ITI 1521. Introduction à l'informatique II

Interface utilisateur graphique

by

Marcel Turcotte

Version du 10 mars 2020

Préambule

Préambule

Aperçu

Interface utilisateur graphique

Nous explorons l'application de concepts vus antérieurement, notamment les interfaces et l'héritage, pour la conception des interfaces utilisateur graphiques. Nous verrons que les interfaces utilisateur graphiques nécessitent un style de programmation tout particulier que l'on nomme « programmation événementielle ».

Objectif général :

- Cette semaine, vous serez en mesure de concevoir l'interface utilisateur graphique d'une application simple.

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Appliquer** les concepts de l'héritage afin de produire le rendu visuel d'une interface utilisateur graphique.
- **Concevoir** un gestionnaire d'événement afin de produire les comportements nécessaires à la suite d'une action de l'usager.

Préambule

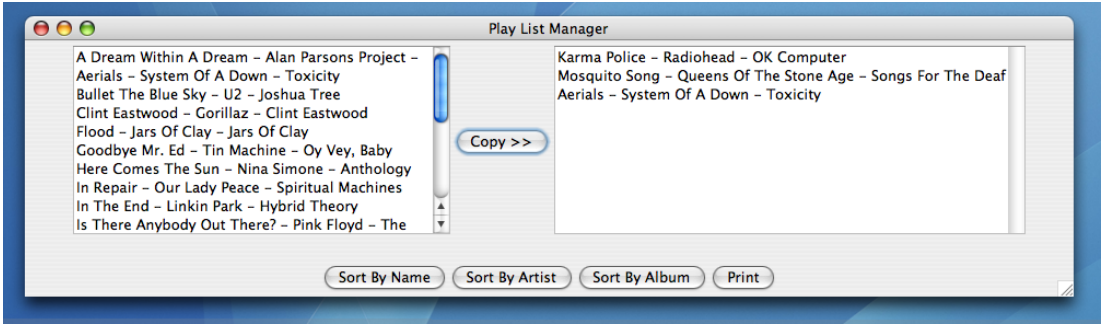
Plan du module

Plan

Rendu graphique

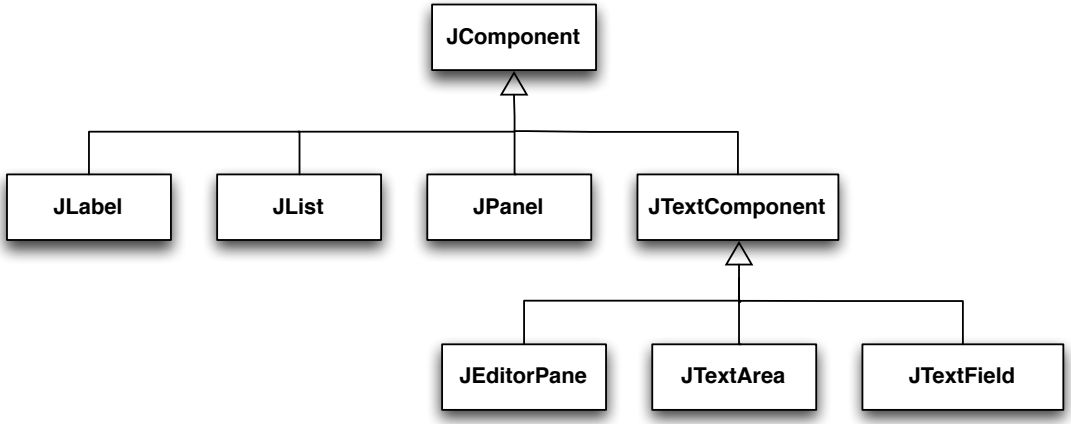
AWT, Swing, et JavaFX

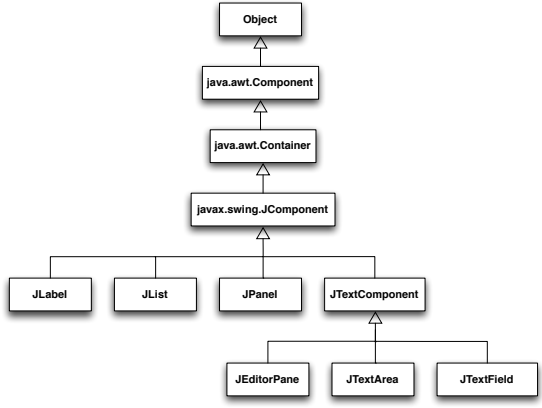
- *Abstract Window Toolkit* (**AWT**) est la plus ancienne librairie de classes utilisée afin de construire des interfaces graphiques en Java. **AWT** a fait partie de Java depuis son tout début.
- **Swing** est une librairie améliorée et plus récente.
- **JavaFX** est le dernier né.



