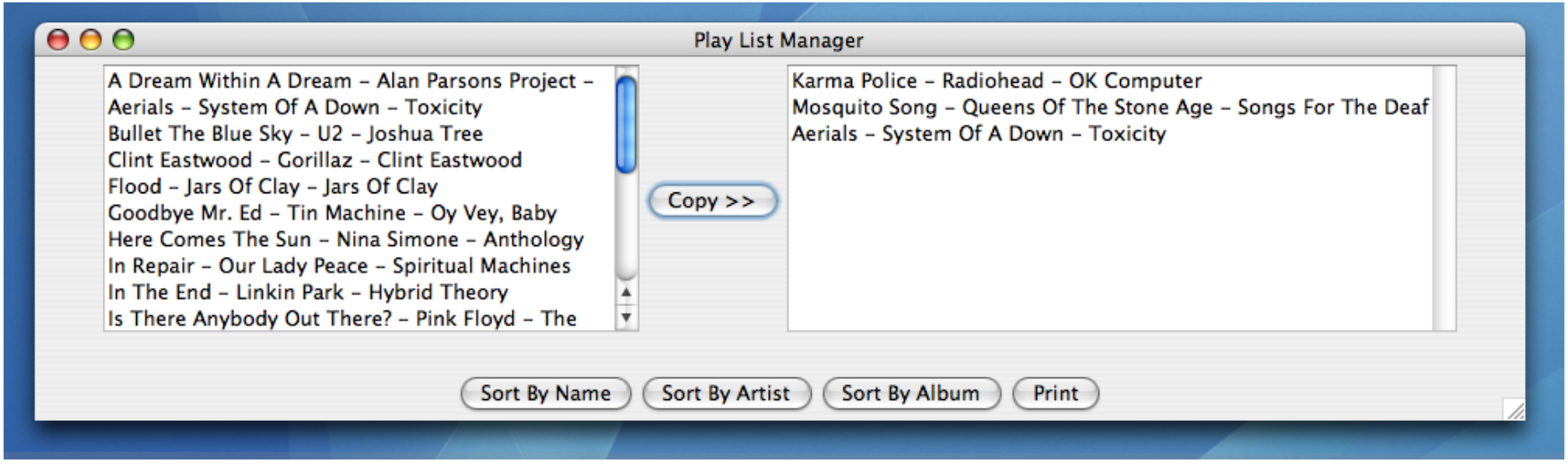
- Éléments graphiques
- **Programmation événementielle**
- Classe imbriquée non-static
- Model-View-Controller

*. Pensez-y, n'imprimez ces notes de cours que si c'est nécessaire!

AWT

Abstract Window Toolkit (**AWT**) est la plus ancienne librairie de classes utilisée afin de construire des interfaces graphiques en Java. **AWT** a fait partie de Java depuis son tout début.

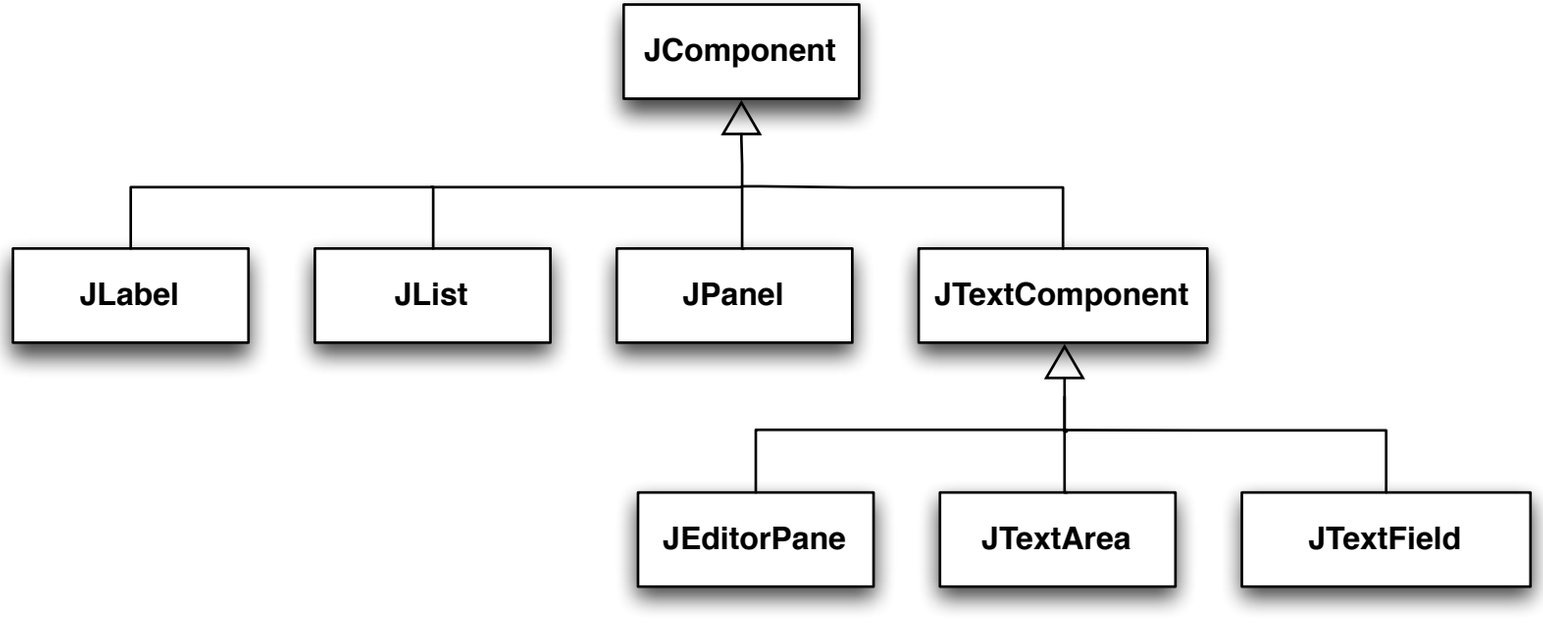
Swing est une librairie améliorée et plus récente.

