Développement Logiciel
L2-S4

Interface Graphique
Composent graphique, événement

anastasia.bezerianos@lri.fr

Les transparents qui suivent sont inspirés du cours de Basé sur :

- le cours du Nicolas Bredeche (Univ. Paris-Sud)

- le cours d'Alexandre Allauzen (Univ. Paris-Sud)

- Bruce Eckel, "Thinking in Java"

Plan

0. Info
1. Interface Graphique
2. Evénements
3. Manipulation d'images (et prochain cours)

Interface graphique, IG, IHM, ...

- Une représentation graphique (visuelle) de :
  - l'information
  - l'interaction entre l'homme est la machine

- Java : un langage indépendant de la plateforme
- Une API pour les interfaces graphiques indépendantes de la plateforme ?
  - aspect graphique : classes et interface pour « dessiner » l'information
  - aspect interaction : gérer les événements d'utilisateur

Java et fenêtre

- Chaque plateforme a son système de gestion d'interface utilisateur : *GUI : Graphical User Interface systems*
  - Linux XWindows
  - Mac OS Quartz
  - Microsoft Windows GDI

  - boîte à outils d'interface : offre une bibliothèque d'objets interactifs (les widgets) que l'on assemble pour construire l'interface.
    - Exemple : Java Swing, Qt (C++), Gtk (C++), ...
    - une langue de programmation, mono/multi platform

Java et fenêtre

- L'API Java doit communiquer avec le GUI cible via des « Adaptateurs» (entre GUI et boîte à outils),

- mais avec quelle stratégie ?
  - faire une utilisation **maximale** du système graphique cible (AWT)
  - faire une utilisation **minimale** du système graphique cible (**SWING**)

Utilisation maximale : *java.awt*

- L'objet *TextField* délègue la plupart de ses tâches à un composant natif.
  - Le programmeur java utilise un objet *TextField*
  - L'objet *TextField* délègue à une classe adaptateur dépendante de l'OS : *MotifTextField*, *GTKTextField*, *WindowsTextField*, *MacOSTextField* ...
  - Le système graphique natif réalise le plus gros du travail
- Pour :
  - un aspect et comportement (l*ook and feel*) comme les autres de la plateforme
  - pas besoin de refaire les composants, juste s'adapter
- Contre
  - un catalogue restreint : l'intersection des GUI
  - le comportement et l'aspect dépendent donc de la plateforme

Utilisation minimale : *javax.swing*

- Utiliser les éléments « natifs » pour le strict nécessaire : ouvrir une fenêtre, dessiner des lignes/du texte, gestion primitive des événements
- Tout le reste est assurer par les classes Java : *JTextField*

- Pour :
  - moins de différences entre plateformes
  - plus de liberté pour créer et ajouter des (nouveaux) composants
- Contre :
  - faut « tout faire »
  - les applications Java n'ont pas le même *look and feel* que les autres.
  - un peu plus lent
- Regarder la javadoc, les tutoriaux de SUN avec les démos.

Architecture Swing

Une application = une fenêtre avec des « choses » bien placée.

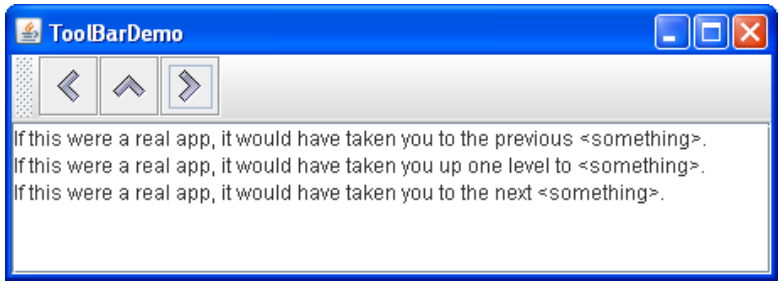
- Un conteneur (*container*) top-level : « LE » conteneur, le composant racine, par exemple la fenêtre.
- Composants atomiques (simples), par ex: un bouton.
- Des composants intermédiaires (composés) qui permettent de diviser la fenêtre : conteneurs pour plusieurs composants, des panneaux.

Un composant graphique doit, pour apparaître, faire partie d'une hiérarchie de conteneur : c'est un arbre avec

- pour feuille des composants atomiques et
- pour racine un *top-level container*.

Un composant ne peut être contenu qu'une seule fois.

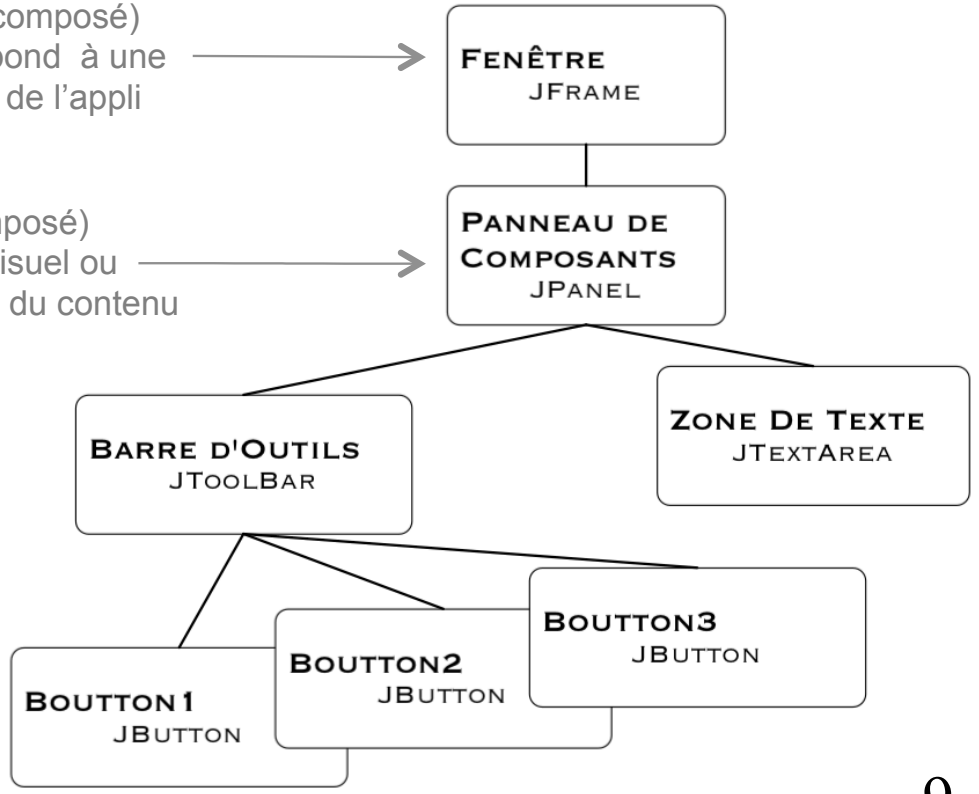
arbre des widgets



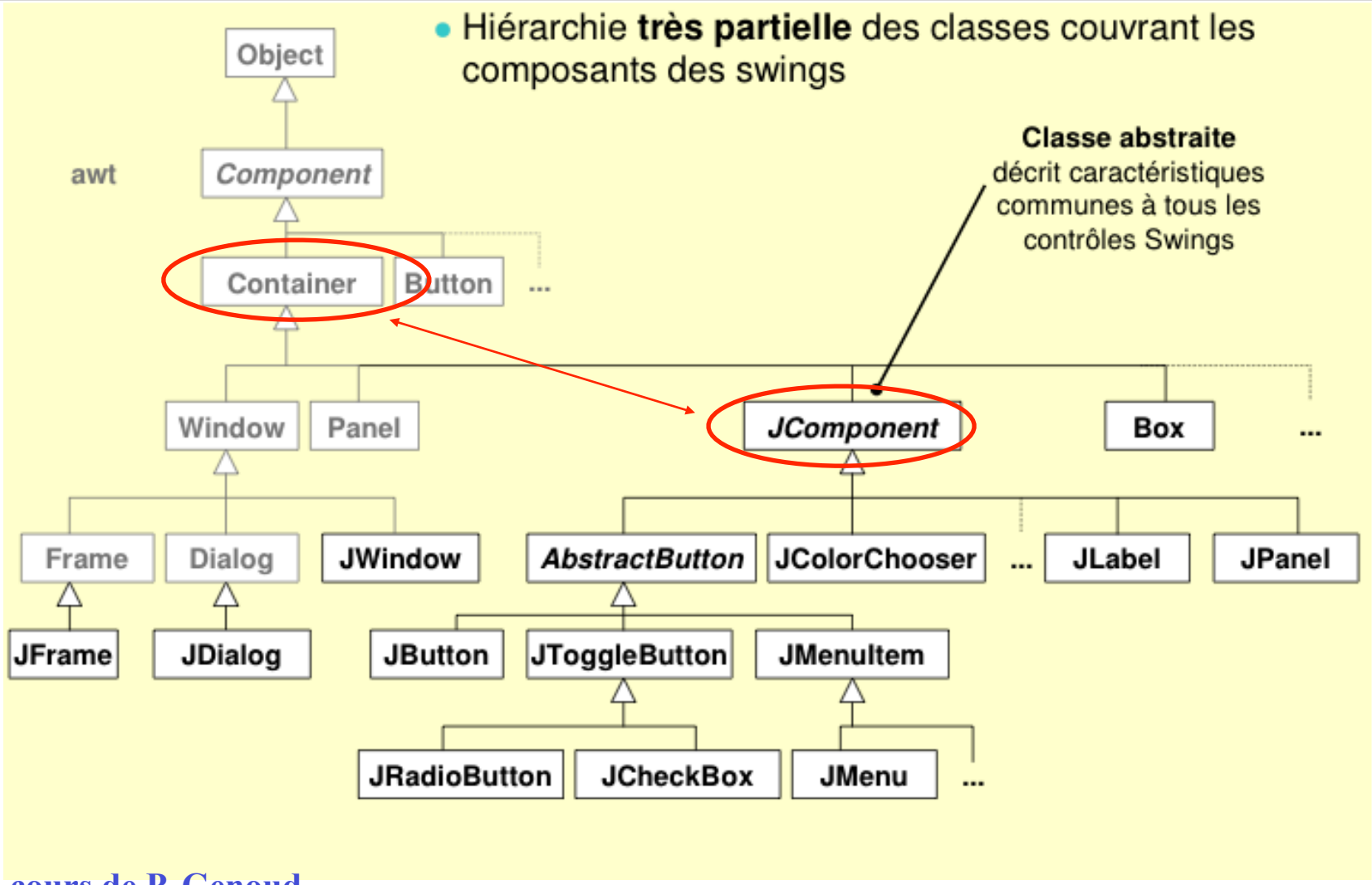
Racine (composé)
correspond à une
fenêtre de l'appli

Nœuds (composé)
Structure visuel ou
fonctionnel du contenu

Feuille (simple)
avec lesquels l'utilisateur
peut interagir



Architecture Swing - version objet

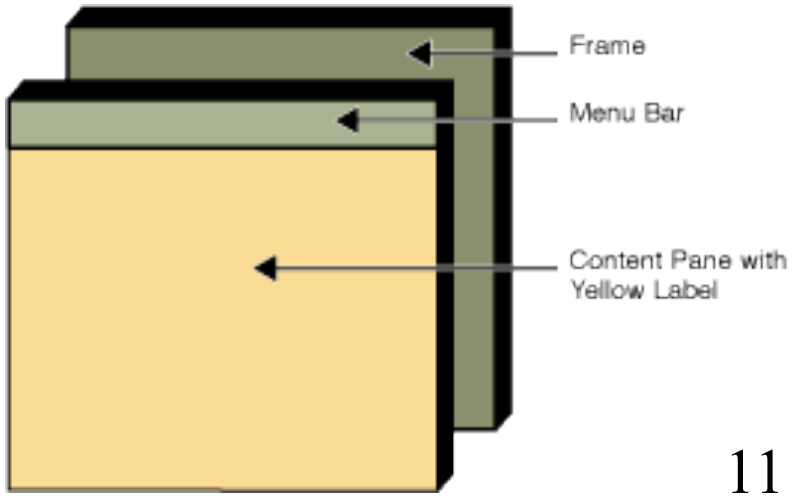


source: cours de P. Genoud

Top-level container : le composant racine

- Une application graphique doit avoir un composant **top-level** comme composant racine (composant qui inclus tous les autres composants).
- Un composant graphique doit pour apparaître faire partie d'une hiérarchie (arbre) d'un conteneur (composant top-level)

- Il en existe 3 types : ***JFrame, JDialog*** *et* ***JApplet***
- C'est un conteneur :
  - il contient d'autres composants dans son champ ***content pane***.