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, etc.





- **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.

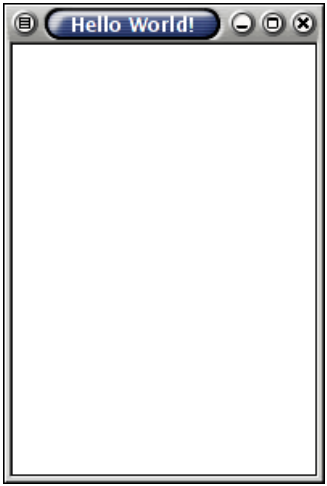
Hello World (1.0)

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

```
import javax.swing.JFrame;

public class Hello {

    public static void main(String[] args) {
        JFrame f;
        f = new JFrame("Hello World!");
        f.setSize(200,300);
        f.setVisible(true);
    }
}
```

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 (1.0)



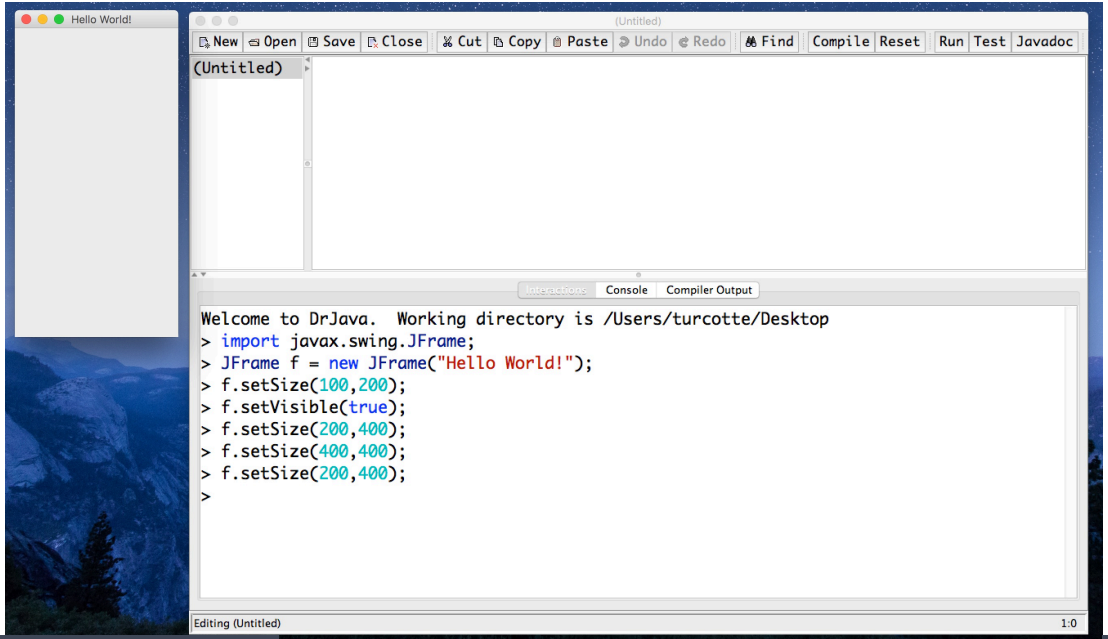
DrJava : Hello World (1.0)

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 (1.0)



Hello World (2.0) : illustration de l'héritage

- Une **classe spécialisée** de **JFrame** 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);
    }
}
```

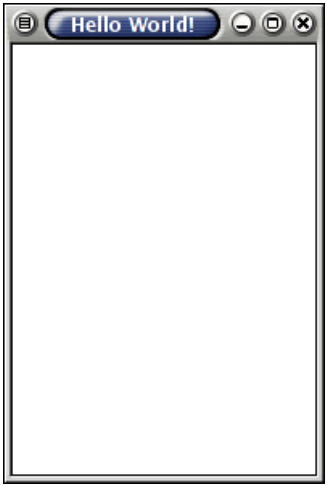
Hello World (2.0)

