IV. Conception d'applications interactives

Evènements

- L'utilisateur interagit avec les composants de l'interface au moyen d'*évènements*

- Il s'agit d'une manière particulière de programmer :
  - L'utilisateur produit un événement (clic souris, ouverture menu…)
  - Le système réagit à cet événement

Préliminaire : interaction avec l'utilisateur (exemple en langage C)

```
void interaction(){
      int choix;
      while (1){
            scanf("%d", &choix);
            switch (choix){
            case 1 : f1(); break;
            case 2 : f2(); break;

            case -1 : exit();
            }
}
```

Le programme reste bloqué en attendant la saisie de l'entrée

Préliminaire : interaction avec l'utilisateur (exemple en langage C)

```
void f1(){...}
void f2(){...}
```

```
void interaction main(){
 int choix;
 void (*f[N])(void);
 f[0] = f1;
 f[1] = f2;
...
 while (1){
  scanf("%d", &choix);
  f[choix-1]();
 }
```

Amélioration : on associe directement la fonction de traitement à l'évènement associé : procédure "callback" avec pointeurs de fonctions.

Programmation guidée par les événements ("callback" avec pointeurs de fonctions)

```
void f1(char x){...}
void f2(char x){...}
```

```
void interaction(){
 int choix;
 char c;
 void (*f[N])(char);
 f[0] = f1;
 f[1] = f2;

 while (1){
  scanf("%d", &choix);
  scanf("%c", &c);
  f[choix-1](c);
 }
```

Procédure "callback" avec paramètres.

Programmation guidée par les événements ("callback" avec pointeurs de fonctions)

```
void f1(char x){...}
void f2(char x){...}
```

```
void (*f[N])(char);

void association(){
f[0] = f1;
 f[1] = f2;
}

void interaction(){
 int choix;
 char c;
 while (1){
  scanf("%d", &choix);
  scanf("%c", &c);
  f[choix-1](c);
 }
}
```

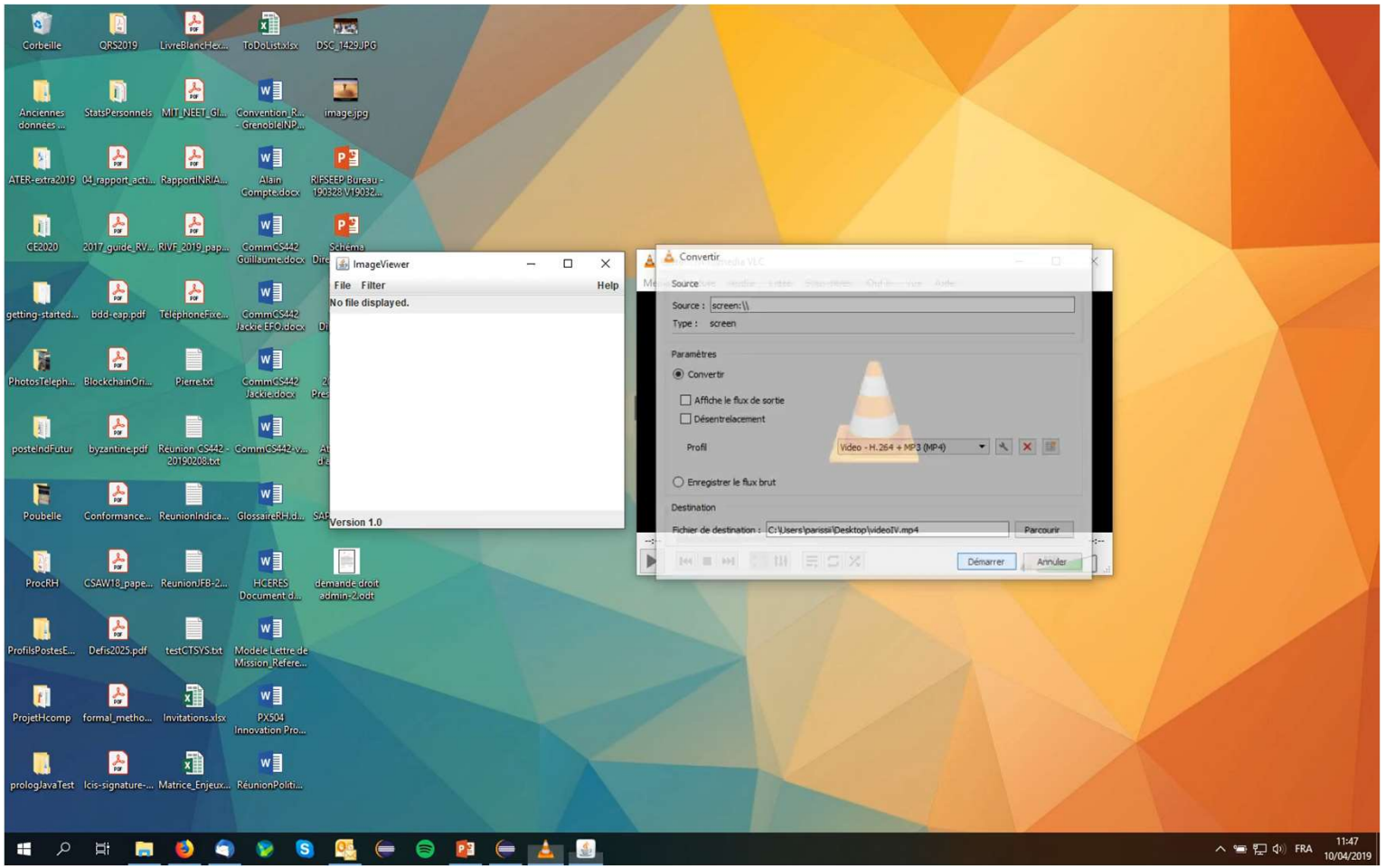
Programmation guidée par les évènements Recapitulatif

- On associe à chaque évènement une fonction de traitement (callback).
  - Les fonctions de traitement ont des paramètres identiques quel que soit l'évènement associé (comme dans l'exemple en C avec tableau de fonctions).
- Chaque évènement utilisateur déclenche l'appel de la fonction de traitement associée.

Programmation guidée par les évènements
modalités d'interaction

- Sur l'exemple précédent
  - Une *modalité* d'interaction en entrée (clavier)
  - Une *modalité* en sortie : écran

- En IHM graphique
  - Plusieurs modalités d'entrée : clavier, souris et plusieurs évènements associés à chaque modalité
    - Clic gauche
    - Clic droit
    - Clic dans une fenêtre
    - Clic sur le bord …
  - Contrairement à l'exemple précédent, l'appel des fonctions de traitement est consécutif à une interruption envoyée par l'évènement (l'application n'est pas bloquée entre deux clic souris, par exemple).

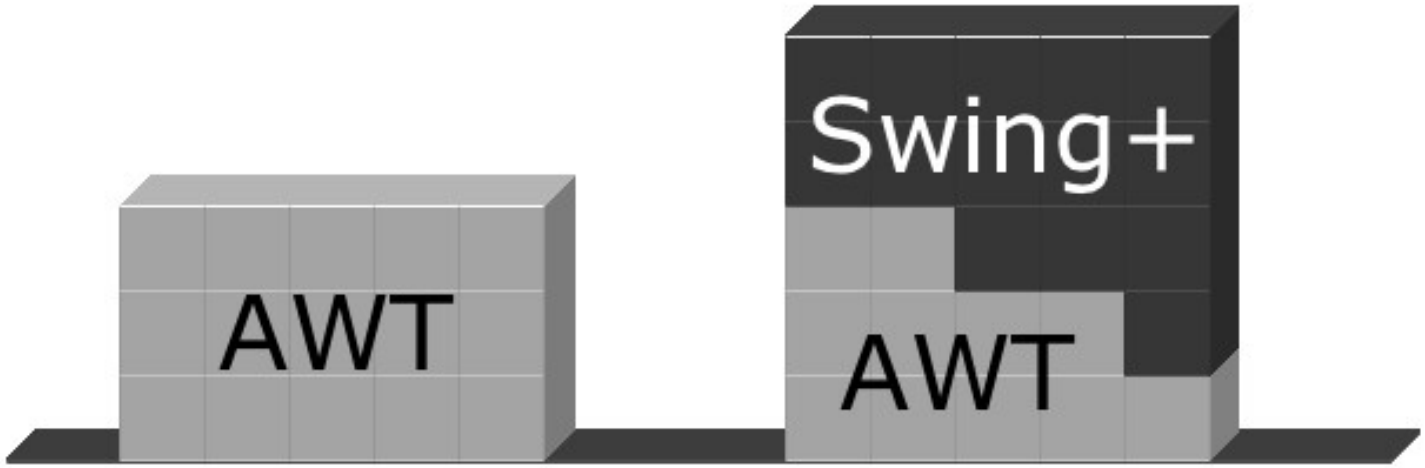

ImageViewer
File Filter Help
No file displayed.
Version 1.0
Convertir
Source
Source : screen:\\
Type : screen
Paramètres
Convertir
Affiche le flux de sortie
Désentrelacement
Profil
Video - H.264 + MP3 (MP4)
Enregistrer le flux brut
Destination
Fichier de destination : C:\Users\parissi\Desktop\videoTV.mp4
Parcourir
Démarrer
Annuler
FRA
11:47
10/04/2019