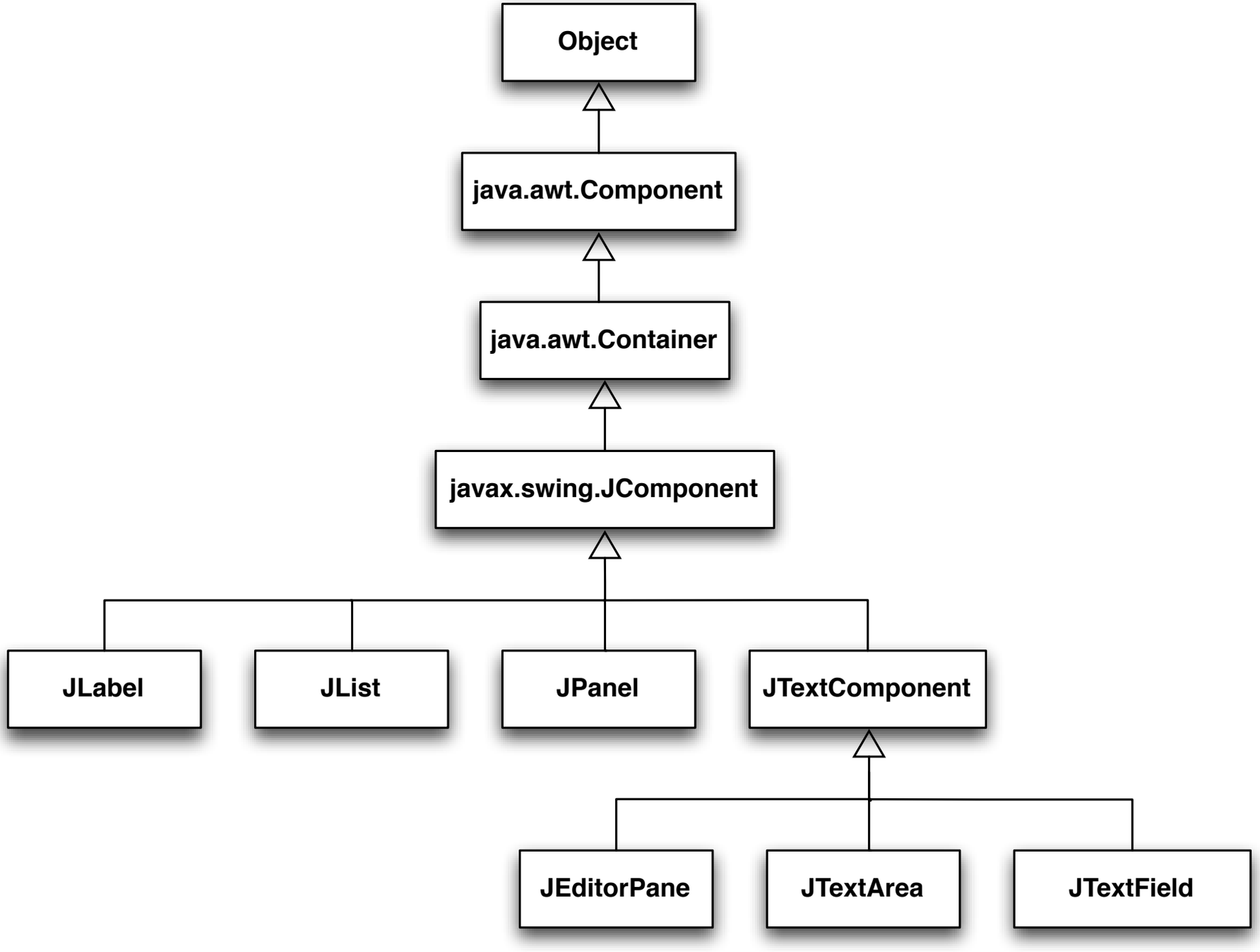


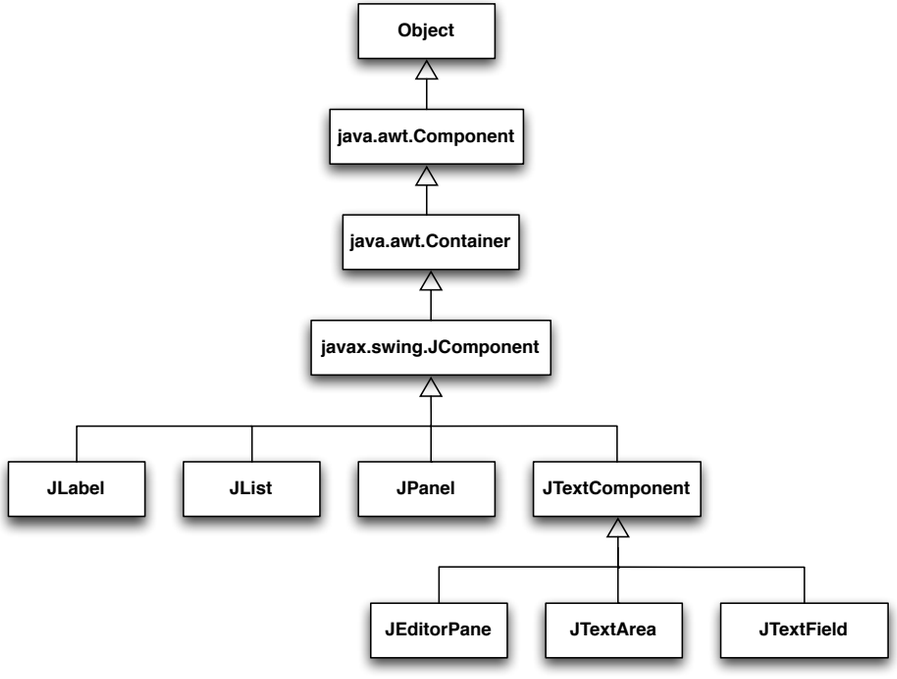
JComponent

Un élément graphique s'appelle une composante graphique (**component**). Conséquemment, il existe une classe nommée **JComponent** qui définit les caractéristiques communes des composantes.

Les sous-classes de **JComponent** incluent : JLabel, JList, JMenuBar, JPanel, JScrollBar, JTextComponent, et.




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



AWT et **Swing** utilisent fortement l'héritage. La classe **Component** définit l'ensemble des méthodes communes aux objets graphiques, telles que **setBackground(Color c)** et **getX()**.