```
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) {
        MyFrame f;
        j = new MyFrame("Hello World");
    }
}
```

Hello World (2.0)



Ajouter des éléments graphiques

MyFrame est une spécialisation de la classe **JFrame**, qui est elle-même une spécialisation de la classe **Frame**, qui spécialise 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

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.

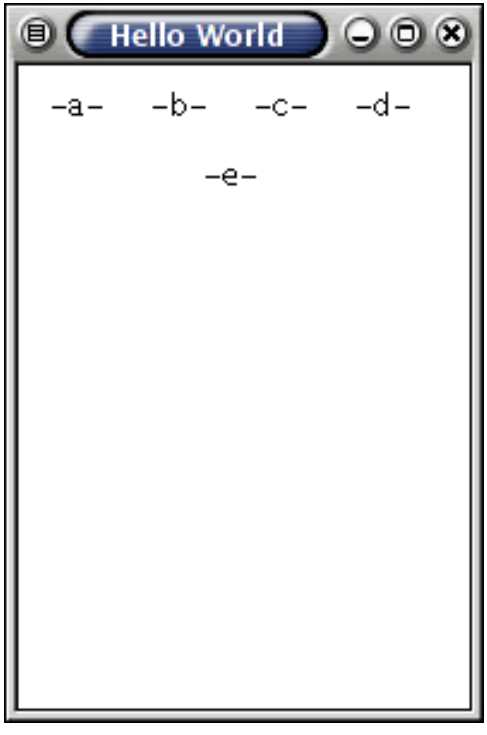
BorderLayout

```
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

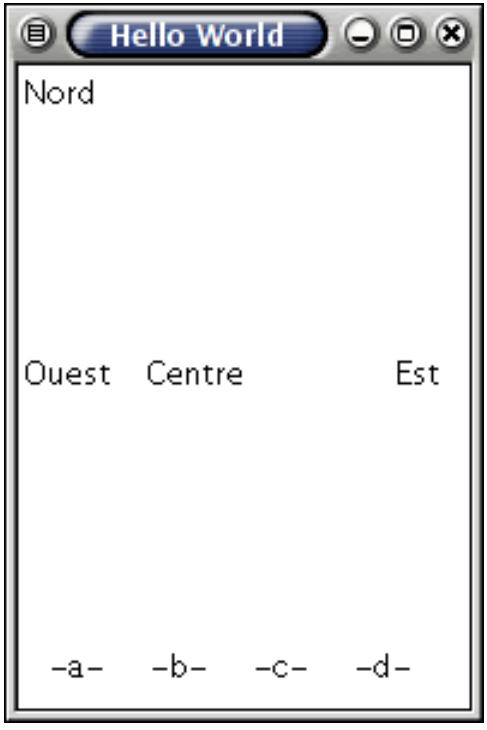
```
import java.awt.*;
import javax.swing.*;

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 : créer des rendus visuels complexes

- 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-"));
        add(p, BorderLayout.SOUTH);       // <----
        setSize(200,300);
        setVisible(true);
    }
}
```


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

Programmation événementielle

Programmation événementielle

(***event-driven programming***)

- Les applications graphiques sont programmées dans un **paradigme** 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.

Programmation événementielle

- 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**.

Fonction de rappel (*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 un objet **ActionEvent**, via un appel à sa méthode **processEvent(ActionEvent e)**.
- **Que faire ?**
- Il faudrait **faire un appel à une méthode de l'application**. Cette méthode fera le traitement nécessaire.
- Quel concept pouvons-nous utiliser afin de **forcer le programmeur à implémenter une méthode ayant une signature bien définie** ? (Un nom spécifique, une liste spécifique de paramètres)

ActionListener

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.
- 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 ?
- Sa méthode **actionPerformed(ActionEvent e)** !
- Cet objet devra réaliser l'interface **ActionListener** !

Exemple : 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.


```
public class Square extends JFrame {

    private JTextField input = new JTextField();

    public Square() {
        super("Square GUI");
        setLayout(new GridLayout(1,2));
        add(input);
        JButton button = new JButton("Square");
        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** :

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

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

public class Square extends JFrame {

    private JTextField input = new JTextField();

    public Square() {
        super("Square GUI");
        setLayout(new GridLayout(1,2));
        add(input);
        JButton button = new JButton("Square");
        add(button);
        pack();
        setVisible(true);
    }

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

Que manque-t-il à notre application ?

- Que **manque-t-il** à notre application ?
- L'application doit savoir qu'il faut **appeler la méthode square** lorsque le bouton est enfoncé !

```java
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

```
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);
}
```

Prologue

- Les classes de **AWT** et **Swing** sont organisée de façon hierarchique (héritage).
- Le placement des éléments graphique est sous le contrôle d'un gestionnaire, un objet qui réalise l'interface **LayoutManager**.
- Les applications utilisateur graphiques sont programmées selon le modèle de **programmation événementielle**.
  - Un classe doit réalisée l'interface **ActionListener**
  - Cette classe doit implémenter la méthode **actionPerformed**.
  - La référence d'un objet dont la classe réalise l'interface **ActionListener** est fournit au bouton via la méthode **addActionListener**

- Types **paramétrés** (« *generics* »)

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa