ITI 1121. Introduction to Computing II

Graphical User Interface (GUI)

by

Marcel Turcotte

Version April 4, 2020

Preamble

Preamble

Overview

Overview

Graphical User Interface (GUI)

We explore the application of previously seen concepts, including interfaces and inheritance, to the design of graphical user interfaces. We will see that graphical user interfaces require a special style of programming called "event-driven programming".

General objective:

- This week you will be able to design the graphical user interface of a simple application.

Preamble

Learning objectives

Learning objectives

- **Apply** inheritance concepts to produce the visual rendering of a graphical user interface.
- **Design** an event handler to produce the necessary behaviors following a user action.

Preamble

Plan

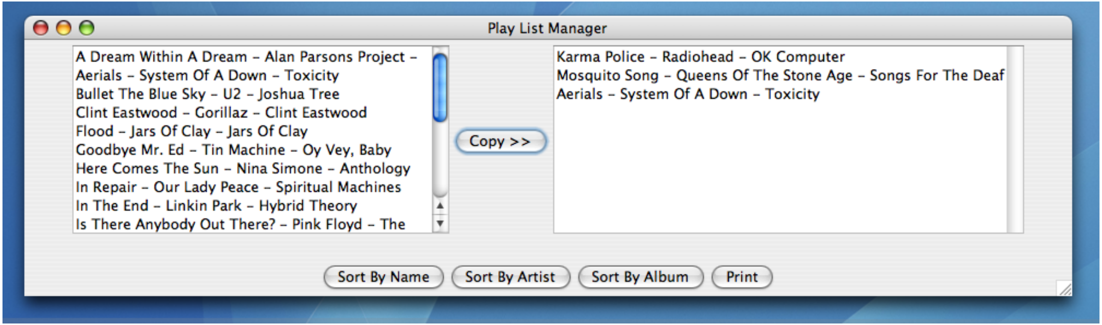
Plan

1. Preamble
2. Graphic rendering
3. LayoutManager
4. Event-oriented programming
5. Prologue

Graphic rendering

AWT, Swing, and JavaFX

- **Abstract Window Toolkit** (**AWT**) is the oldest class library used to build graphical interfaces in Java. **AWT** has been part of Java since its very beginning.
- **Swing** is an improved and newer library.
- **JavaFX** is the latest.



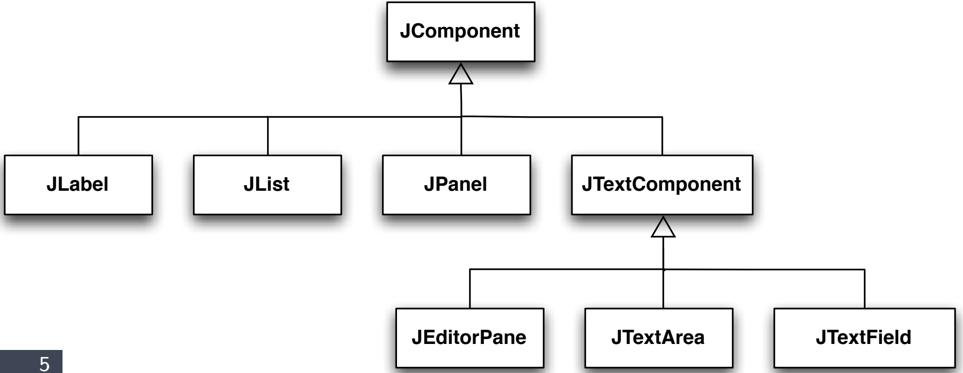
Play List Manager

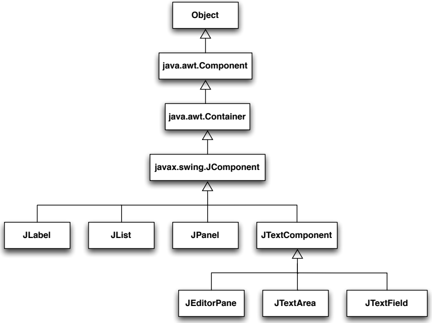
A Dream Within A Dream – Alan Parsons Project –
Aerials – System Of A Down – Toxicity
Bullet The Blue Sky – U2 – Joshua Tree
Clint Eastwood – Gorillaz – Clint Eastwood
Flood – Jars Of Clay – Jars Of Clay
Goodbye Mr. Ed – Tin Machine – Oy Vey, Baby
Here Comes The Sun – Nina Simone – Anthology
In Repair – 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 graphical **component**. Consequently, there is a class named **JComponent** which defines the characteristics co mmunes of the components.
- Subclasses of **JComponent** include: JLabel, JList, JMenuBar, JPanel, JScrollBar, JTextComponent, etc.