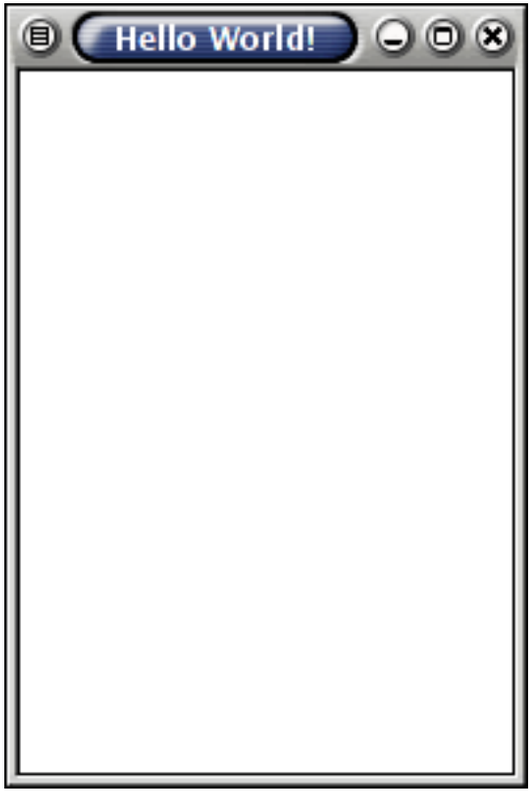
La classe **Container** définie le comportement des objets graphiques pouvant contenir des objets graphiques, la classe définit les méthodes **add(Component component)** et **setLayout(LayoutManager mgr)**, entre autres.

Hello World -1-

La classe **JFrame** décrit un élément graphique ayant un titre, une bordure.

```
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);

    }
}
```

$\Rightarrow$ Les objets des classes **JFrame**, **JDialog** et **JApplet** ne peuvent être insérés à l'intérieur d'autres composantes graphiques (en anglais on dit qu'il s'agit de ≪top-level components≫).


Hello World!