Construction d'interfaces graphiques

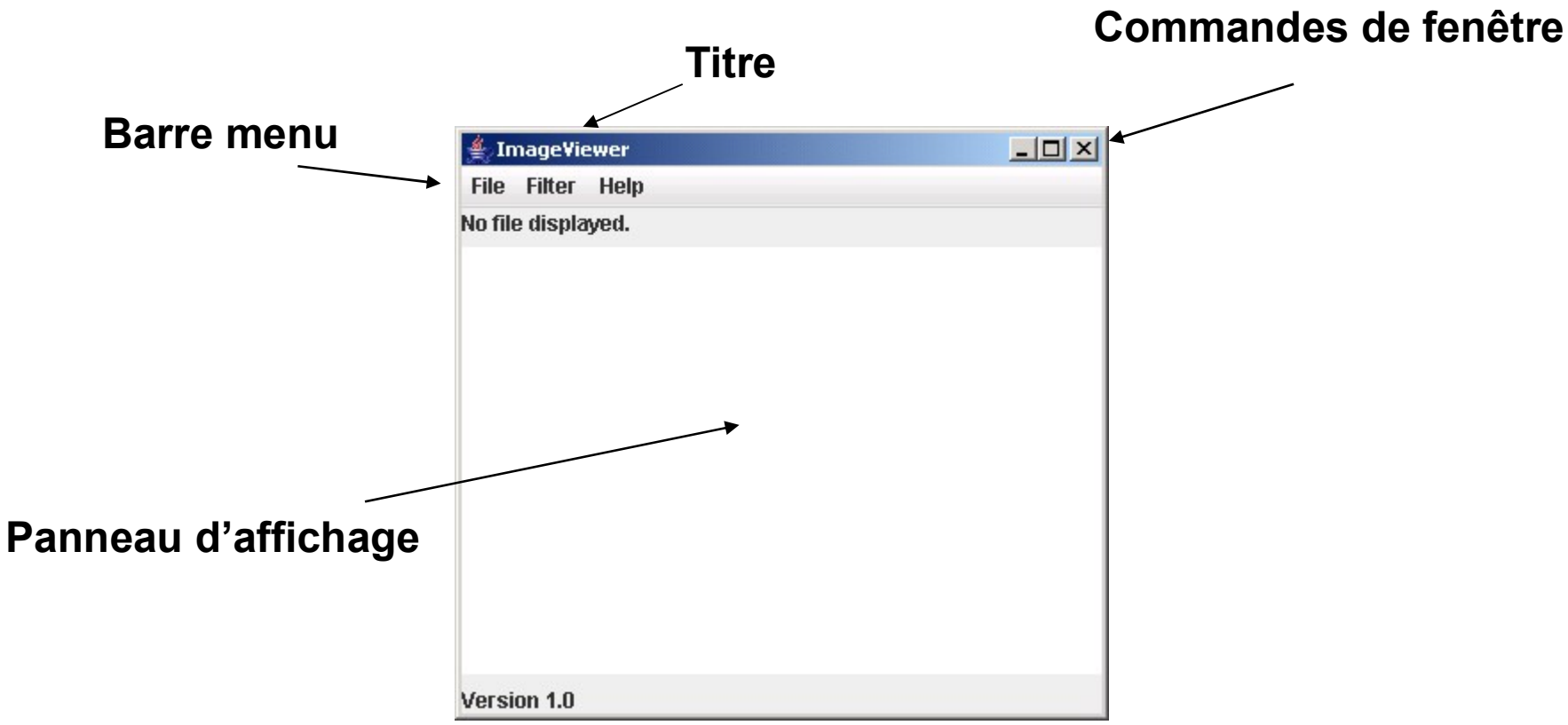
- Une interface graphique est construite à partir de composants de base
  - Boutons, menus, barres de défilement …
  - Plusieurs composants fournis par des librairies java
- Les composants peuvent être organisés de différentes manières
- L'interaction avec l'utilisateur est faite par *événements*
  - "Un bouton a été appuyé", "un menu a été ouvert" …
- Programmation des interfaces graphiques "guidée par les événements"

AWT & Swing

- Librairie swing (javax.swing), construite sur la base de Abstract Window Toolkit (java.awt)



Exemple : un afficheur d'images



Exemple : construction du cadre

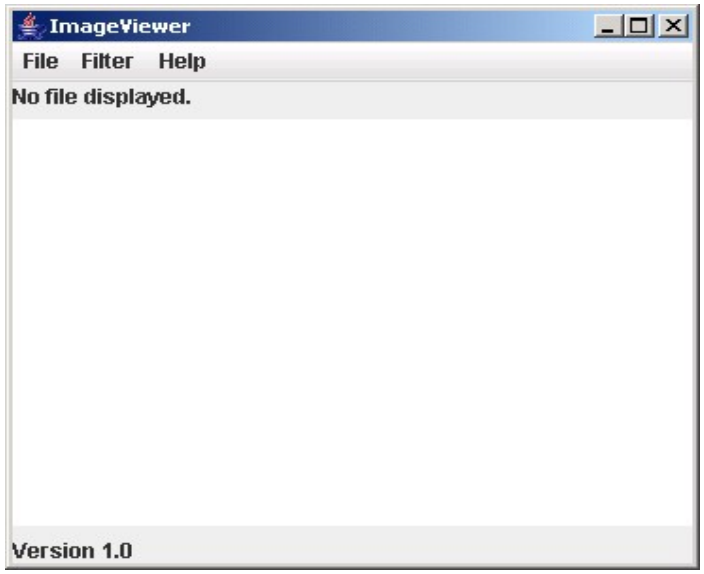
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ImageViewer
{
    private JFrame frame;