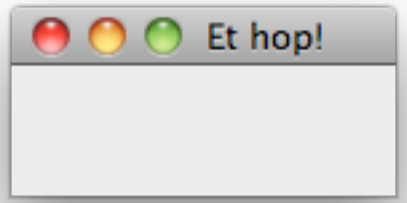


Top-level container : *JFrame*

- Une fenêtre avec une barre de menu.

```
public static void main(String[] args) {
    JFrame jf = new JFrame("Et hop!");
    jf.setVisible(true);
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    System.out.println("c'est terminé ? ! ?");
    System.out.println("pourtant ça marche encore");
}
```

```
"c'est terminé ? ! ?
pourtant ça marche encore
POURQUOI ?
```



public JFrame();
public JFrame(String name);
public Container getContentPane();
public void setJMenuBar(JMenuBar menu);
public void setTitle(String title);
public void setIconImage(Image image);

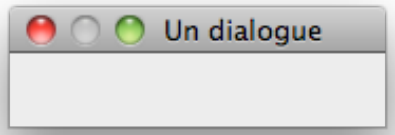
Top-level container : *JDialog*

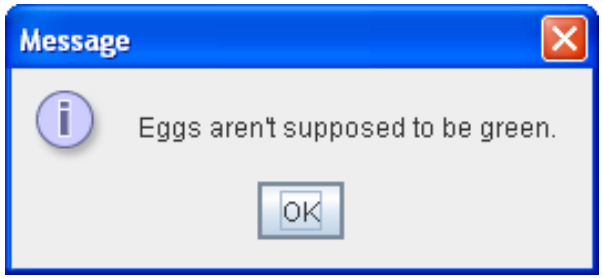
- Une fenêtre pour l'échange d'information, peut être « modale » (bloquant).
- Elle dépend d'une autre fenêtre, si celle-ci est fermée l'objet *JDialog* également.

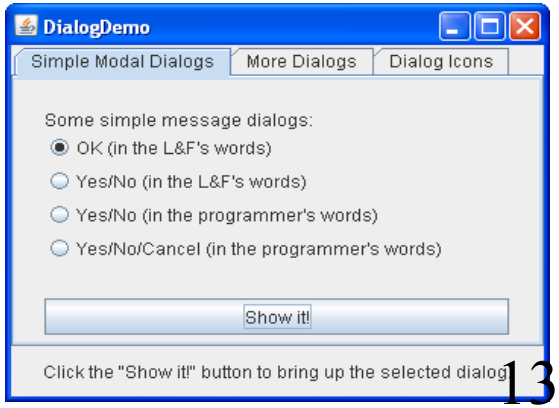
```
public static void main(String[] args) {
    JFrame jf = new JFrame("Et hop!");
    jf.setVisible(true);
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   JDialog jd = new JDialog(jf,"Un dialogue",true);
    jd.setVisible(true);
}
```

modal

dépend de

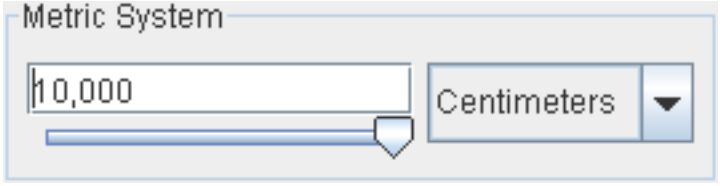


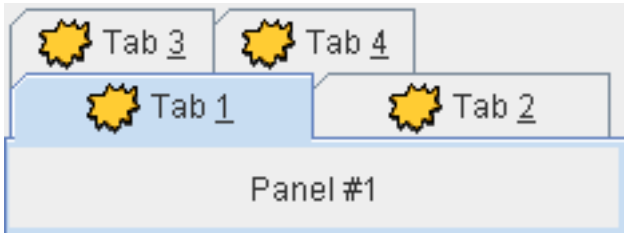


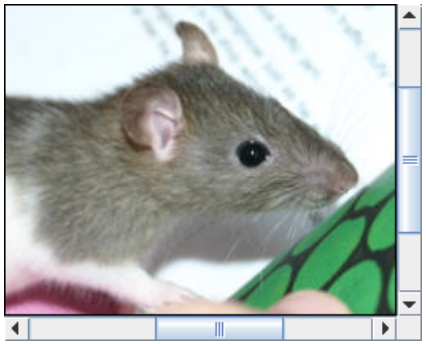


Conteneurs intermédiaires

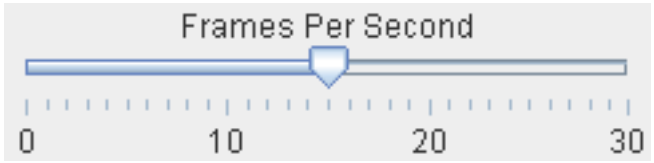
- Les conteneur intermédiaire sont utilisés pour structurer l'application graphique
- Le composant top-level contient des composants conteneurs intermédiaires
- Un conteneur intermédiaire peut contenir d'autres conteneurs intermédiaires
- Les choix de Swing : *JPanel* (le plus simple), *JScrollPane, JSplitPane, JTabbedPane, JToolBar ...*





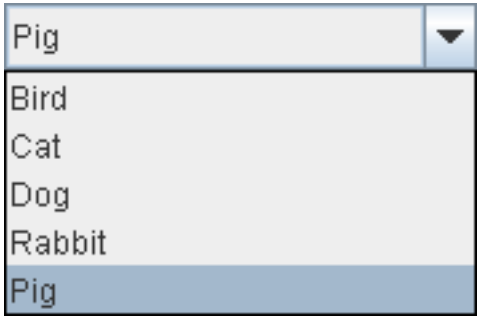


Composants atomiques : les contrôles de bases (interactives, widgets)



JSlider

JButton



JComboBox

JTexField

JCheckBox

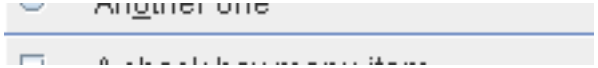
JRadioButton

JList

Les composants non-editables

JLabel

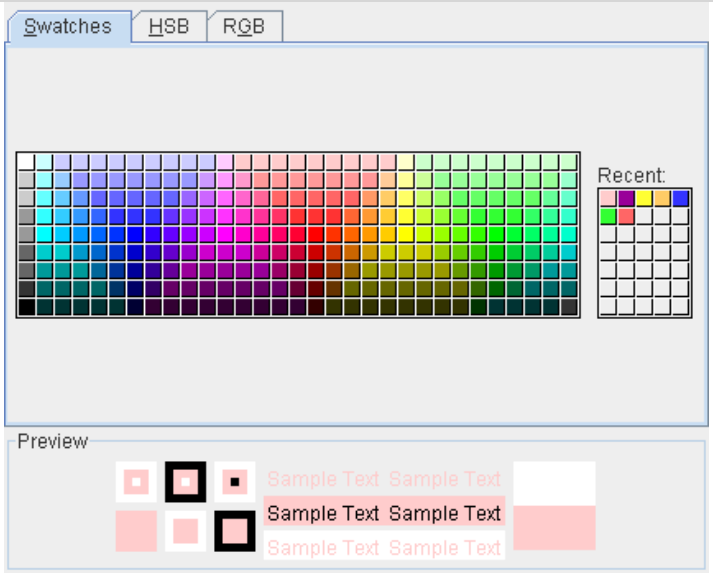
JProgressBar



JSeparator

JToolTip

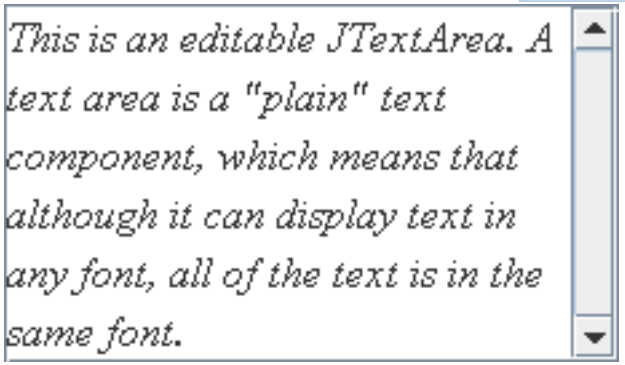
Plus compliqués

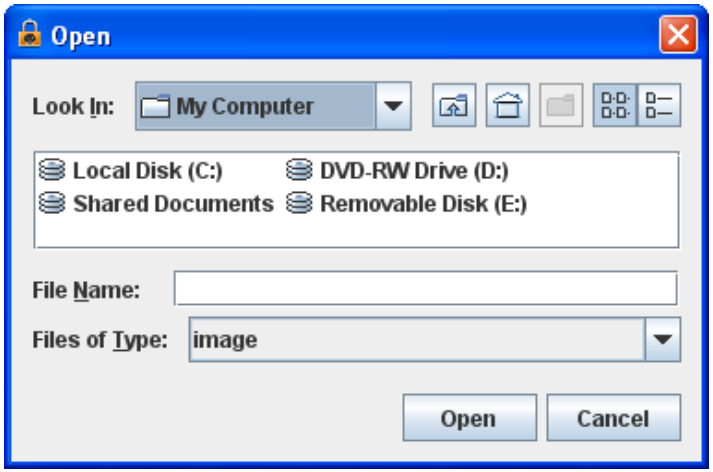


JColorChooser

JEditorPane and JTextPane

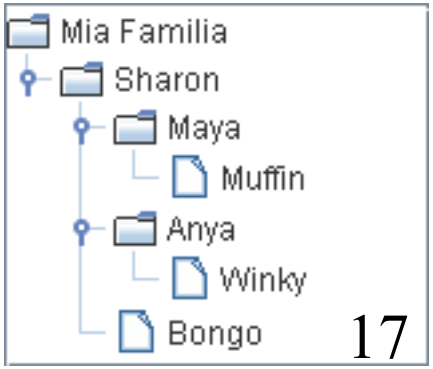
JTextArea





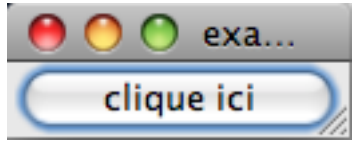
JFileChooser

JTree



Les composants Swing

- http://docs.oracle.com/javase/tutorial/ui/features/components.html

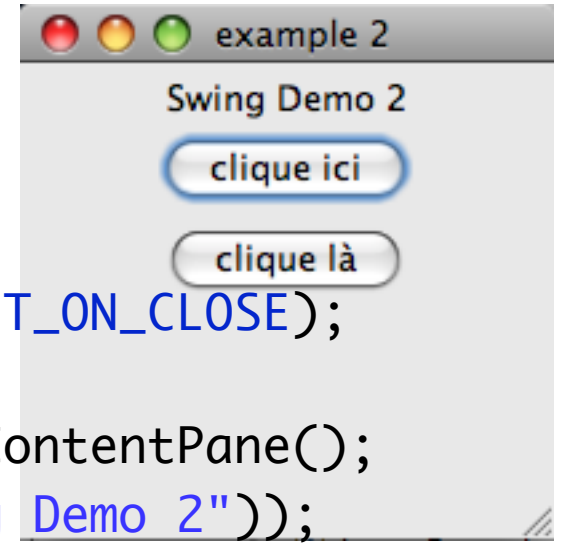


ou est le Label?

```
import javax.swing.*;

public class SwingDemo1 {

  public static void main(String[] args) {
    JFrame frame = new JFrame();

    frame.setTitle("example 1");

    frame.setDefaultCloseOperation
                (javax.swing.JFrame.EXIT_ON_CLOSE);

    frame.getContentPane().add(new JLabel("Swing Demo 1"));
    frame.getContentPane().add(new JButton("clique ici"));

    frame.setSize(100,50);
    frame.setVisible(true);
  }
}
```


Inspiré de: Bruce Eckel, Thinking in Java, 2e édition

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

public class SwingDemo2 extends JFrame {
        public void init() {
            this.setTitle("example 2");
            this.setDefaultCloseOperation(EXIT_ON_CLOSE);

            Container contentPane = this.getContentPane();
            contentPane.add(new JLabel("Swing Demo 2"));
            contentPane.setLayout(new FlowLayout());
            contentPane.add(new JButton("clique ici"));
            contentPane.add(new JButton("clique là"));
        }

        public static void main(String[] ar
                JFrame frame = new SwingDemo2
                ((SwingDemo2)frame).init();
                frame.setSize(200,200);
                frame.setVisible(true);
        }
}
```

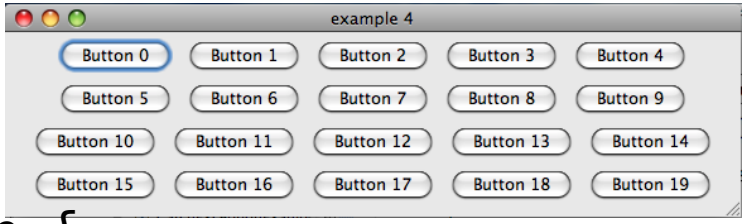


déplacer le contenu de init() dans le constructeur

(transps suivants aussi)

Inspiré de: Bruce Eckel, Thinking in Java, 2e édition

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

public class SwingDemo4 extends JFrame {
    public void init() {
        Container cp = getContentPane();
        this.setTitle("example 4");
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);

        cp.setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            cp.add(new JButton("Button " + i));
    }

    public static void main(String[] args) {
          SwingDemo4 frame = new SwingDemo4();
          frame.init();
          frame.setSize(200,700);
          frame.setVisible(true);
    }
}
```

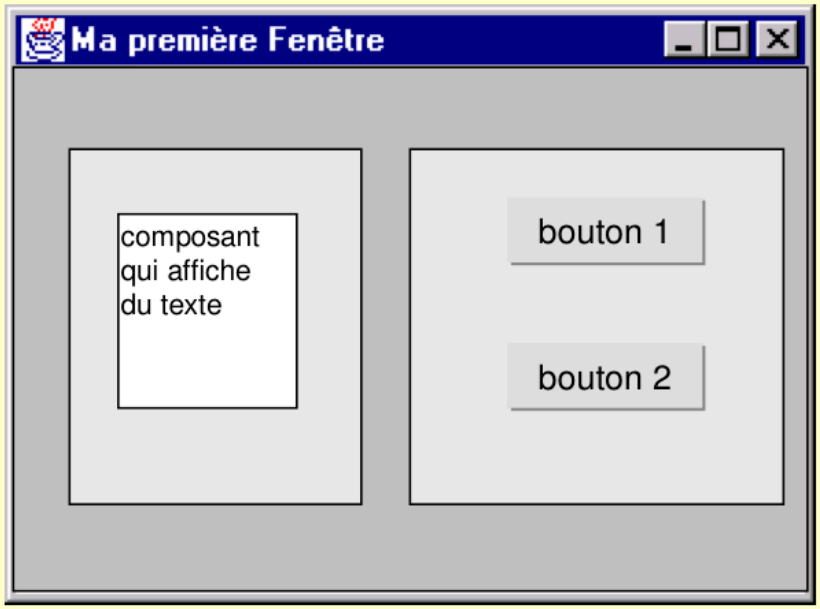


Inspiré de: Bruce Eckel, Thinking in Java, 2e édition

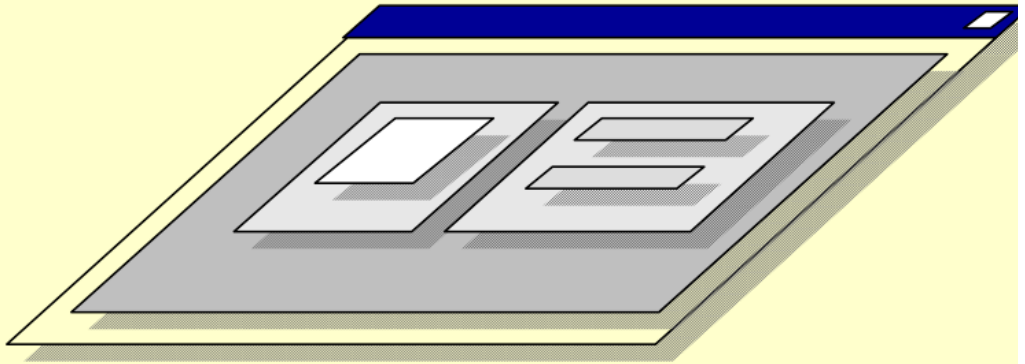
```
import javax.swing.*;
import java.awt.*;

public class SwingDemo5 extends JFrame {
    public void init() {
        Container cp = getContentPane();
        this.setTitle("example 5");
        this.setDefaultCloseOperation(EXIT_ON_CLOS

        cp.setLayout(new GridLayout(7,3));
        for(int i = 0; i < 20; i++)
          cp.add(new JButton("Button " + i));
    }

    public static void main(String[] args) {
      SwingDemo5 frame = new SwingDemo5();
      frame.init();
      frame.setSize(200,700);
      frame.setVisible(true);
   }
}
```

Inspiré de: Bruce Eckel, Thinking in Java, 2e édition

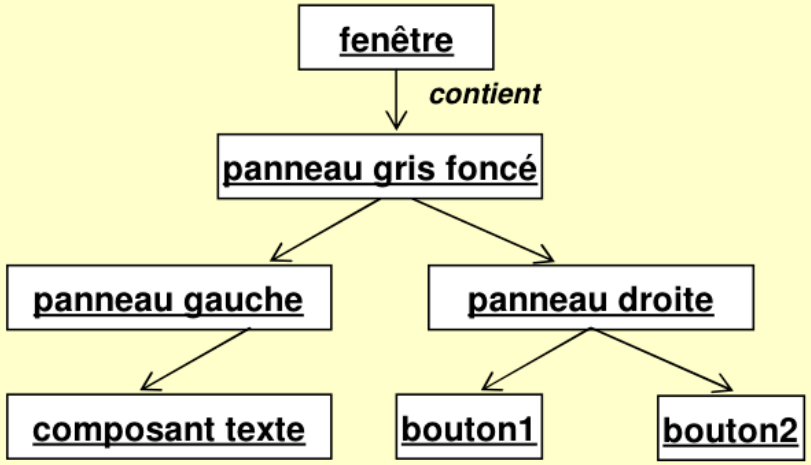
Organisation d'une fenêtre



Repris du cours de P. Genoud

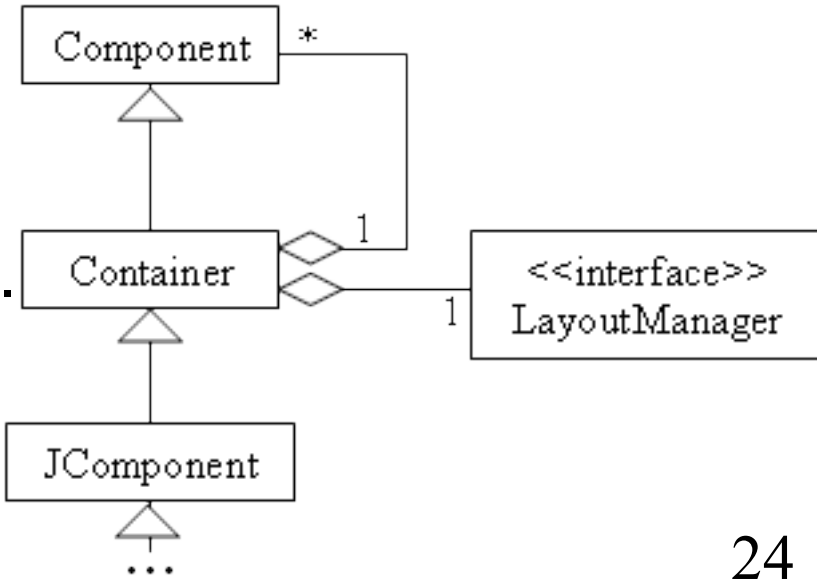


```
panneauGauche.add(composantTexte);

panneauDroite.add(bouton1);
panneauDroite.add(bouton2);

panneauGris.add(panneauGauche);
panneauGris.add(panneauDroite);

fenêtre.add(panneauGris);
```



Gestion de l'espace : *LayoutManager*

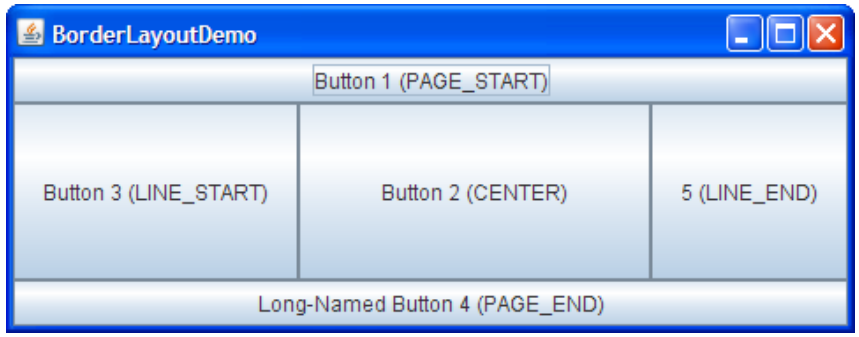
- Chaque conteneur est un "composite" : il peut contenir soit des composants atomiques, soit d'autres conteneurs.
- Le placement des composants dans un conteneur correspond à une stratégie de placement.
- Chaque conteneur (top-level ou autre) délègue à un *LayoutManager* la responsabilité de placer les composants en fonction
  - de leurs tailles préférées,
  - des contraintes du conteneur.
- *LayoutManager* est une interface.



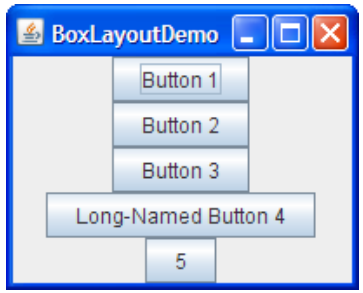
Exemples de *Layout Manager*

BorderLayout : 5 composants :

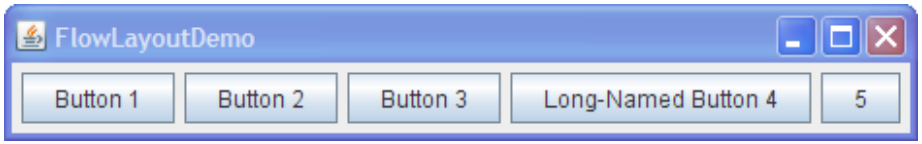
- North, South, East, West et Center.



BoxLayout : en ligne ou en colonne

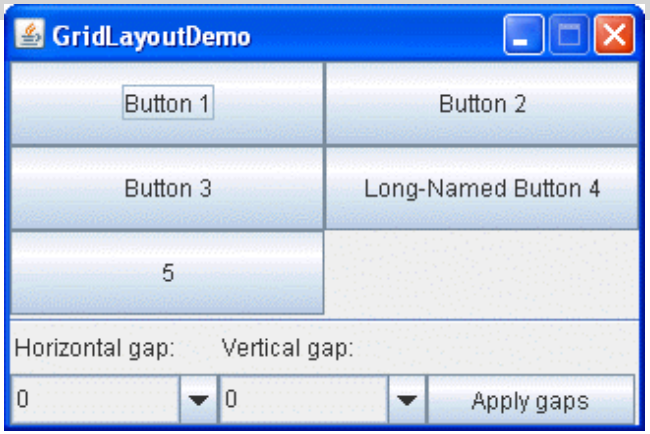


FlowLayout : le défaut, en ligne

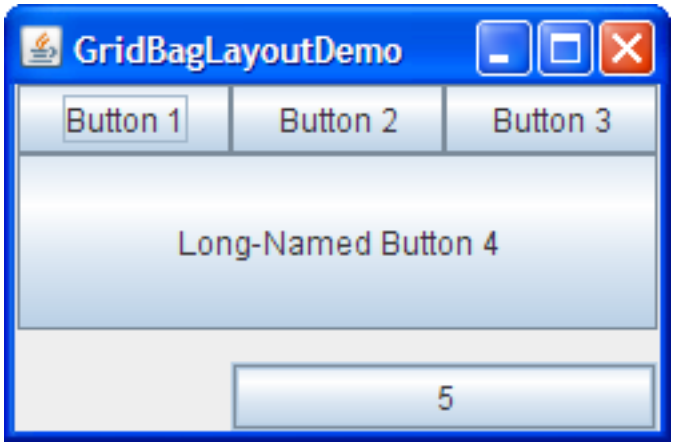


Exemples de *Layout Manager - 2*

GridLayout : en grille



GridBagLayout : en grille mais plus sophistiqué.



Layout Manager liste et exemples

http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

Architecture Swing - Synthèse

Une application = une fenêtre avec des « choses » bien placées.

- Un conteneur (*container*) top-level : « LE » conteneur, le composant racine, par exemple la fenêtre.
- Il contient d'autres composants dans son champ content pane.
- Ces composants peuvent être :
  - Composants atomiques (simples), par ex: un bouton.
  - Des composants intermédiaires (composés) qui permettent de diviser la fenêtre : conteneurs pour plusieurs composants, des panneaux.

Un composant graphique doit, pour apparaître, faire partie d'une hiérarchie de conteneur : c'est un arbre avec

- pour feuille des composants atomiques et
- pour racine un *top-level container*.
- Un composant ne peut être contenu qu'une seule fois.
- Le placement des composants dans un conteneur correspond à une stratégie de placement, délègué à un *LayoutManager*

Seconde partie : événements

Rappel sur les Threads 1/2

- La création d'un thread passe par la création d'un objet de la classe java.lang.Thread.
- Un objet de type Thread représente un thread réel et sert à le manipuler (contrôle, priorité, synchronisation)
- Il faut indiquer au thread quelle méthode exécuter (faire un *run*)

- Créons un objet qui soit « *runnable* » ou hérite de « Thread », il servira de cible au thread.
- Un objet runnable ou Thread est un objet qui implémente l'interface Runnable, avec une méthode *run()*.

- Tout thread débute dans la vie par l'exécution d'une méthode run().

Rappel sur les Threads 2/2

- Une fois créé, un thread ne fait rien tant qu'il n'a pas commencer : start().
- start() lance l' exécution de run().
- Une fois lancer, un thread va jusqu'au bout de run(), du moins tant qu'on ne le stop pas : stop().

```
class Animation implements Runnable {
    ...
    public void run() {
        while ( true ) {
            // Draw Frames
            ...
            repaint();
        }
    }
}
```

Une première manière, pas très objet.

```
Animation happy = new Animation("Mr. Happy")
Thread myThread = new Thread( happy );
// Jusqu'ici rien n'est lancé !
myThread.start();
```

Interaction, programmation événementielle

- Le principal objectif d'une application graphique est la programmation événementielle :
  - l'utilisateur peut déclencher des événements et réagir à ce qui se passe dans la fenêtre.
  - La communication est « asynchrone »
  - Au contraire d'un programme en mode console, dans lequel le programme régit les actions de l'utilisateur à sa guise (synchrone).

=> utilisation des threads!

Interaction, programmation événementielle

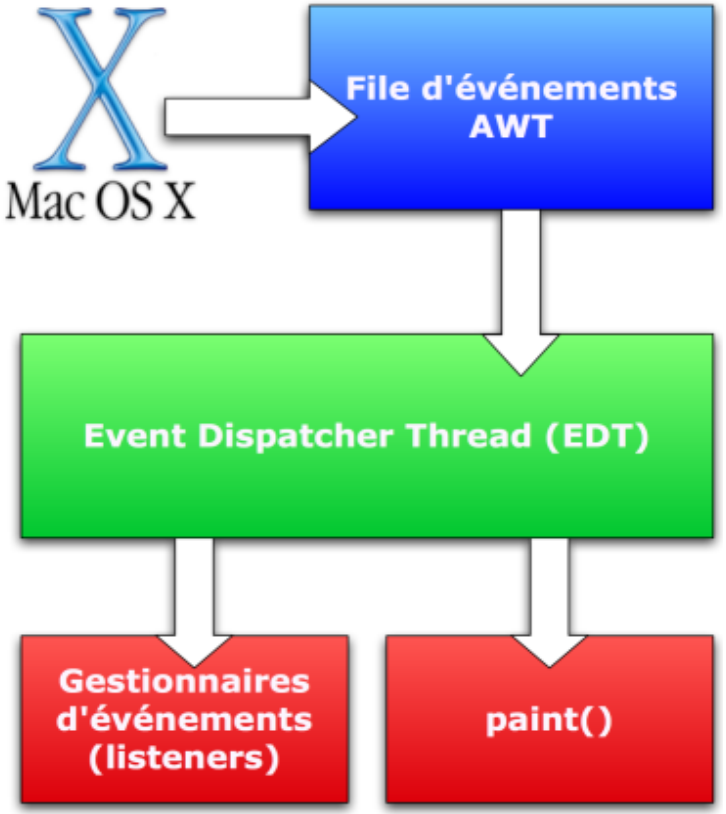
- Exemple d’un bouton :
  - Un bouton est un élément graphique sur lequel l'utilisateur peut cliquer pour déclencher une action.
  - Le bouton ne fait rien tant que l'utilisateur n'a pas cliqué dessus.
  - Lors d’un clique un événement est crée ... reste à le traiter !

=> utilisation des threads!

Le graphique, les événements et les threads

Les 3 threads de la JVM :

- Le premier est le ***"main application thread"*** qui se charge d'exécuter la méthode main() de l'application.
- Le deuxième thread est le ***"toolkit thread"*** dont le rôle est de recevoir les événements du système d'exploitation, par exemple un clic de souris, et de les placer dans une file d'attente d'événements, pour être traités par
- Le troisième thread, appelé *"event dispatching thread"* ou ***EDT*** :
  - il répartit les événements reçus vers les composants concernés et
  - invoque les méthodes d'affichage.



Le clique sur un bouton

- Lors d'un clique sur un bouton :
  - un événement est crée
  - cet événement est placé dans une file d'attente des événements (AWT Event Queue)
  - dans l'attente d'être traité par l'*EDT*.

- Attention :
  - il y a un seul thread (EDT) pour traiter les événements et redessiner.
  - L'EDT traite un événement après l'autre.
  - Il faut attendre la fin du traitement pour passer à autre chose.

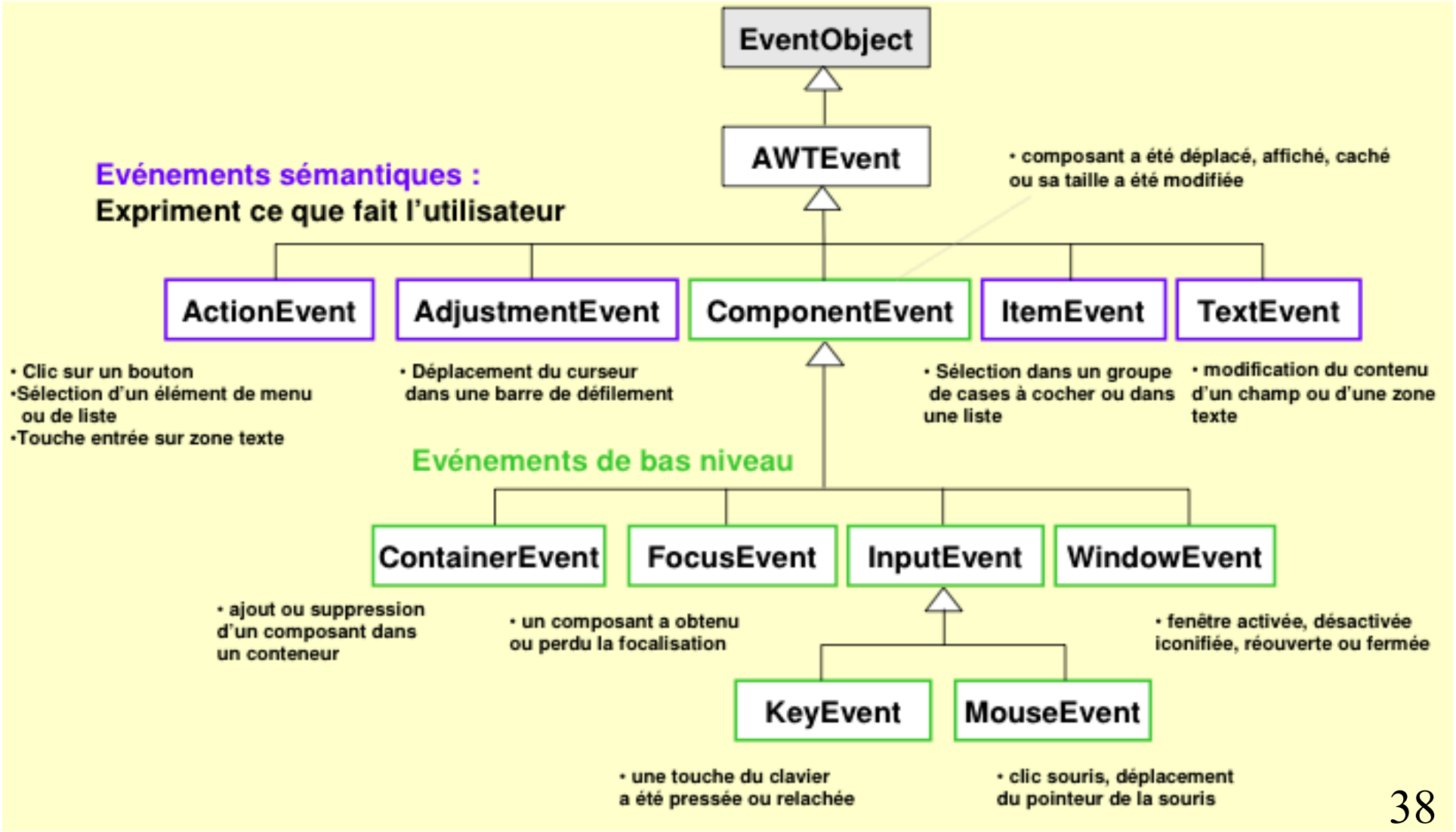
Gestion des événements

- Un composant qui crée des événements est appelé **source**.
- Le composant source délègue le traitement de l'événement au composant **auditeur**. Un composant qui traite un événement est appelé auditeur (**listener**).
- Un composant auditeur doit s'inscrire auprès du composant source des événements qu'il veut traiter.

- Un événement peut provenir :
  - du clavier
  - un clique souris
  - un passage de la souris
  - ....
- A chaque type d'événement, une classe (existante) !
- A chaque type d'événement, son type d'écouteur (à faire)!

```
public class SwingDemoEvent0 extends JFrame implements
                ActionListener{

        public void actionPerformed(ActionEvent event) {
                System.exit(0);
        }

        public final void init() {
                JButton quitButton = new JButton("Quit");
                quitButton.addActionListener(this);
                getContentPane().add(quitButton);
        }

        public static void main(String[] args) {
                SwingDemoEvent0 frame = new SwingDemoEvent0();
                frame.init();
                frame.setTitle("Quit button");
                frame.setSize(100, 100);
                frame.setVisible(true);
        }
}
```

Hiérarchie des événements



ActionEvent, de qui, pour qui ?

Les sources :

- Boutons : JButton, JRadioButton, JCheckBox, JToggleButton
- Menus : JMenuItem, JMenu, JRadioButtonMenuItem, JCheckBoxMenuItem
- Texte : JTextField
- ...

Les auditeurs :

- Il faut implémenter l'interface qui correspond au type de l'événement
- e.x. ActionEvent => ActionListener :

```
public interface ActionListener extends EventListener {
    /** Invoked when an action occurs.*/
    public void actionPerformed(ActionEvent e)
}
```

Evénements / Auditeur

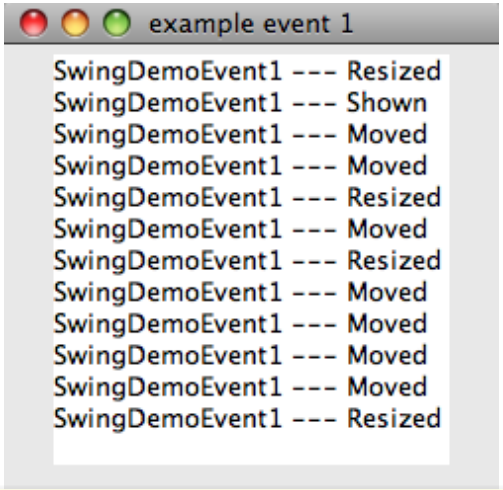
- Tout événement hérite de la classe EventObject
- Tout auditeur correspond à une interface qui hérite de EventListener.

- toute classe désirant recevoir des notifications d'un type d'événement donné devra implémenter l'interface correspondante :
  - ActionEvent — ActionListener
  - MouseEvent — MouseListener
  - KeyEvent — KeyListener
  - ...

Auditeur : prendre son abonnement

- Un auditeur doit s'abonner auprès du composant
- Un auditeur peut avoir plusieurs abonnements
  - e.x. un auditeur traite les événements de plusieurs boutons
- Un composant peut avoir plusieurs auditeurs.
  - e.x. un pour les événements "click" et mouvement sur le bouton.

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

public class SwingDemoEvent1 extends JFrame implements ComponentListener {
    JTextArea display;

    public void init() {
        this.setTitle("example event 1");
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        display = new JTextArea();
       display.setEditable(false);
       cp.add(display);
       this.addComponentListener(this);
    }

   protected void displayMessage(String message) {
       display.append(message + "\n");
       display.setCaretPosition(display.getDocument().getLength());
    }

   // ComponentListener methods
   public void componentHidden(ComponentEvent e) {
       displayMessage(e.getComponent().getClass().getName()
       + " --- Hidden");
   }
   public void componentMoved(ComponentEvent e) { ...
   public void componentResized(ComponentEvent e) { ...
   public void componentShown(ComponentEvent e) { ...
```



déplacer le contenu de init() dans le constructeur

```java
public static void main(String[] args)
{
        SwingDemoEvent1 frame = new
        SwingDemoEvent1();
        frame.init();
        frame.setSize(250,700);
        frame.setVisible(true);
}
}
```

Interaction entre composant

- Souvent un événement créé par un composant modifie un autre composant (e.x. JColorChooser modifie le couleur d'un JPanel).

- Comment faire communiquer l'auditeur avec tout le monde ?
  - en respectant le principe d'encapsulation !
    - et sans variables globales
  - sans un code trop complexe !

- La solution les classes internes ("inner class")

Parenthèse : les classes internes et anonymes

Inner Class : Classe interne

- Définition d'une classe à l'intérieur d'une autre classe.
  - Regroupement logique et cohérent de classes.
  - Maîtrise de leurs visibilités et de leurs accessibilités.

```
// Definition
public class Paquet {

     class Tarif {
       private int i = 10;
       public int value() {return i; }
     }

     class Destination {
       private String label;
       Destination(String whereTo) {
         label = whereTo;
       }
       String readLabel()
       { return label; }
     }
```

```
// Usage
  public Destination envoi(String s){
       Tarif c = new Tarif();
       Destination d=new Destination(s);
       System.out.println(d.readLabel());
       return d;

     }

     public static void main(String[] args){
       Paquet p = new Paquet();
       p.envoi("Tanzanie");
     }
}// Adapté du livre de Bruce Eckel
```

Inner class : Définition, usage, compilation.

- Une classe interne se définit exactement comme une classe « normale », mais dans le block d'une autre classe.
- Elle s'utilise comme une classe « normale ».
- Si elle n'est pas déclarée *private* ou si elle n'est pas anonyme elle est accessible hors de la classe englobante :

```
ClassEnglobante.ClasseInterne foo=new ClassEnglobante.ClasseInterne();
Paquet.Destination p = new Paquet.Destination(''Joyeuse'');
```

- Compilation :

```
➢ javac Paquet.java
➢ ls -1
   Paquet$Destination.class
   Paquet$Tarif.class
   Paquet.class
   Paquet.java
```

Inner class : Pourquoi ?

- Remarques préliminaires :
  - une classe interne peut être private, protected, ou public;
  - une classe peut être interne à une classe, à une méthode.

- Pour cacher/masquer des mécanismes, des implémentations

- Une classe interne a un accès total aux éléments de la classe englobante.
- L'utilité vient avec l'association des classes internes et *upcasting.*
- Surtout avec l'implémentation d'interface.

Inner class et upcasting

- Supposons que nous disposions des interfaces :

```
public interface Tarif { int value(); }
public interface Destination { String readLabel(); }
```

```
// Definition
public class Paquet {

    private class PTarif implements Tarif{
        ...
     }

    private class PDestination implements Destination{
       ...
    }

    public Destination envoi(String s){
       Tarif c = new PTarif();
       Destination d=new PDestination(s);
       System.out.println(d.readLabel());
       return d;
    }
```

L'implémentation est totalement cachée. Seule est visible une référence sur l'implémentation de l'interface

Inner class : classe anonyme

```
public class Paquet2 {

        String dest;
        public Destination envoi(String s) {

                dest =s;

                Tarif c = new Tarif(){
                        private int i = 10;
                        public int value() { return i;}
                   };

                Destination d = new Destination(){
                        private String label = dest;
                        public String readLabel() { return label; }
                };

                System.out.println("envoie en "+ d.readLabel());
                return d;
        }
}
```

Tarif et Destination sont des interfaces

Un instance necessaire

Fin de la parenthèse, retour aux interfaces graphiques

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

public class SwingDemoEvent2 extends JFrame  implements ComponentListener {
    JLabel label;
    JButton button;
    int clicCount = 0;

    class MyButtonActionListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
          clicCount++;
          label.setText(Integer.toString(clicCount));
        }
      }

    public void init() {
       this.setTitle("example event 2");
       this.setDefaultCloseOperation(EXIT_ON_CLOSE);

       Container cp = getContentPane();
       cp.setLayout(new GridLayout(1,2));

       button = new JButton("clique?");
       label = new JLabel();
       label.setText("...");

       button.addActionListener(new MyButtonActionListener());

       cp.add(button);
       cp.add(label);

       this.addComponentListener(this);
    }
```

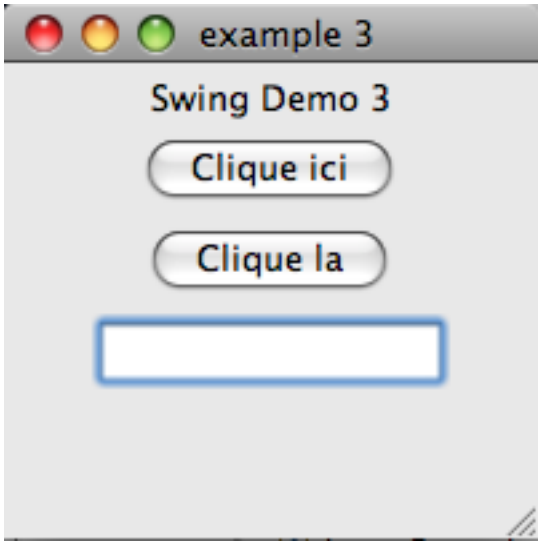
inner class


```
    public void componentHidden(ComponentEvent e) {}
    public void componentMoved(ComponentEvent e) {}
    public void componentResized(ComponentEvent e) {}
    public void componentShown(ComponentEvent e) {}

    public static void main(String[] args)
    {
        SwingDemoEvent2 frame = new SwingDemoEvent2()

        frame.init();

        frame.setSize(300,300);
        frame.setVisible(true);
    }
}
```


Inspiré de: Bruce Eckel, Thinking in Java, 2e édition

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

public class SwingDemo3 extends JFrame {

    JButton b1 = new JButton("Clique ici");
    JButton b2 = new JButton("Clique la");
    JTextField txt = new JTextField(10);

    class ButtonListener implements ActionListener // INNER CLASS DEF.
    {
           public void actionPerformed(ActionEvent e) {
                      String name = ((JButton)e.getSource()).getText();
                      txt.setText(name);
           }
    } // END OF INNER CLASS DEFINITION

    ButtonListener bl = new ButtonListener();

    public void init() {

           b1.addActionListener(bl);
           b2.addActionListener(bl);

           Container cp = this.getContentPane();
           this.setTitle("example 3");

           cp.add(new JLabel("Swing Demo 3"));
           cp.setLayout(new FlowLayout());

           cp.add(b1);
           cp.add(b2);
           cp.add(txt);
    }

    public static void main(String[] args)
    {
                SwingDemo3 frame = new SwingDemo3();

                frame.init();

                frame.setSize(200,200);
                frame.setVisible(true);
    }

} // end of SwingDemo3 class definition
```

inner class



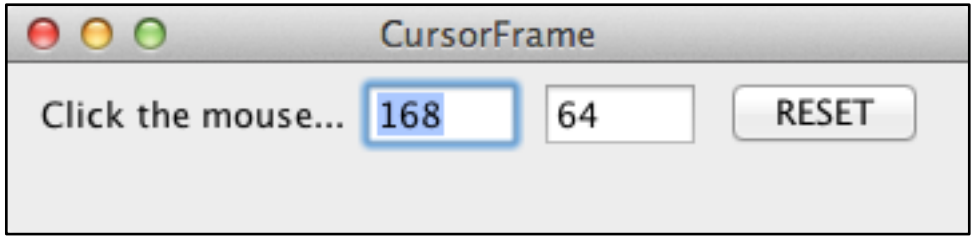
plus concentrée

- Au lieu de “inner class”, utiliser “Anonymous Inner classes"
  - “new <nom-de-classe> ([argument-list]) { <corps> }”

- Cette construction fait deux choses :
  - elle crée une nouvelle classe, sans nom, qui est une sous-classe de <nom-de-classe> définie par <corps>
  - elle crée une instance (unique) de cette nouvelle classe et retourne sa valeur

- Intéressant si nous avons besoin une seule instance (objet) de la classe.
- Cette classe a accès aux variables et méthodes de la classe dans la quelle elle est définie.

```
import ...

public class SwingDemoEvent3 extends JFrame {
    JTextField a, b;
    JButton btn;

    public SwingDemoEvent3() {
        super("CursorFrame");
        setSize(400, 200);
        setLayout(new FlowLayout());
        add(new JLabel("Click the mouse..."));
        a = new JTextField("0", 4);
        b = new JTextField("0", 4);
        btn = new JButton("RESET");
        add(a); add(b); add(btn);

        addMouseListener(new MouseAdapter() {
             public void mousePressed(MouseEvent e) {
                        a.setText(String.valueOf(e.getX()));
                        b.setText(String.valueOf(e.getY()));
             }
        });
        addWindowListener(new WindowAdapter() {
             public void windowClosing(WindowEvent e) {
                        setVisible(false);
                        dispose();
                        System.exit(0);
             }
        });
        btn.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e) {
                        a.setText("0");
                        b.setText("0");
             }
        });
    }
```



anonymous inner classes

```
    public static void main(String[] args) {
        SwingDemoEvent3 app = new SwingDemoEvent3();
        app.setVisible(true);
    }
}
```

Exercice

- Faire une calculatrice
  - des boutons pour: les chiffres et les opérateurs (+, -, /, *, =)
  - zone texte pour le résultat

Pour aller plus loin...

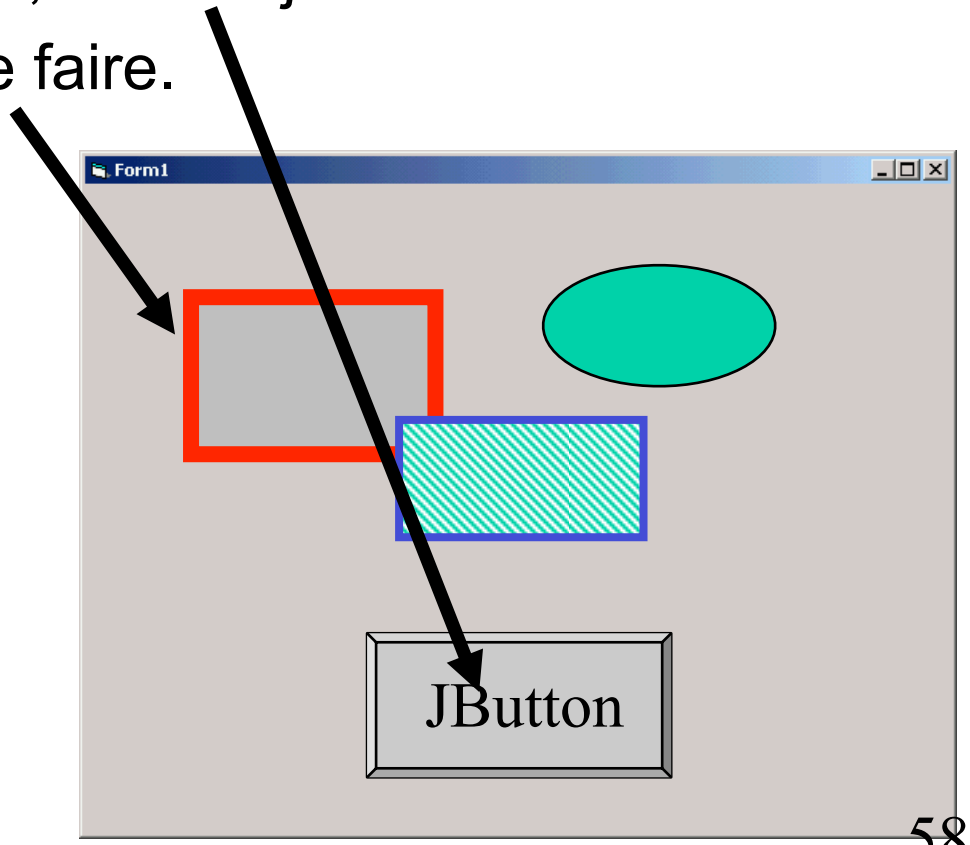
http://download.oracle.com/javase/tutorial/uiswing/examples/components/index.html

Bruce Eckel, Thinking in Java (2nde édition), Chapitre 13 (creating windows and applets)

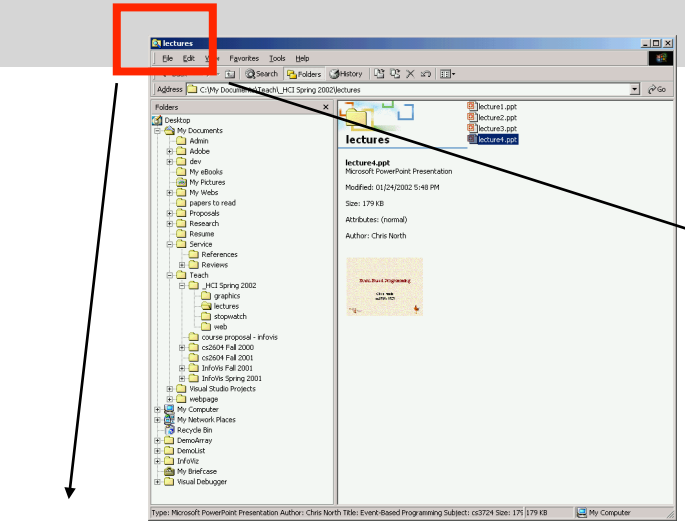
Troisième partie :
Dessiner son propre composant graphique

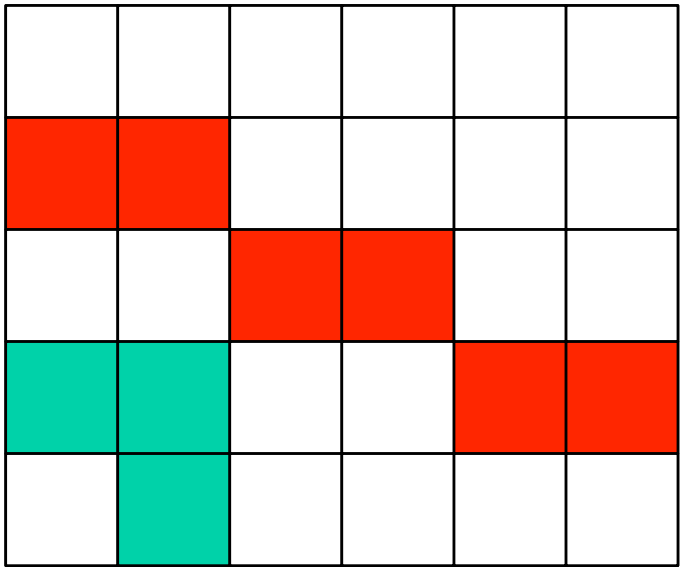
Dessin des composants

- Une fenêtre (ou un panneau) est un canevas (canvas) sur lequel l'application dessine ou peint :
  - Les composants de l'API, c'est déjà fait.
  - Le reste, c'est à vous de faire.



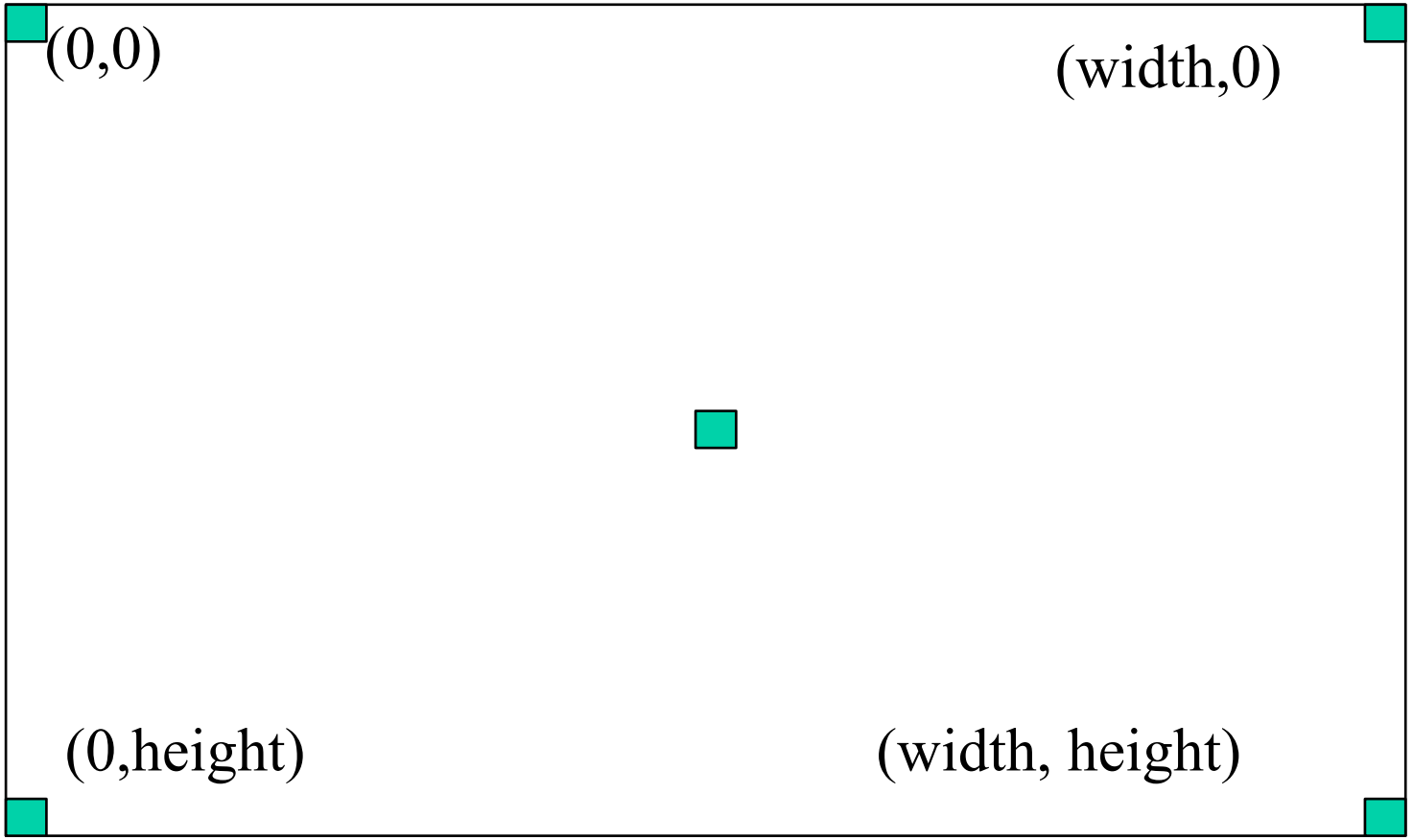
Pixel = picture element





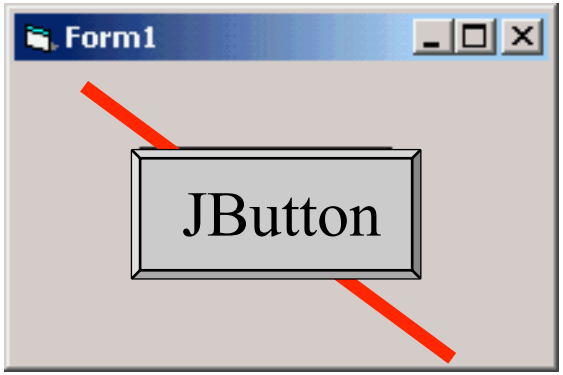
Le système de coordonnées

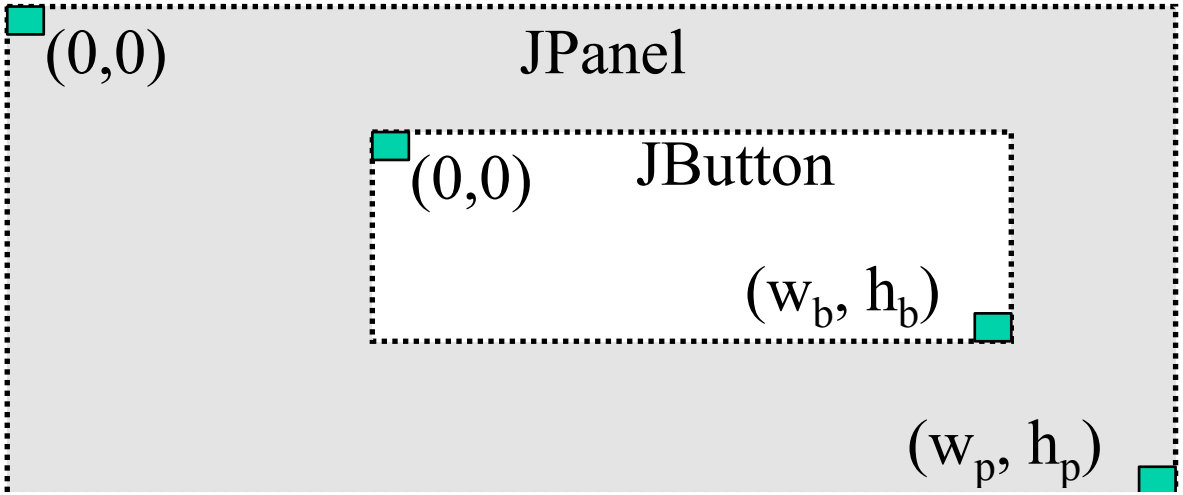
- Presque du cartésien :
  - (0,0) haut gauche



Fenêtre et sous-fenêtre

- Chaque composant possède son espace de dessin :
  - sa sous-fenêtre, *subwindow*
  - Subwindow = Aire rectangulaire dans le composant parent avec son propre système de coordonnées
- Clipping, les règles : un composant ne peut peindre
  - hors de sa sous-fenêtre
  - sur un de ses composants.





Mon propre composant graphique

- Hérite de Component, JComponent ou **JPanel**
- Redéfinir la fonction paint

```
void paint (Graphics G_Arg) {
    Graphics2D g2 = (Graphics2D) G_Arg;
}
```

- Hérite de repaint() pour lancer paint(…)
  - Appeler repaint() si nous voulons mettre à jour le composant
  - Asynchrone, gestion automatique du *Graphics*
- Hérite de méthodes externes dont il faut tenir compte
  - setSize(), setEnable(), setBackground(), setFont(), setForeground(), etc.

void paint (Graphics G_Arg) {

- Une instance de *Graphics* est donnée par java pour ce composant afin de dessiner

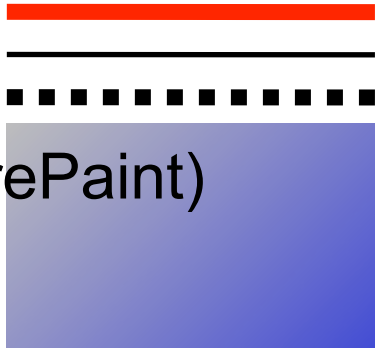
- Un Graphics possède un état :
  - Translation à l'origine pour le rendu : translate()
    - 0,0 = coin haut gauche par défaut
  - Zone d'effet (!= rectangulaire) = Clip
    - Par défaut : tout, mais on peut se restreindre
  - Couleur de dessin
    - Color col1 = new Color (255, 0, 0); RGB mais aussi HSB
  - Police de caractère
    - Font font1 = new Font("SansSerif", Font.BOLD, 12);

Fonctions de dessin avec Graphics

- Exemple : public void drawLine (x1, y1, x2, y2)
  - Dépend de la couleur courante
- **fill*() / draw*()** = remplissage ou contour
  - * = { Rect, Oval, String, Arc, Polygon, PolyLine }
- Fonction **clear()** pour nettoyer
- Une fonction FontMetrics getFontMetrics()
  - Renvoi une instance qui mesure le texte
- Fonction **drawImage()** pour le dessin d'image
  - Nécessite une instance de "Image"
  - Asynchrone. Possibilité d'écoute : ImageObserver

Dessin avec Graphics2D

- Fonction *public void paint(Graphics g)* appelé par Java
  - Mais *Graphics* = ***Graphics2D*** depuis v1.1
  - Transtypage : `Graphics2D g2 = (Graphics2D) g;`



- Etat de dessin plus élaboré (attributs)
  - Paint : peinture (Color, GradientPaint ou TexturePaint)
  - Font : police de caractère
  - Clip : zone de restriction du dessin
  - Stroke : pinceau = forme, épaisseur (1p), joins aux angles
  - Transform : Matrice affine de transformation
    - Translation, rotation, zoom, penchant (sheer)
  - Composite : règle de superposition d'un pixel de couleur avec un autre
  - Liste de RenderingHint définissant la qualité de rendu

Peinture, mode d'emploi