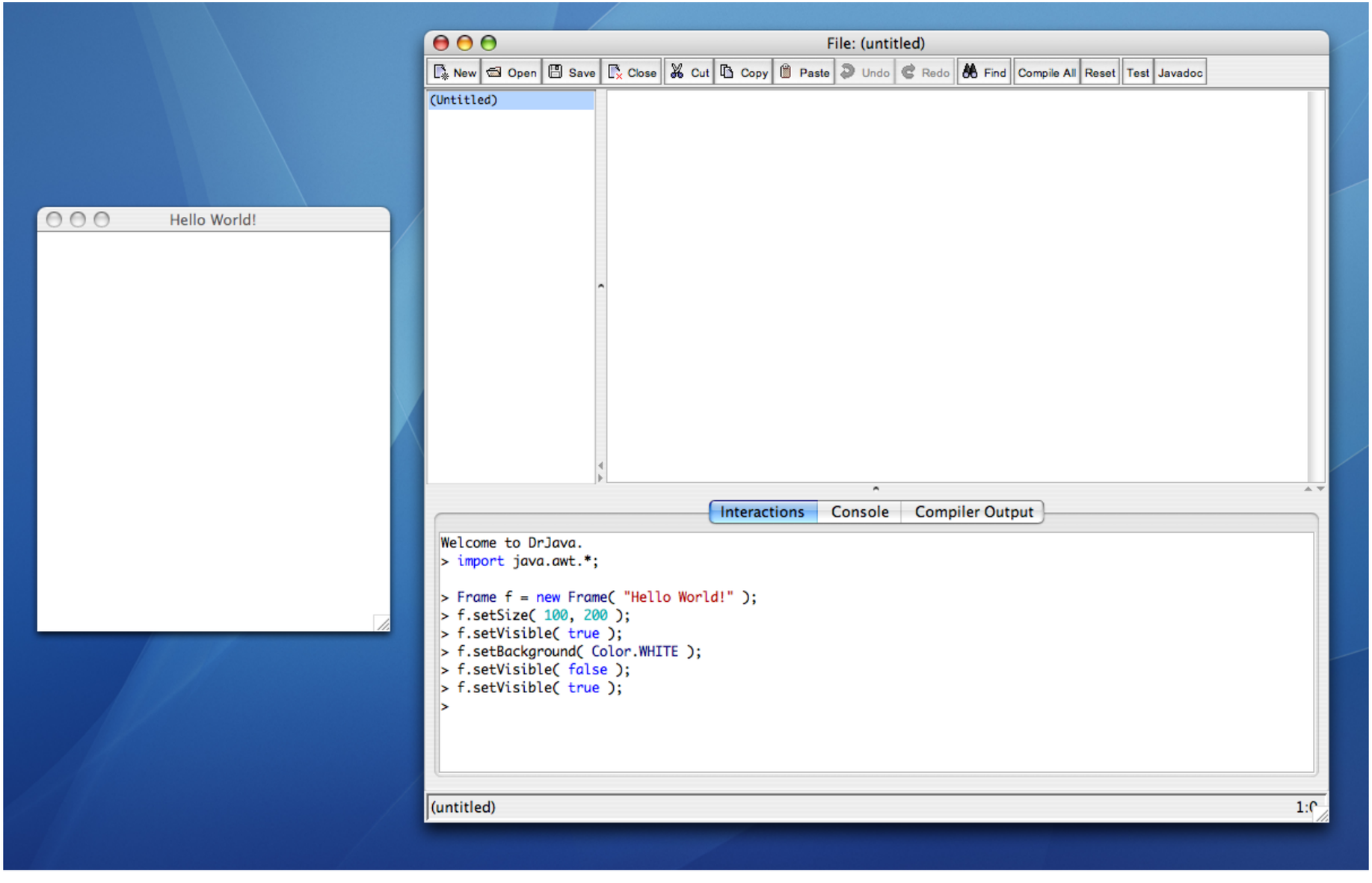
DrJava

On peut aussi expérimenter à partir de la fenêtre d'interactions de **DrJava**. Exécutez les énoncés suivants un à un.

```
> 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);
```

Vous verrez que la fenêtre n'est pas visible au départ.

DrJava


Hello World!
File: (untitled)
New
Open
Save
Close
Cut
Copy
Paste
Undo
Redo
Find
Compile All
Reset
Test
Javadoc
(Untitled)
Interactions
Console
Compiler Output
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);
>
(untitled)

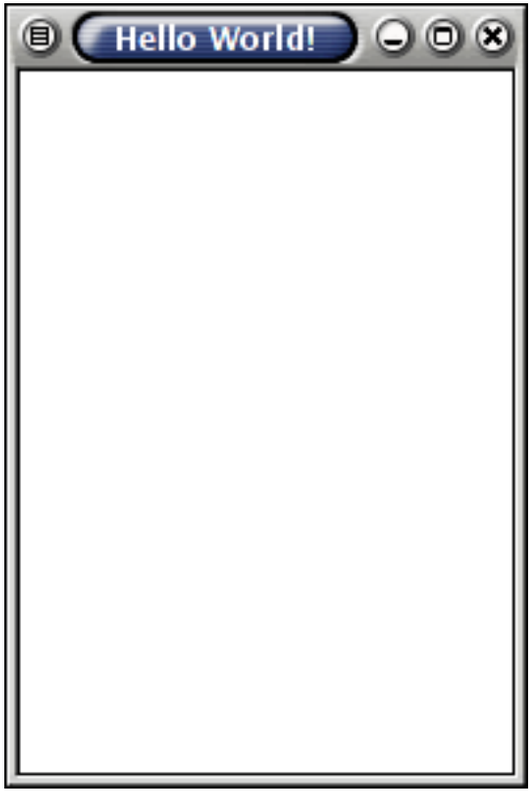
Hello World -2-

Créons une classe spécialisée de **Frame** ayant toutes les caractéristiques requises pour cette application. Le constructeur se charge de déterminer l'aspect initial de la fenêtre.

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

qu'on utilise comme ceci :

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


Hello World!