    /**
     * Create an ImageViewer show it on screen.
     */
    public ImageViewer()
    {
        makeFrame();
    }

    // rest of class omitted.
}
```

Exemple : panneau d'affichage

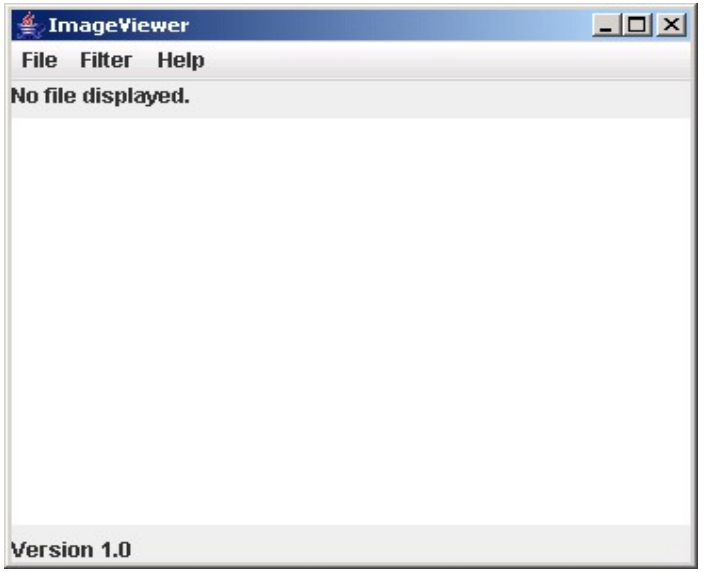
```
/**
 * Create the Swing frame and its content.
 */
private void makeFrame()
{
    frame = new JFrame("ImageViewer");
    Container contentPane = frame.getContentPane();

    JLabel label = new JLabel("I am a label.");
    contentPane.add(label);

    frame.pack();
    frame.setVisible(true);
}
```

Construction de menus

- `JMenuBar`
  - Est affiché sous le titre de la fenêtre
- `JMenu`
  - (p.ex. *File*). Correspond à un menu déroulant.
- `JMenuItem`
  - (p.ex. *Open*) Items des menus



```
private void makeMenuBar(JFrame frame)
{
    JMenuBar menubar = new JMenuBar();
    frame.setJMenuBar(menubar);

    // create the File menu
    JMenu fileMenu = new JMenu("File");
    menubar.add(fileMenu);

    JMenuItem openItem = new JMenuItem("Open");
    fileMenu.add(openItem);

    JMenuItem quitItem = new JMenuItem("Quit");
    fileMenu.add(quitItem);
}
```



Java et swing

- Il existe différents types d'évènement par composant
  - **`WindowEvent`** pour les cadres
  - **`ActionEvent`** pour les menus
- Un objet peut recevoir une notification quand survient un évènement
  - Objet *listener*
- Comment récupérer un évènement?
  - Comment associer un objet 'listener' à un évènement?

`ActionListener` est une interface possédant une seule mét
`actionPerformed`

Toute classe implémentant `ActionListener` peut recevoi
notifications.

Exemple ImageViewer
gestion centralisée des évènements

```
public class ImageViewer implements
ActionListener
{
    …
    public void actionPerformed(ActionEvent e)
    {
        String command = e.getActionCommand();
        if(command.equals("Open")) {
            …
        }
        else if (command.equals("Quit")) {
            …
        }
        …
    }
    …
    private void makeMenuBar(Jframe frame)
    {
        …
        openItem.addActionListener(this);
        quitItem.addActionListener(this);
        …
    }
}
```

- `ImageViewer` implémente `ActionListener`
- La méthode `actionPerformed` est appelée à chaque fois qu'un événement survient; cet événement est passé en paramètre.
- Un seul objet gère tous les événements
- L'objet gestionnaire des événements s'enregistre auprès des composants.
  - **`item.addActionListener(this)`**
  - En cas d'événement, actionPerformed sera appelé sur this

Gestion centralisée : inconvénients (cf. exemple en C vu plus haut)

```
public class ImageViewer implements
ActionListener
{
    …
    public void actionPerformed(ActionEvent e)
    {
        String command = e.getActionCommand();
        if(command.equals("Open")) {
            …
        }
        else if (command.equals("Quit")) {
            …
        }
        …
    }
    …
    private void makeMenuBar(Jframe frame)
    {
        …
        openItem.addActionListener(this);
        quitItem.addActionListener(this);
        …
    }
}
```

- Une seule méthode actionPerformed est définie
  - Elle sera appelée pour tous les événements
  - Elle doit déterminer quel composant est concerné par l'événement
- Inconvénients
  - Si beaucoup de composants, problème de performance
  - L'identification des composants par leur nom ("Quit") n'est pas sûre.

Autre solution : classes internes, classes anonymes

- Une classe peut être définie dans une autre classe