```
import java.awt.Graphics
import java.awt.Graphics2D // Java2
```

1. récupérer le "graphics context" du composant

```
Graphics g = myJPanel.getGraphics( );
Graphics2D g2 = (Graphics2D) g;
```

2. Peindre

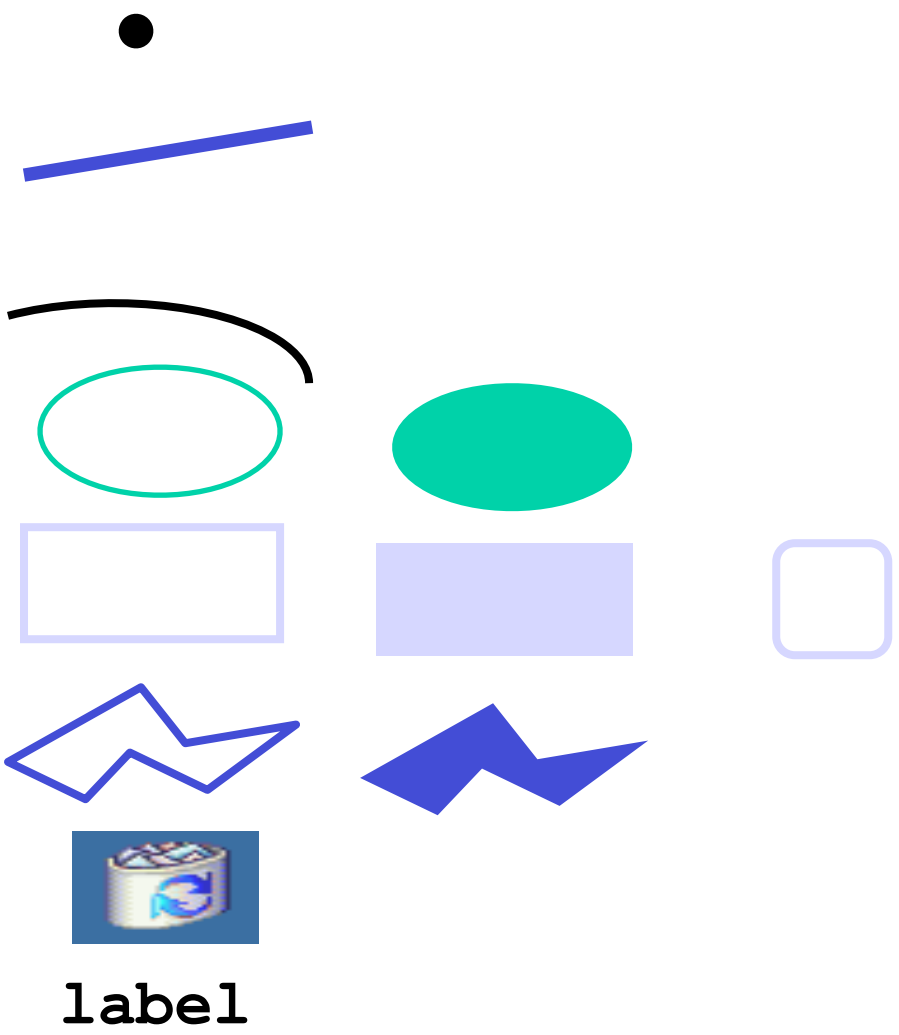
```
g2.drawLine(x1,y1, x2,y2);
```

Nouveau composant, un exemple

```
public class MyPanel extends JPanel {

  // like paint(Graphics g) but only interior
   public void paintComponent(Graphics g){
     super.paintComponent(g);          // erases background
     Graphics2D g2 = (Graphics2D)g;    //cast for java2