MyFrame est une spécialisation de la classe **JFrame**, qui est elle-même une spécialisation de la classe **Frame**, qui specialise la classe **Window**, qui elle même spécialise **Container**. Ainsi, **MyFrame** peut donc contenir d'autres éléments graphiques.

```
import javax.swing.*;

public class MyFrame extends JFrame {

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

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

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

De quelle méthode **add** s'agit-il ?


Hello World
Some text!

LayoutManager

Lorsqu'on ajoute des éléments graphiques, on souhaite contrôler leur disposition.

On appelle *layout manager*, l'objet qui contrôle la disposition et la taille des objets dans un conteneur.

LayoutManager est une interface et Java fournit plus de 20 implémentations pour elle. Les principales classes sont :

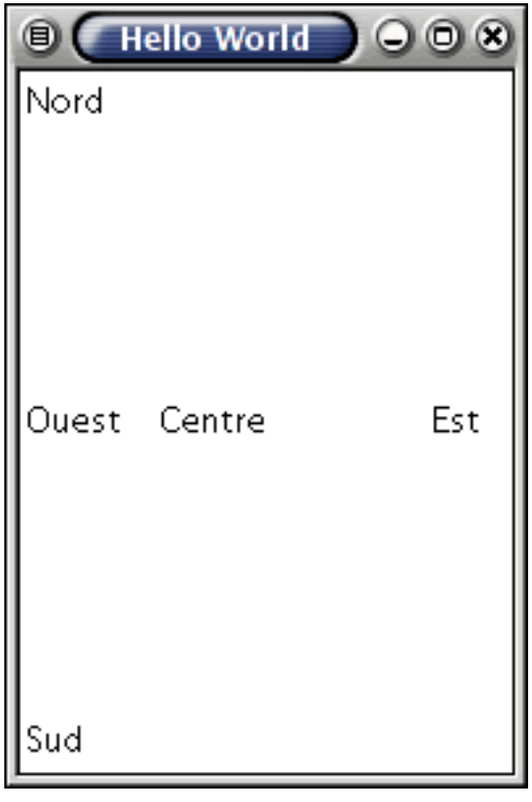
FlowLayout ajoute les éléments graphiques de gauche à droite et de haut en bas ; c'est le gestionnaire de défaut pour **JPanel** (le plus simple des conteneurs).

BorderLayout divise le conteneur en 5 zones : nord, sud, est, ouest et centre, le défaut pour la classe **JFrame**.

GridLayout divise le conteneur en $m \times n$ zones.

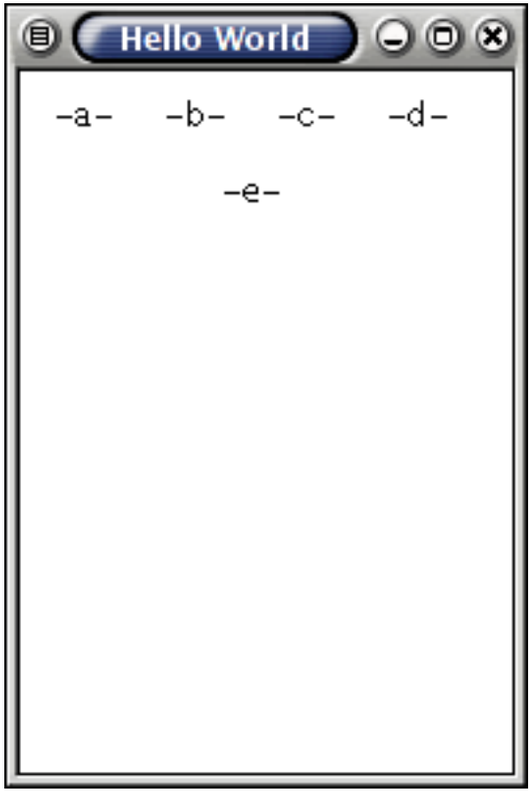
BorderLayout

```
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 );
    }
}
```

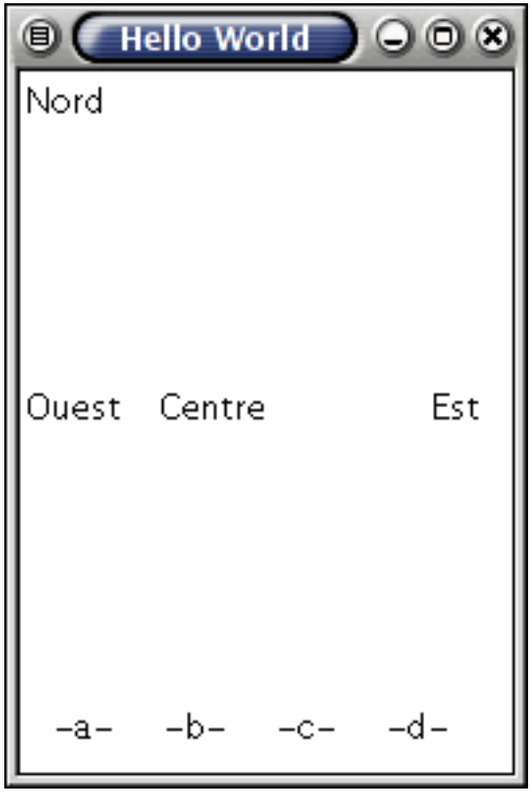

Hello World
-a-
-b-
-c-
-d-
-e-

JPanel

La classe **JPanel** définit le conteneur le plus simple.

Un **JPanel** permet de regrouper plusieurs éléments graphiques et de leur associer un layout manager.

```
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-

Programmation événementielle

(*event-driven*)

Les applications graphiques sont programmées dans un style qui diffère des autres types d'applications.

L'application est presque toujours en attente d'une action de la part de l'usager ; cliquer sur un bouton par exemple.

Un événement est un objet qui représente l'action de l'usager à l'intérieur de l'application graphique.

En Java, les éléments graphiques (Component) sont la source des événements.

On dit qu'un objet génère un événement ou en est la source.

Lorsque qu'un bouton est pressé puis relâché, AWT envoie une instance de la classe **ActionEvent** au bouton, par le biais de la méthode **processEvent** de l'objet de la classe **JButton**.

Callback

Comment associer des actions aux éléments graphiques ?

Mettons-nous dans peau de la personne responsable de l'implémentation de classe **JButton** de Java.

Lorsque le bouton sera enfoncé puis relâché, le bouton recevra, via un appel à sa méthode **processEvent(ActionEvent e)**, un objet de la classe **ActionEvent** représentant cet événement.