- **AWT** and **Swing** use inheritance heavily. The **Component** class defines the set of methods common to graphical objects, such as **setBackground(Color c)** and **getX()**.
- The class **Container** defines the behavior of graphical objects that can contain graphical objects, the class defines the methods **add(Component component)** and **setLayout(LayoutManager mgr)**, among others.

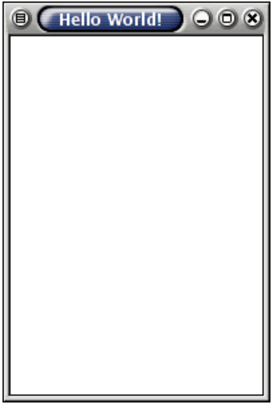
Hello World (1.0)

The **JFrame** class describes a graphical element with a title and a border.

```java
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);
    }
}
```

Objects of the classes **JFrame**, **JDialog** and **JApplet** cannot be inserted inside other graphical components (we say that they are "top-level components").

Hello World (1.0)



DrJava: Hello World (1.0)

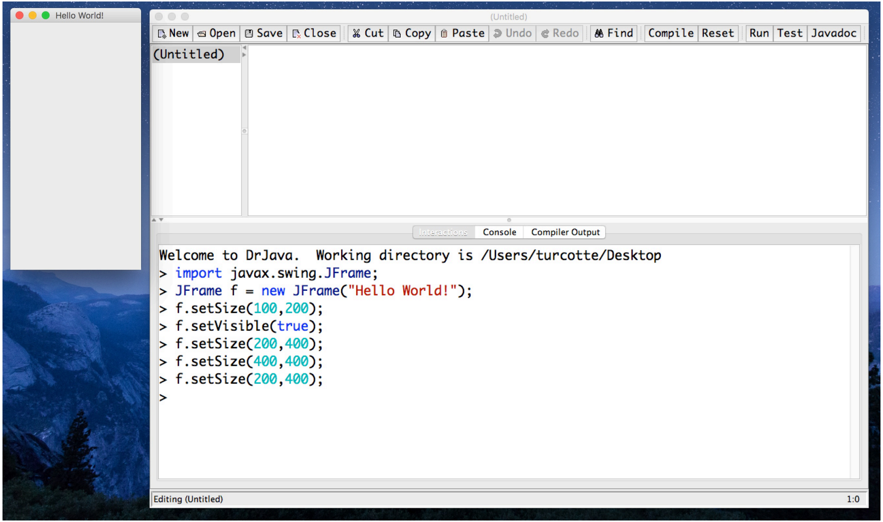
We can also experiment from the interaction window in **DrJava** *. Run the following statements one by one.

```
> 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 will see that the window is not visible at first.

*Alternatively, you can use **jshell**.

DrJava: Hello World (1.0)