```
public class Enclosing
{
     …
     private class Inner
     {
          …
     }
}
```

- Les instances de classes internes (*inner classes*)
  - n'existent que dans la classe englobante.
  - ont accès aux attributs privés des classes englobantes.
- Une classe interne peut être anonyme :

Classes internes *anonymes*

```
JMenuItem openItem = new JMenuItem("Open");

openItem.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
         openFile();
     }
});
```

Listener anonyme : explication

```
openItem.addActionListener(
    new ActionListener()
    {
          public void actionPerformed(ActionEvent e)
          {
              openFile();
          }
    }
);
```

Création d'objet anonyme

Définition de classe (anonyme)

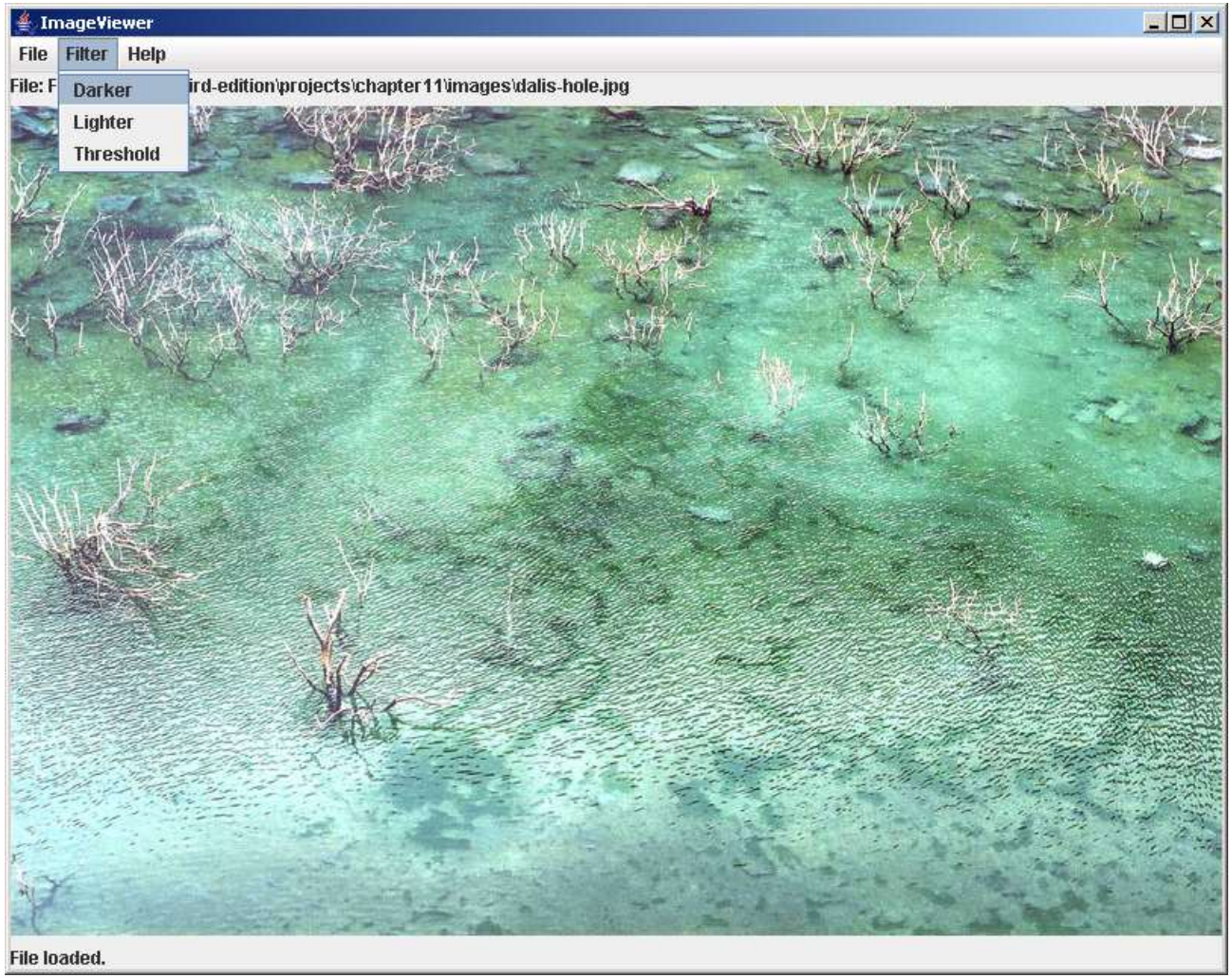
Paramètre de openItem.addActionListener

Exemple : sortie de l'application

```
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
});
```

WindowAdapter fournit une implementation de l'interface WindowListener.

Traitement d'image



Responsabilités des classes

- `ImageViewer`
  - Mise en place de la structure du GUI.
- `ImageFileManager`
  - Méthodes statiques pour le chargement et l'enregistrement des images.
- `ImagePanel`
  - Affiche l'image à l'intérieur du GUI.
- `OFImage`
  - Modélise une image 2D.

OFImage

- Notre sous-classe de `BufferedImage`.
- Représente un tableau 2D de pixels.
- Méthodes importantes:
  - `getPixel`, `setPixel`
  - `getWidth`, `getHeight`
- Chaque pixel possède une couleur.
  - Nous utilisons `java.awt.Color`.

Ajout de *ImagePanel*