Que faire ?

Il faudrait faire un appel à une méthode de l'application. Cette méthode fera le traitement nécessaire, par exemple imprimer la liste des items du client.

Quel concept pouvons-nous utiliser afin de forcer le programmeur à implémenter une méthode ayant une signature bien définie ? (un certain nom, une certaine liste de paramètres)

Non. Bravo !

En effet, le concept d'interface peut-être utilisé afin de forcer l'implémentation d'une méthode, ici **actionPerformed**.

```
public interface ActionListener extends EventListener {

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

    public void actionPerformed( ActionEvent e );

}
```

Analogie du répondeur téléphonique

Nous sommes toujours dans peau du programmeur de l'implémentation de la classe **JButton** de Java.

Notre stratégie sera la suivante : demandons à l'application de nous laisser ses "coordonnées" (addListener) et nous la rappellerons (actionPerformed) lorsque le bouton aura été pressé [1].

La méthode **addListener(...)** du bouton permet à un objet de s'enregistrer comme auditeur (listener) :

"lorsque le bouton aura été pressé, appelle-moi"

Quel est le type du paramètre de la méthode **addListener(...)** ?

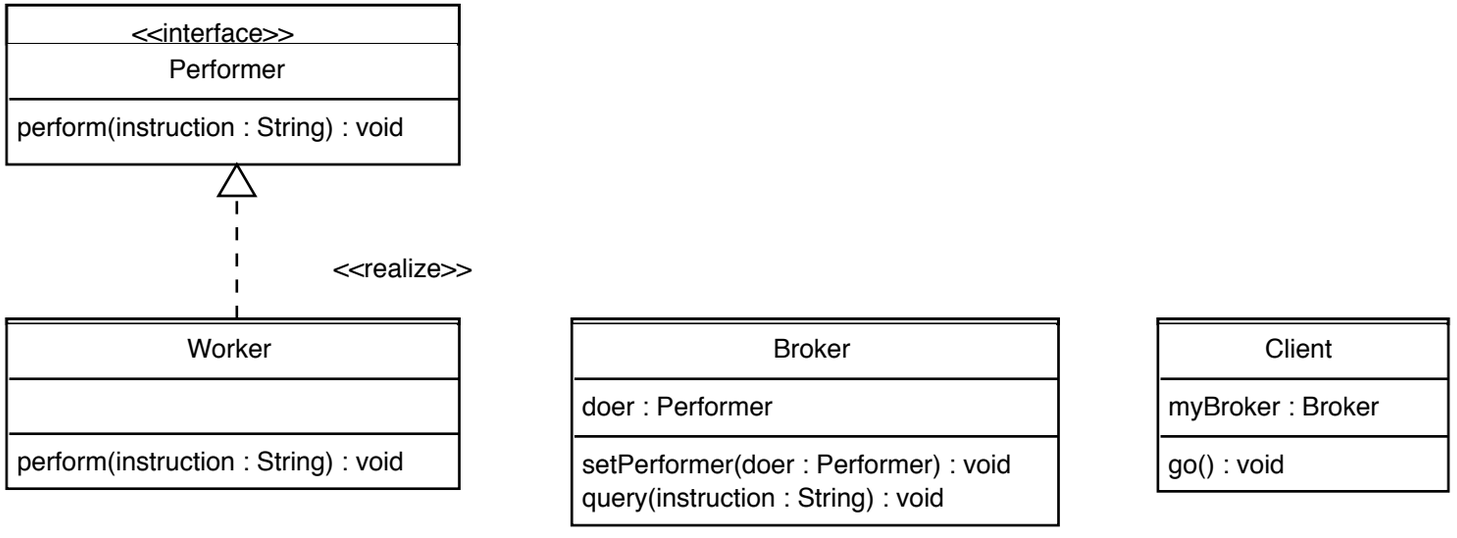
Hum, comment allez-vous interagir avec cet auditeur ?

À l'aide de sa méthode **actionPerformed(ActionEvent e)** !

Cet objet devra être un **ActionListener** !.

1. cette stratégie s'appelle "callback"

Exemple simple de callback



Exemple simple de callback

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

Exemple simple de callback

```
public class Worker implements Performer {

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

}
```

Exemple simple de callback

```
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 );
    }
}
```