```
Welcome to DrJava.  Working directory is /Users/turcotte/Desktop
> import javax.swing.JFrame;
> JFrame f = new JFrame("Hello World!");
> f.setSize(100,200);
> f.setVisible(true);
> f.setSize(200,400);
> f.setSize(400,400);
> f.setSize(200,400);
>
```

Hello World (2.0): An illustration of inheritance

- A **specialized class** of **JFrame** with all the characteristics required for this application.
- The **constructor** is responsible for determining the initial appearance of the window.

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

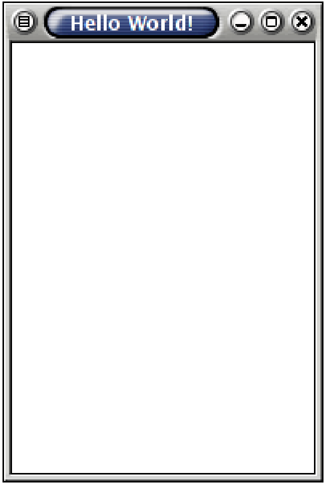
Hello World (2.0)

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

that we use like this:

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

Hello World (2.0)

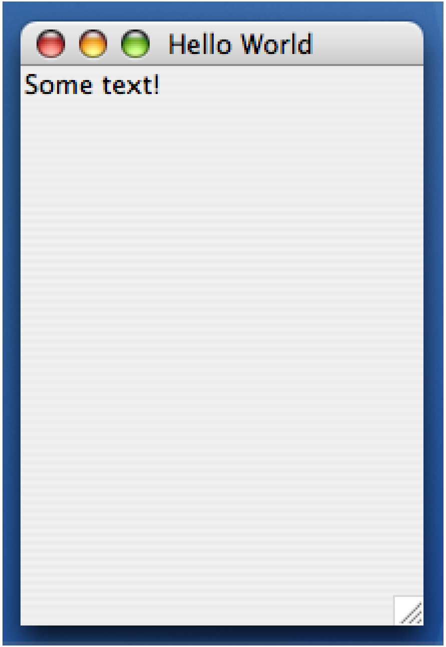


Adding graphic elements

MyFrame is a specialization of the class **JFrame**, which is itself a specialization of the class **Frame**, which specializes the class **Window**, which itself specializes **Container**. Thus, **MyFrame** can contain other graphical elements.

```
import javax.swing.*;

public class MyFrame extends JFrame {

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

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

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

Which method **add** is that?



Some text!

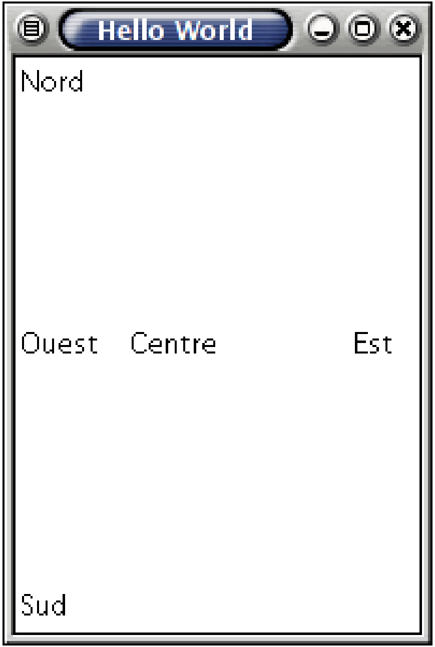
LayoutManager

LayoutManager

- When adding graphical elements, you want to **control their layout**.
- We call **layout manager**, the object that controls the layout and size of objects in a container.
- **LayoutManager** is a **interface** and Java provides over 20 implementations for it. The main classes are:
  - **FlowLayout** adds the graphical elements from left to right and top to bottom; this is the default manager for **JPanel** (the simplest of the containers).
  - **BorderLayout** divides the container into 5 zones: north, south, east, west and center, the default for the class **JFrame**.
  - **GridLayout** divides the container into $m \times n$ zones.

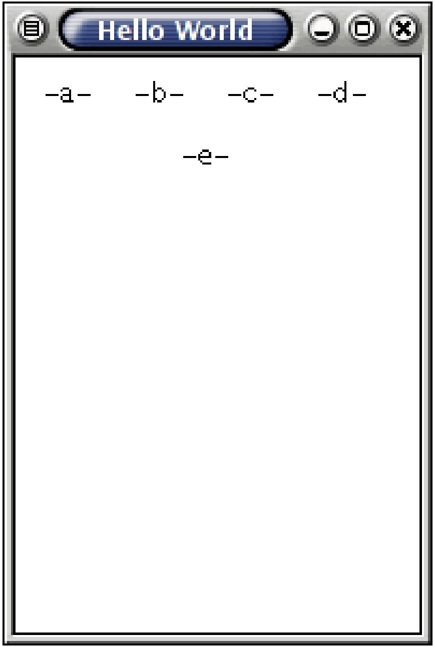
BorderLayout

```
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);
    }

}
```