```
public class ImageViewer
{
    private JFrame frame;
    private ImagePanel imagePanel;

    …

    private void makeFrame()
    {
        Container contentPane = frame.getContentPane();
        imagePanel = new ImagePanel();
        contentPane.add(imagePanel);
    }

    …
}
```

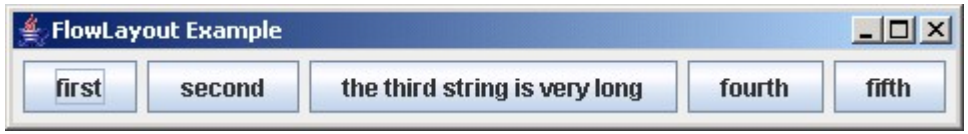
Chargement d'une image

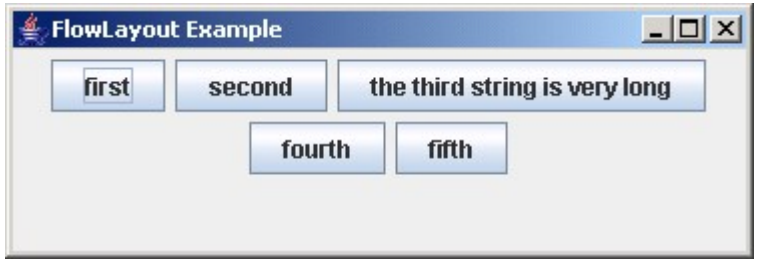
```
public class ImageViewer
{
    private JFrame frame;
    private ImagePanel imagePanel;

    …

    private void openFile()
    {
        File selectedFile = …;
        OFImage image =
            ImageFileManager.loadImage(selectedFile);
        imagePanel.setImage(image);
        frame.pack();
    }