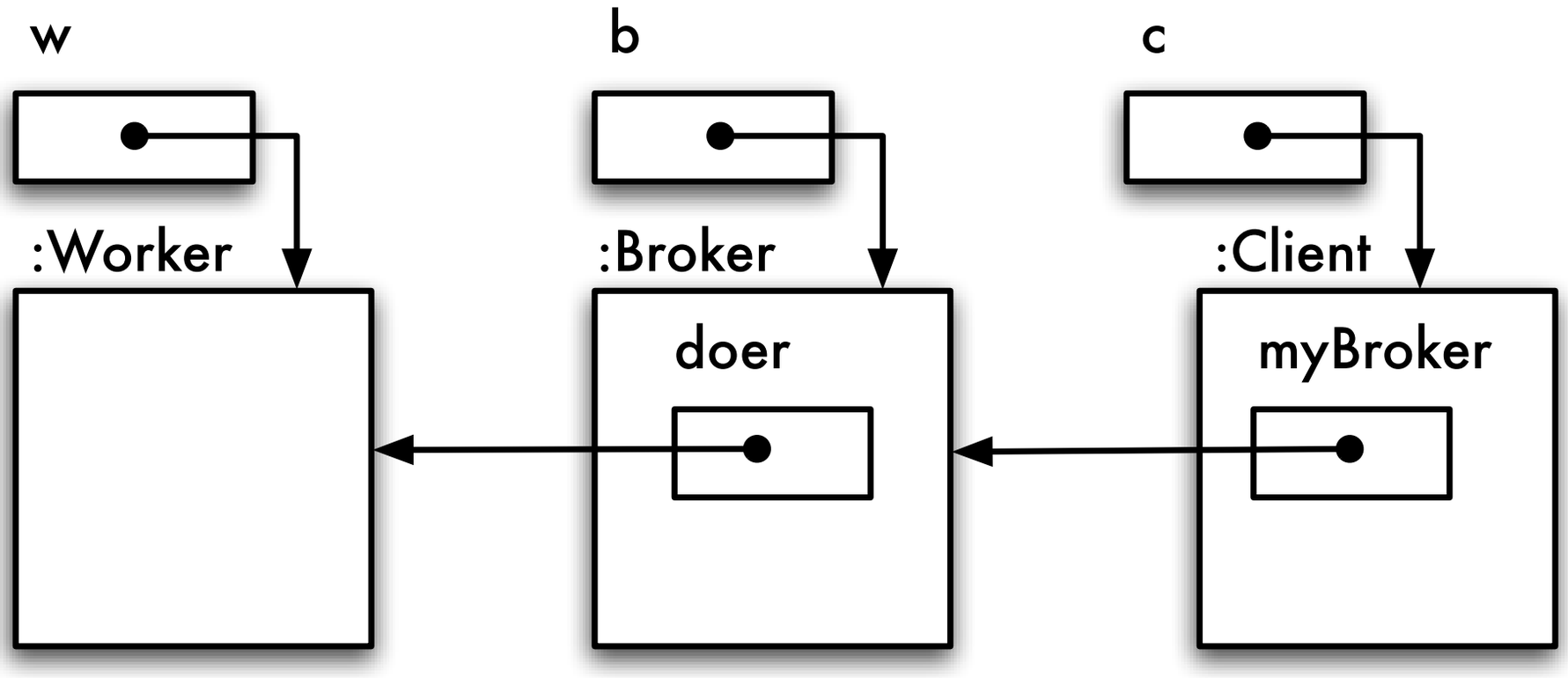
Exemple simple de callback

```
public class Client {

    private Broker myBroker;

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

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

}
```

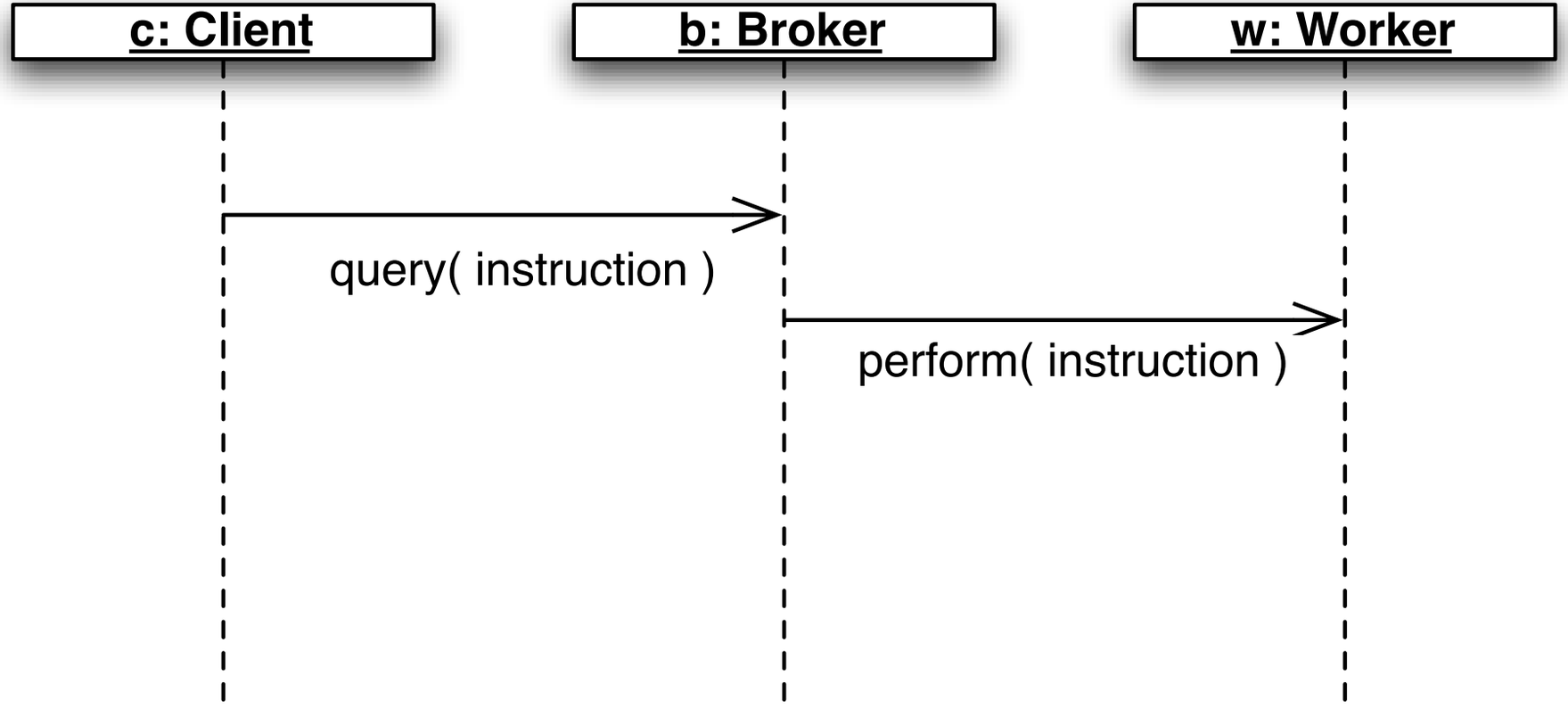
Exemple simple de callback

```
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();

    }
}
```