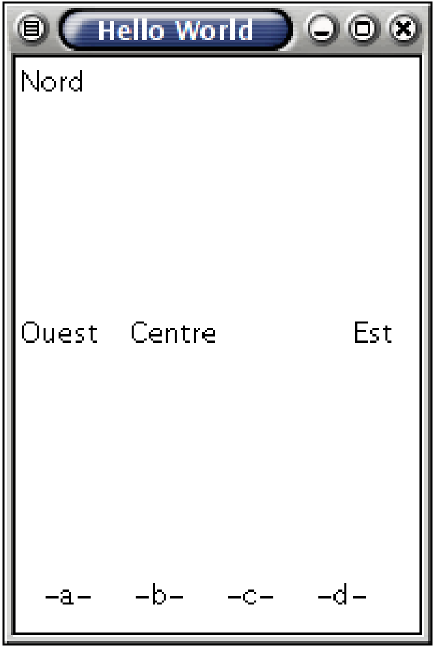
-a- -b- -c- -d-

-e-

JPanel: create complex visual renderings

- The class **JPanel** defines the simplest container.
- A **JPanel** is used to group together several graphical elements and associate them with a layout manager.

```java
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-

Event-oriented programming

Event-oriented programming

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

- Graphical applications are programmed in a **paradigm** that differs from other types of applications.
- The application is almost always **waiting for an action** from the user; click on a button for example.
- An **event is an object** that represents the user's action within the graphical application.

Event-oriented programming

- In Java, **the graphical elements (Component) are the source of the events.**
- An object is said to either generate an event or be the **source** of one.
- When a button is pressed and released, AWT sends an instance of the class **ActionEvent** to the button, through the **processEvent** method of the object of the class **JButton**.

Callback methods (functions)

- How do you **associate actions** with graphical elements?
- Let's put ourselves in the shoes of the person in charge of the Java **JButton** class implementation.
- When the button is pressed and released, the button will receive an **ActionEvent** object, via a call to its **processEvent(ActionEvent e)** method.
- **What to do?**
- We'd have to **make a call to a method of the application**. That method will do the necessary processing.
- What concept can we use in order to **force the programmer to implement a method with a well-defined signature**? (A specific name, a specific list of parameters)

ActionListener

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

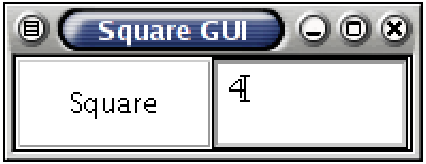
```
public interface ActionListener extends EventListener {

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

    public void actionPerformed(ActionEvent e);

}
```

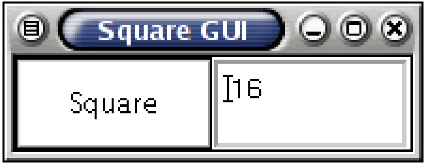
Answering machine analogy

- We are still in the skin of the programmer of the Java **JButton** class implementation.
- Our strategy will be the following: let's ask the application to leave us its "coordinates" (**addListener**) and we will call it back (**actionPerformed**) when the button has been pressed.
- The button's **addListener(. . .)** method allows an object to register as a listener:
  - "when the button has been pressed, call me"
- What is the parameter type of the **addListener(. . .)** method?
- Um, how will you interact with this listener?
- Its method **actionPerformed(ActionEvent e)**!
- This object will have to implement the **ActionListener** interface!

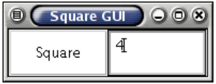
Example: Square

In order to better understand, we will create a small application displaying **the square of a number**!





Here are the declarations necessary to create the **graphical aspect** of the 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);
    }
}
```

Doing the work!

- The class **JTextField** has a method **getText()**, which we will use to obtain the user's string.
- As well as a method **setText(String)**, which we will use to replace the user's string by its square.

So this is the content of the **square** method:

```
    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));
    }
}
```

What’s missing from our application?

- What’s **missing** from our application?
- The application must know to **call the method square** when the button is pressed!

```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

Summary

- The classes of **AWT** and **Swing** are organized hierarchically (inheritance).
- The placement of the graphical elements is under the control of a manager, an object that realizes the **LayoutManager** interface.
- Graphical user applications are programmed according to the **event driven** programming model.
    - A class must realize the interface **ActionListener**
    - This class must implement the method **actionPerformed**.
    - The reference of an object whose class realizes the **ActionListener** interface is provided to the button via the method **addActionListener**.

Next module

- **Parameterized** types (« *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

School of Electrical Engineering and **Computer Science** (EE**CS**)

University of Ottawa