    …
}
```

Gestionnaires de disposition

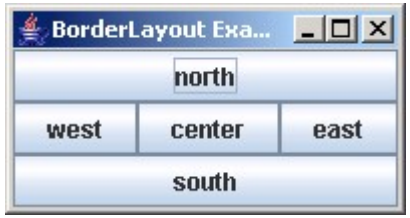
- Gère l'espace limité pour l'emplacement des composantes.
  - `FlowLayout`, `BorderLayout`, `GridLayout`, `BoxLayout`, `GridBagLayout`.
- Gère des objets `Container` , c'est-à-dire des zones de contenu.
- Chacun impose son propre style.

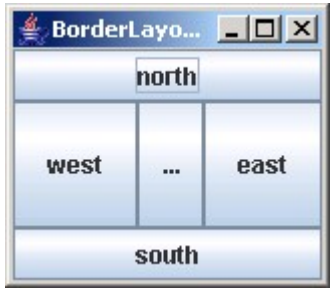
FlowLayout

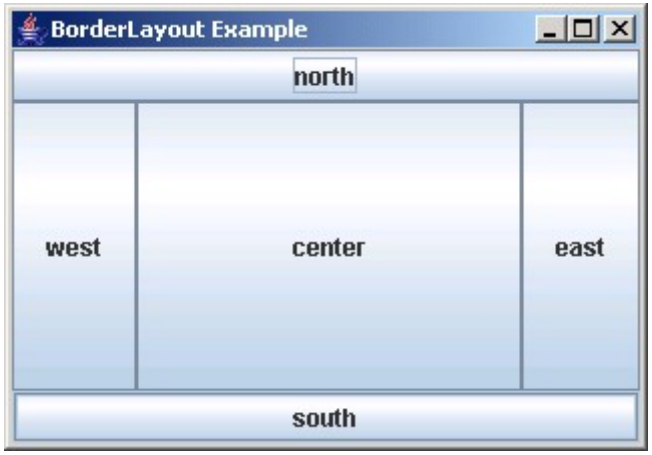

FlowLayout Example
first
second
the third string is very long
fourth
fifth


FlowLayout Example
first
second
the third string is very long
fourth
fifth

BorderLayout

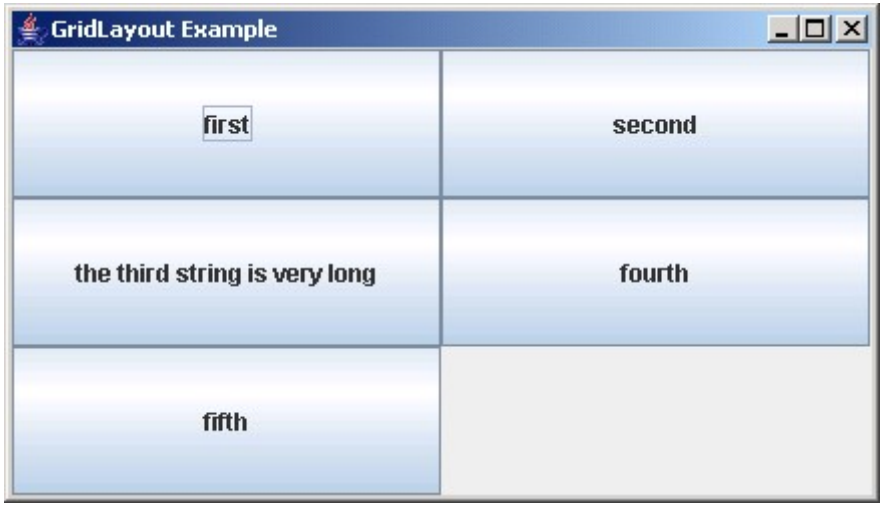




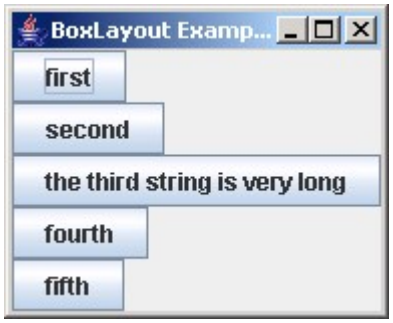


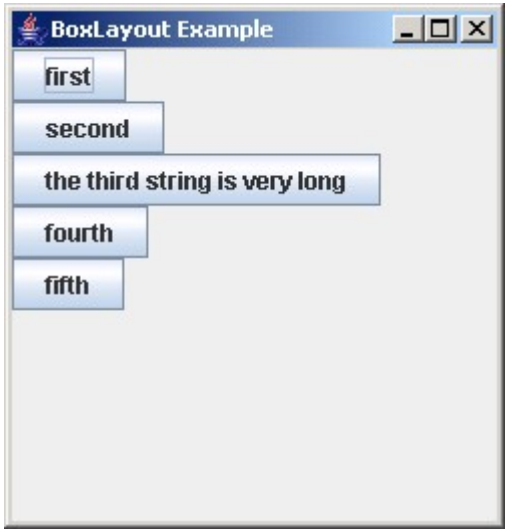
GridLayout


GridLayout Example
first
second
the third string is very long
fourth
fifth


GridLayout Example
first
second
the third string is very long
fourth
fifth

BoxLayout





Note: pas de redimensionnement des composantes.

Conteneurs imbriqués

- Des dispositions sophistiquées peuvent être obtenues en imbriquant des conteneurs.
  - Utiliser `JPanel` comme conteneur de base.
- Chaque conteneur aura son propre gestionnaire de disposition.
- Souvent préférable d'utiliser `GridBagLayout`.

Boîtes de dialogue

- Les boîtes de dialogue modales bloquent toute autre interaction.
  - Force une réponse de la part de l'utilisateur.
- Les boîtes de dialogue non-modales permettent d'autres interactions.
  - C'est parfois désirable.
  - Peut être difficile d'éviter les incohérences.

Boîtes de dialogues standard de JOptionPane

- Boîte de dialogue de Message
  - Message texte plus bouton OK.
- Boîte de dialogue de Confirmation
  - Options Yes, No, Cancel
- Boîte de dialogue de saisie (Input)
  - Message texte et champ de saisie.
- Des variantes sont possibles.

Une boîte de dialogue de Message

```
private void showAbout()
{
    JOptionPane.showMessageDialog(frame,
                "ImageViewer\n" + VERSION,
                "About ImageViewer",
                JOptionPane.INFORMATION_MESSAGE);
}
```

Filtres d'images

- Fonctions appliquées à l'image entière.

```
int height = getHeight();
int width = getWidth();
for(int y = 0; y < height; y++) {
    for(int x = 0; x < width; x++) {
        Color pixel = getPixel(x, y);
        alter the pixel's color value;
        setPixel(x, y, pixel);
    }
}
```

Ajout d'autres filtres

```
private void makeLighter()
{
    if(currentImage != null) {
        currentImage.lighter();
        frame.repaint();
        showStatus("Applied: lighter");
    }
    else {
        showStatus("No image loaded.");
    }
}

private void threshold()
{
    if(currentImage != null) {
        currentImage.threshold();
        frame.repaint();
        showStatus("Applied: threshold");
    }
    else {
        showStatus("No image loaded.");
    }
}
```

Ajout d'autres filtres

- Définir une superclasse `Filter` (abstraite).
- Créer des sous-classes pour les fonctions spécifiques.
- Créer une collection d'instances de ces sous-classes dans `ImageViewer`.
- Définir une méthode générique `applyFilter`.
- Voir *imageviewer2-0*.

Boutons et dispositions imbriquées

Un GridLayout dans un FlowLayout dans un BorderLayout.

Bordures

- Utilisées pour ajouter de la décoration autour des composantes.

- Définies dans `javax.swing.border`
  - `BevelBorder`, `CompoundBorder`, `EmptyBorder`, `EtchedBorder`, `TitledBorder`.

Ajout d'espacements

```
JPanel contentPane = (JPanel)frame.getContentPane();
contentPane.setBorder(new EmptyBorder(6, 6, 6, 6));