Exemple simple de callback



Exemple simple de callback



Application Square

Afin de mieux comprendre, nous allons créer une petite application affichant le carré d'un nombre !

Voici les déclarations nécessaires afin de créer l'aspect graphique de l'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);
    }
}
```

L'usager va entrer ses informations à l'aide de l'objet graphique **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);
    }
}
```

La classe **JTextField** possède une méthode **getText()**, que nous utiliserons afin d'obtenir la chaîne de l'usager, ainsi qu'une méthode **setText(String)**, que nous utiliserons afin de remplacer la chaîne de l'usager par son carré. Voici donc le contenu de la méthode **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 ));
    }
}
```

Qu'est-ce qui manque ? Associer un auditeur au bouton !

L'interface **ActionListener** n'a qu'une méthode **actionPerformed(ActionEvent e)**.

Un objet **SquareActionListener** fera un appel à la méthode **square** d'un objet **Square**, il lui faut donc une référence vers cet objet.

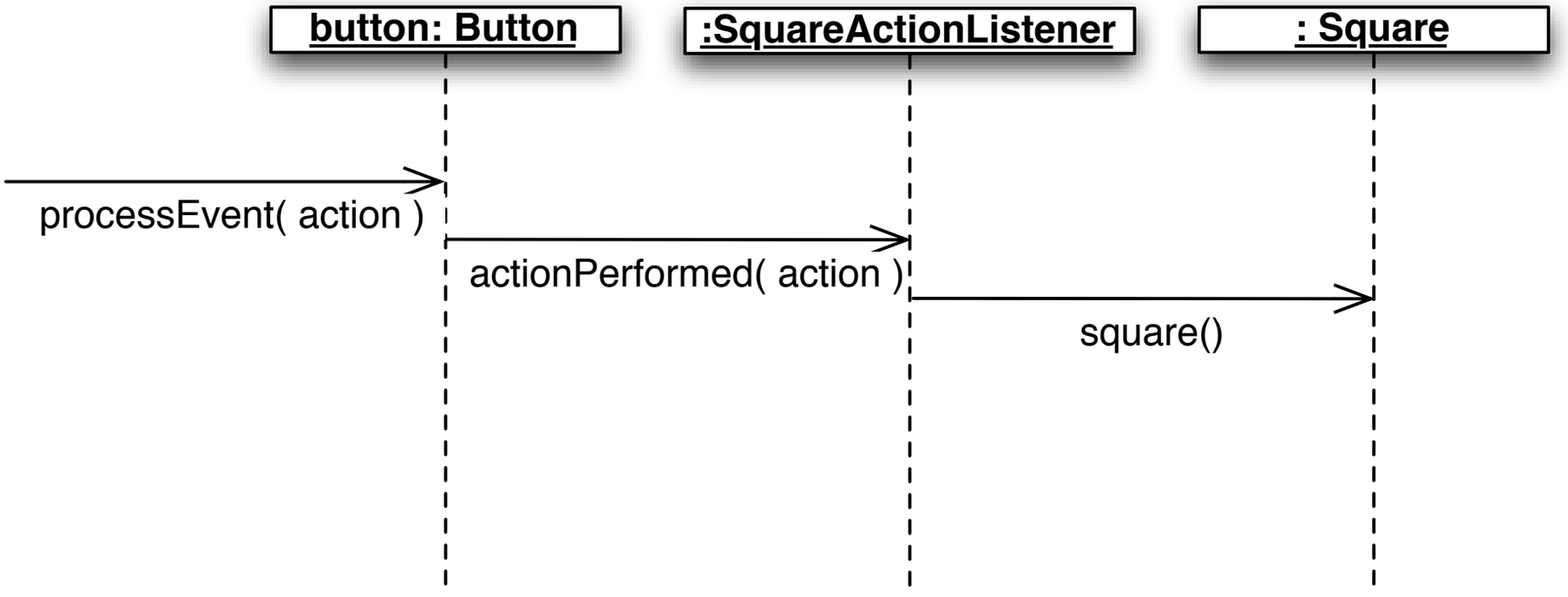
```
class SquareActionListener implements ActionListener {

    private Square appl;

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

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

Lorsqu'un objet **SquareActionListener** est créé, nous lui disons, "voici l'application dont tu es responsable". Lors d'un appel à ta méthode **actionPerformed** tu devras appeler la méthode **square** de l'objet désigné par **appl**.

```
class SquareActionListener implements ActionListener {

    private Square appl;

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

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

Afin de gérer les événements générés par le bouton, on doit ajouter (enregistrer) un objet réalisant l'interface **ActionListener** au bouton.

```
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 (prise 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));
    }
}
```

Fin !