// Specify the layout manager with nice spacing
contentPane.setLayout(new BorderLayout(6, 6));

imagePanel = new ImagePanel();
imagePanel.setBorder(new EtchedBorder());
contentPane.add(imagePanel, BorderLayout.CENTER);
```

A retenir

- Viser des structures d'applications avec beaucoup de cohésion.
  - Garder le plus possible les éléments du GUI séparés des fonctionnalités de l'application.
- Les composantes prédéfinies simplifient la création d'interfaces graphiques sophistiquées.
- Les gestionnaires de disposition (Layout managers) gèrent la juxtaposition des composantes.

Bilan des concepts introduits

- Gestion des évènements utilisateur
- Interface graphique en Java, Swing et AWT
  - Concepts et classes/interfaces de base
  - Réalisation de “callback” : choix de realisation des objets “listeners”

V. Conception de classes et interfaces génériques (généricité)

Classes génériques

- Nous avons déjà utilisé la classe générique `ArrayList<Type>`
- `Type` désigne toute classe existante
- Ainsi, on paramètre la définition d'une classe par un type.

Définition d'une classe générique simple

```
/**
* modélise les fonctions d'une cellule
capable
* d'héberger une information
* @author P.Morat
*/
public class Cellule<T> {
  /**
  * la valeur hébergée
  */
  private T valeur;
  /**
  * Affecte la cellule avec une information
  * @param v l'information à stocker
  */
  public void set(T v) {
    valeur = v;
  }
  /**
   * restitue l'information contenue dans la
cellule
   * @return l'information contenue
  */
  public T get() {
    return valeur;
  }
}
```

```
/* Utilisation de la classe Cellule */

public class Essai {
  private Cellule<Integer> cellule;
  public void m() {
    cellule = new Cellule<Integer>();
    cellule.set(new Integer(3));
    Integer i = cellule.get();
...

}
```

Exemple d'interface générique avec deux paramètres

```
public interface MyMap<K,V> {
  public V put(K key, V value);
  public V get(K key);
}
```

Avantages de la généricité

- On code une seule fois une classe qui peut être réutilisée de plusieurs manières différentes (ArrayList<Person>, ArrayList<Vehicle>…)
- On peut, lors de la conception d'une classe, s'abstraire des détails inutiles
  - ArrayList<T> est réalisée sans tenir compte du type T

Généricité contrainte

- Pourquoi?
  - La généricité telle qu'on l'a vu jusqu'ici peut être trop permissive
  - `Cellule<T>` : quand on réalise la classe cellule, on est obligé de voir `T` comme `Object`.
  - En d'autres mots, on ne peut pas choisir la nature de l'information sur T dont on s'abstrait.
  - La généricité contrainte peut répondre (au moins en partie) à cette problématique

Généricité contrainte
Exemple

```
/**
* modélise les fonctions d'une liste ordonnée sans doublon.
* @author P.Morat
*/
public class SortedList<T extends Comparable> {
  private List<T> l;

  public void insert(T v) {
    if(l.size() == 0){l.add(v); return;}
    ListIterator it = l.listIterator();
    Comparable e;
    for(e = it.next();
       it.hasNext() && e.compareTo(v) < 0;
       e = it.next());
    if(e.compareTo(v) > 0) {
      it.previous();
      it.add(v);
    }
  }
}
```

Cet exemple montre l'intérêt de la généricité contrainte en mettant en évidence la nécessité de connaître certaines propriétés du type T.

La classe SortedList permet de construire et gérer des listes dans lesquelles les éléments sont rangés selon la *relation d'ordre* qui les caractérise. Les éléments sont donc supposés être d'un type T héritant de `Comparable`.

Interface multi-générique contrainte

```
public interface MyMap<K extends comparable,V> {
  public V put(K key, V value);
  public V get(K key);
}
```

Héritage et généricité

- On peut étendre une classe générique par une autre classe générique, à condition que les variables de type soient les mêmes

```
public class Pair <T> {
   ...
}
public class Triplet <T> extends Pair <T> {
   T three;
   public Triplet (T one, T two, T three){
      super(one,two);
      this.three = three;
   }
   public void setThird(T three){this.three = three;}
   public T getThird(){return this.three;}
}
```

Méthodes génériques

- Une méthode peut être paramétrée par un type, qu'elle soit dans une classe générique ou non
- L'appel de la méthode nécessite de l'instancier par un type, sauf si le compilateur peut réaliser une *inférence de type*

```
public class MyClass{
  public static <T> void permute(T[] tab, int i, int j){
    T temp = tab[i];
    tab[i] = tab[j];
    tab[j] = temp;
  }
}
...
String[] tabs = new String[]{"toto","titi","tutu"};
Float[] tabf = new Float[]{3.14f,27.12f};
MyClass.<String>permute(tabs,0,2);
MyClass.permute(tabf,0,1);
```

Bilan des concepts introduits

- Généricité simple
- Multi-généricité
- Généricité contrainte
- Méthodes génériques