     // my graphics:
     g2.setColor(new Color(255,0,0));
     g2.fillRect(10,10,200,50); // left, top, width, height
     g2.setColor(new Color(0,0,0));
     g2.drawString("Hello World", 20, 20);// S, left, BOTTOM
  }
}
```

Peinture, exemples de « draw » et « fill »

- Point (x,y)
- Line (pt1,pt2)
- PolyLine (pt list)
- Arc
- Oval (pt, w,h)
- Rectangle (pt, w,h)
    - RoundRectangle
- Polygon (point list)
- Image (file, x,y)
- Text (string, x,y)

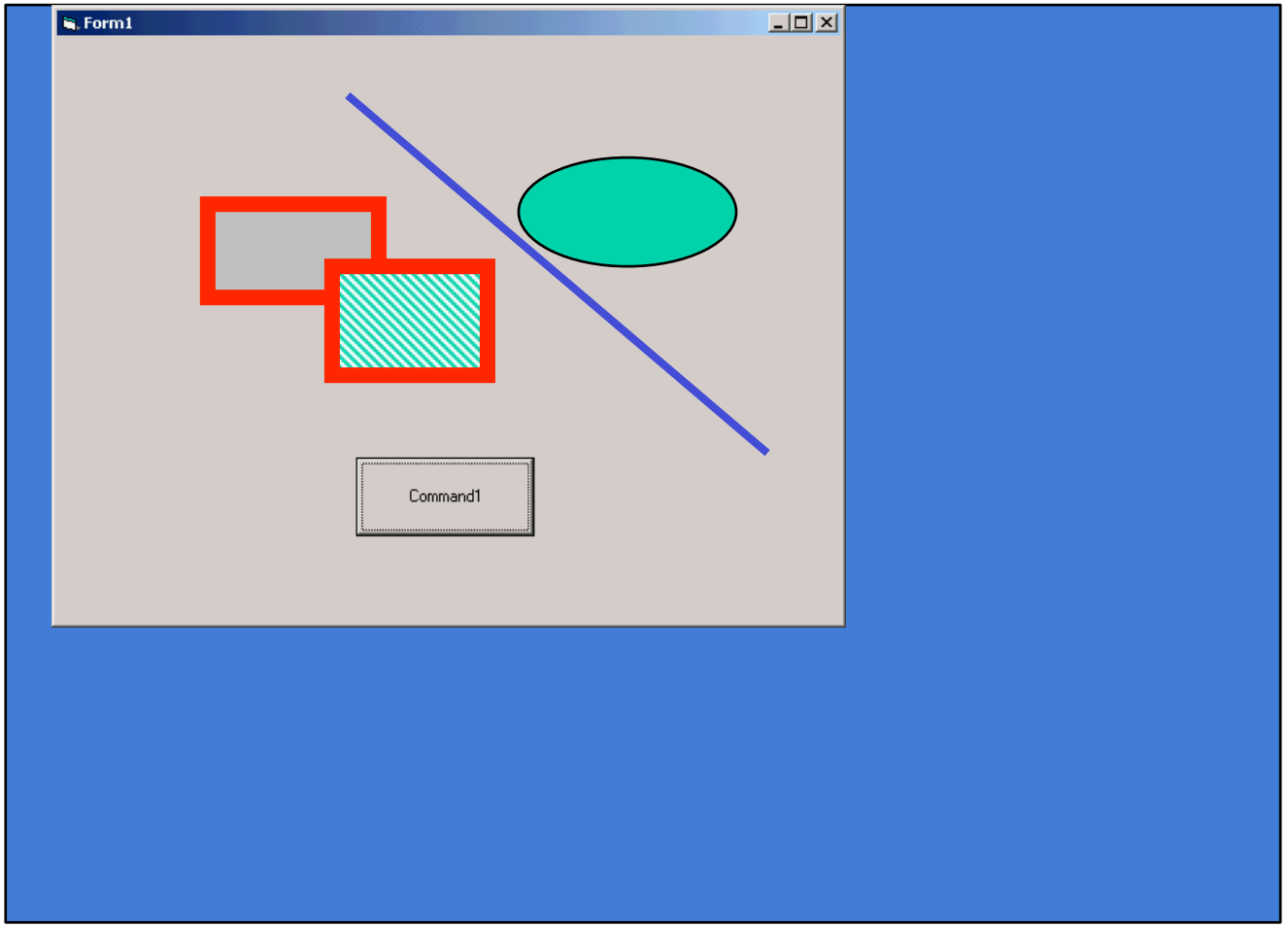


Quand repeindre ?

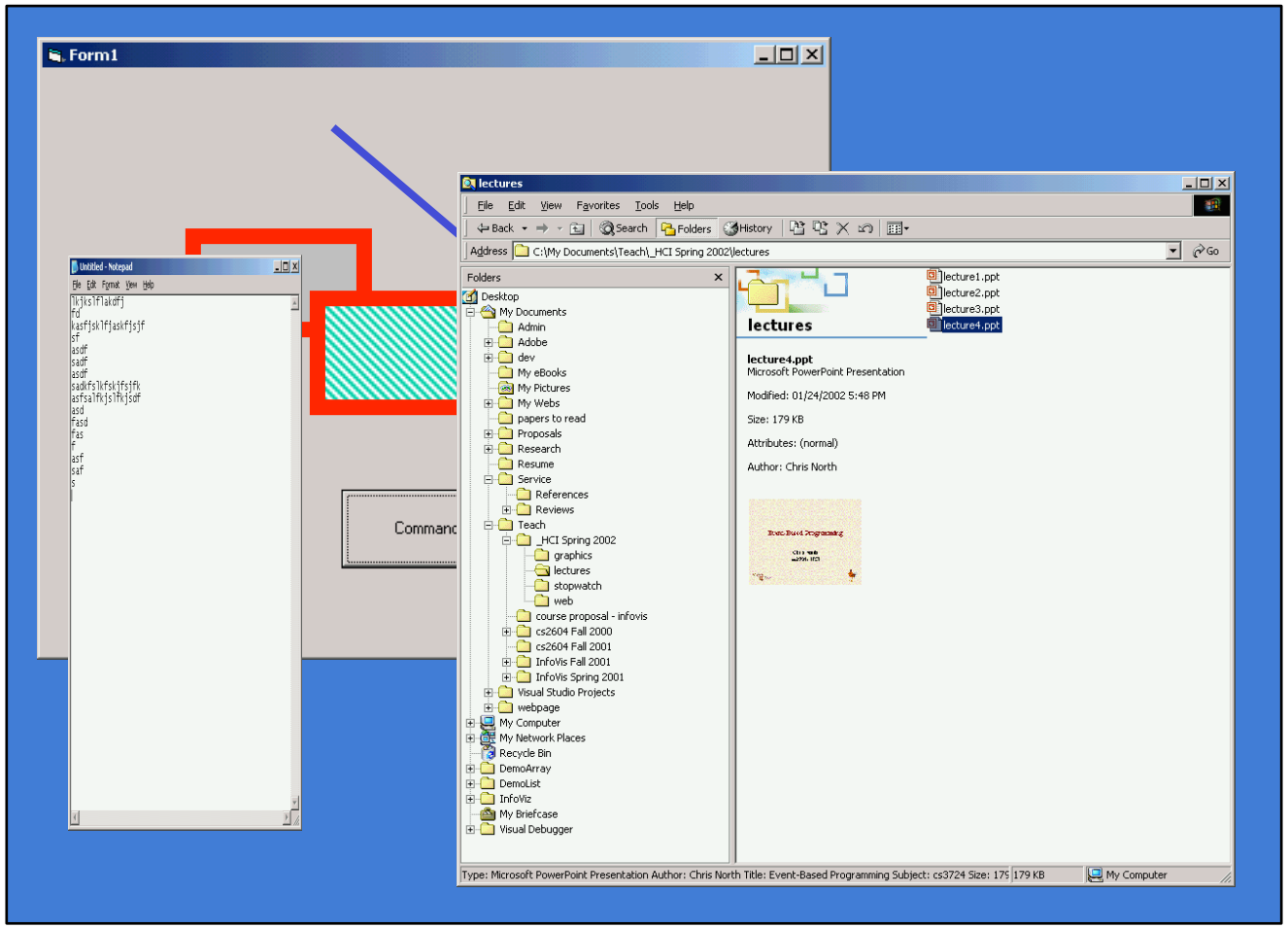
- L'écran est une feuille de dessin unique
  - Toutes les fenêtres sont peintes sur la même feuille.
  - Les fenêtres ne se souviennent pas de ce qu'elle cache.
  - Besoin de repeindre, dès qu'une nouvelle zone de l'écran apparait.

- événements de repaint
    - ouverture, changement de dimension, mise au premier (ou arrière) plan.
    - quand d'autre fenêtre viennent modifier l'écran

Un écran avec une application

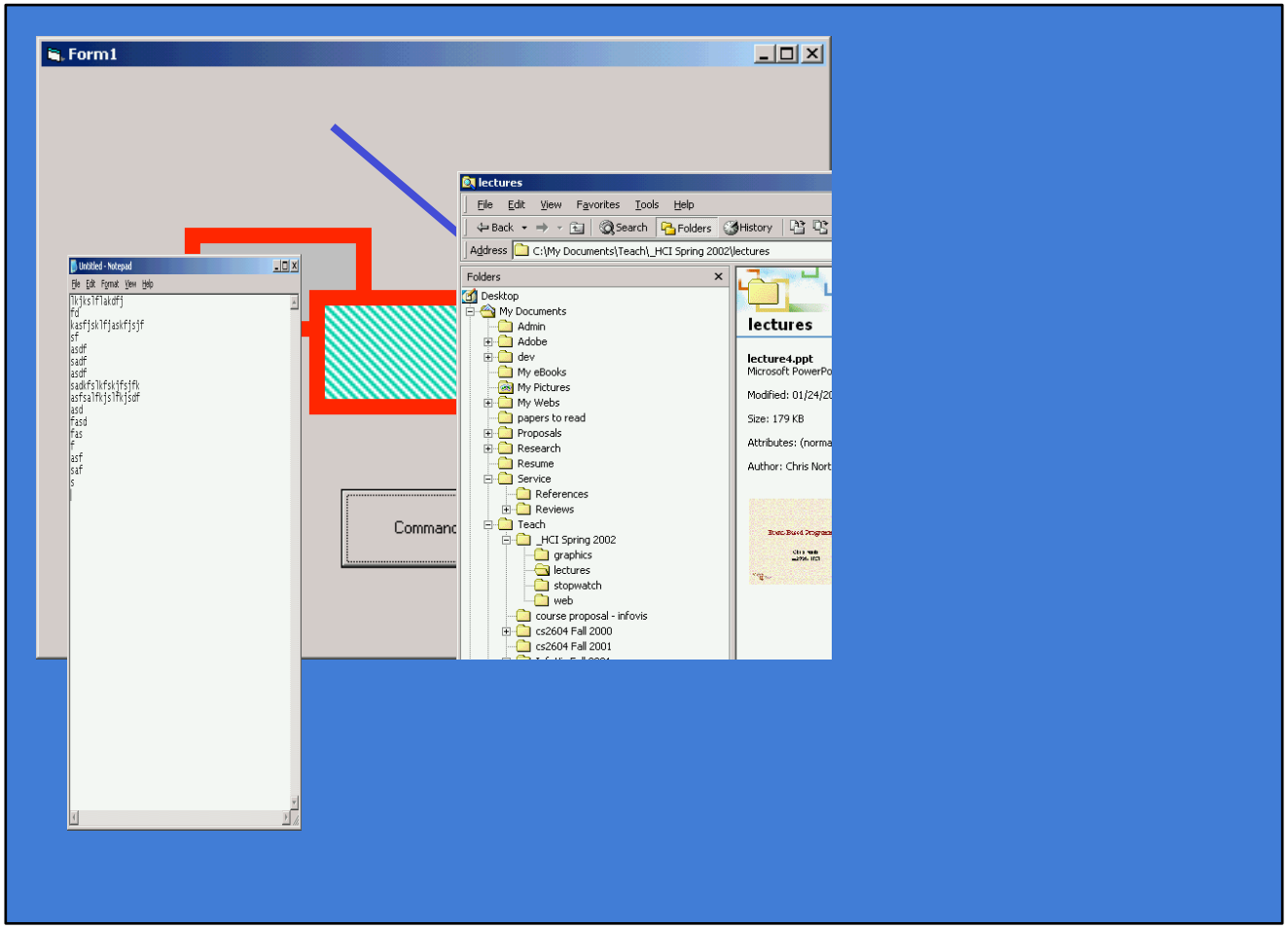


Un écran avec 3 applications



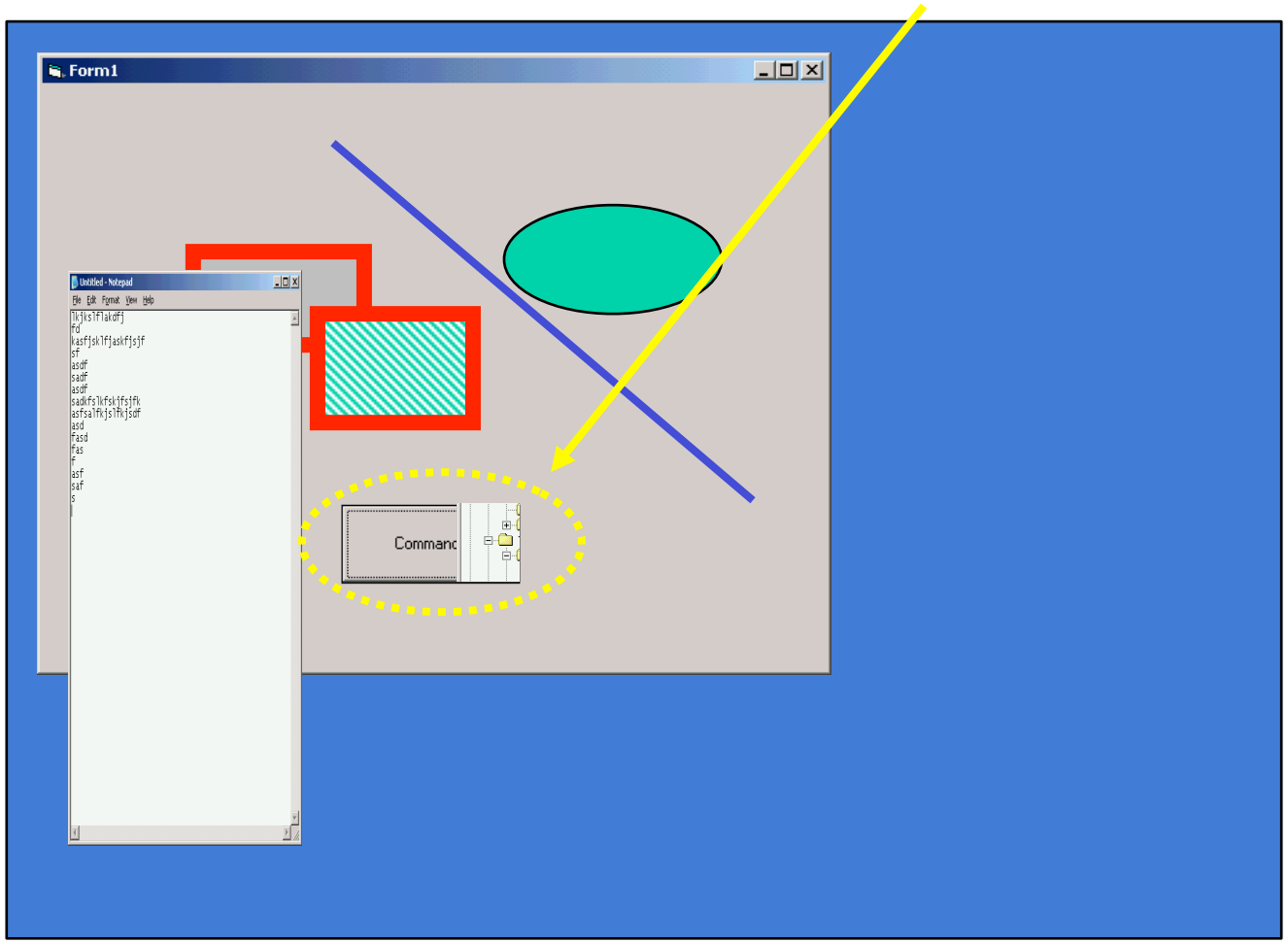
Un écran, fermons une application

Envoie d'événements aux fenêtre restantes : repaint



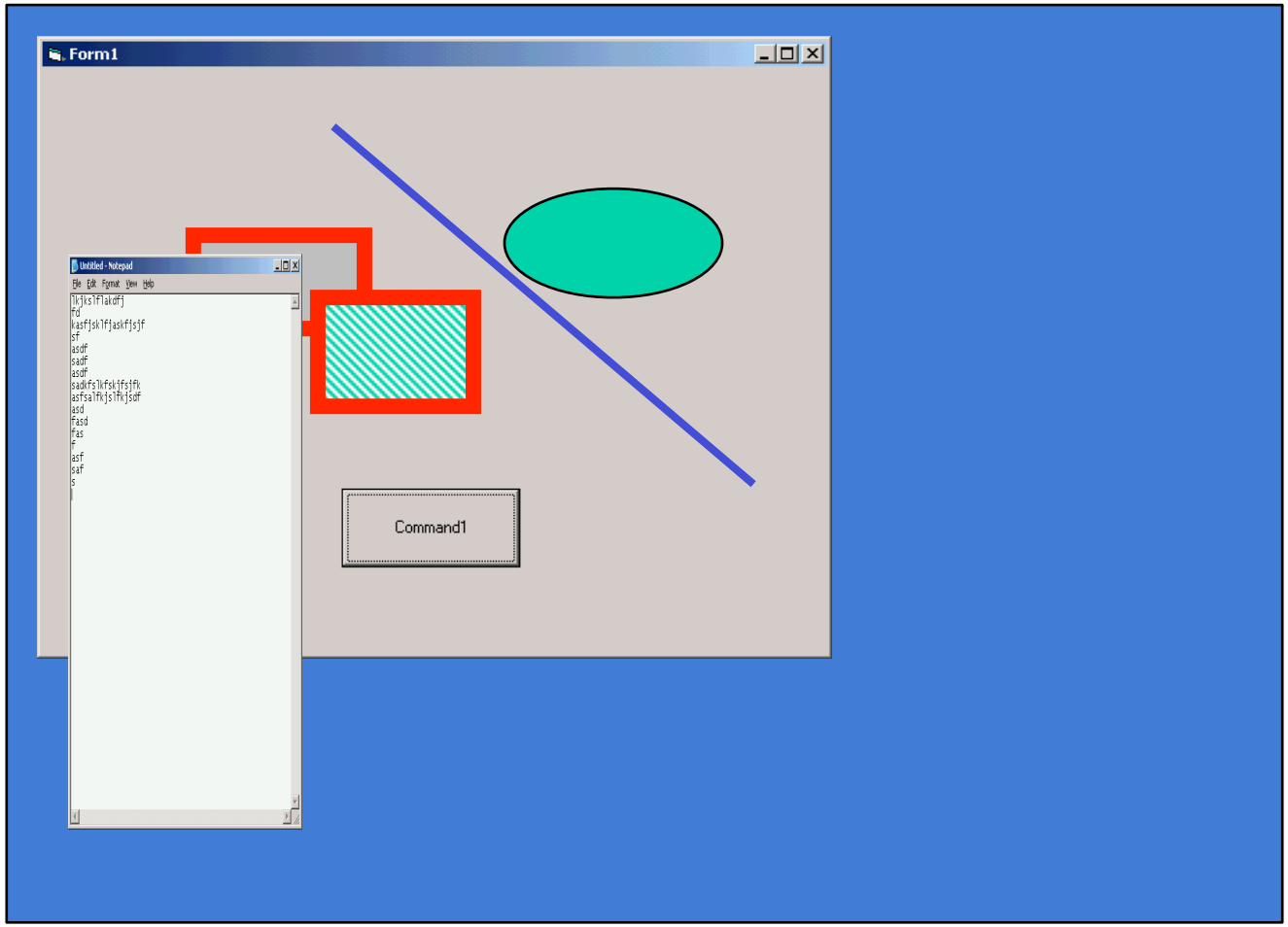
Un écran, fermons une application - 2

dès fenêtre restantes vers ses composants : repaint



Un écran, fermons une application - 3

dès fenêtre restantes vers ses composants : repaint



Peinture en Java : *repaint*

- Repaint event:
  - Les composants Java Swing attrapent les événements *repain*t
  - appel des méthodes ***paintComponent****( )*
  - héritage d'un composant et redéfinition de *paintComponent()*
- Appel explicite : *repaint( )* --> paintComponent( )
- paint() et paintComponent().
  - paint() vient de AWT, et en Swing paint() lance:
    - paintComponent, paintBorder, and paintChildren
  - en générale avec Swing nous pouvons faire paintComponent

Nouveau composant, un exemple

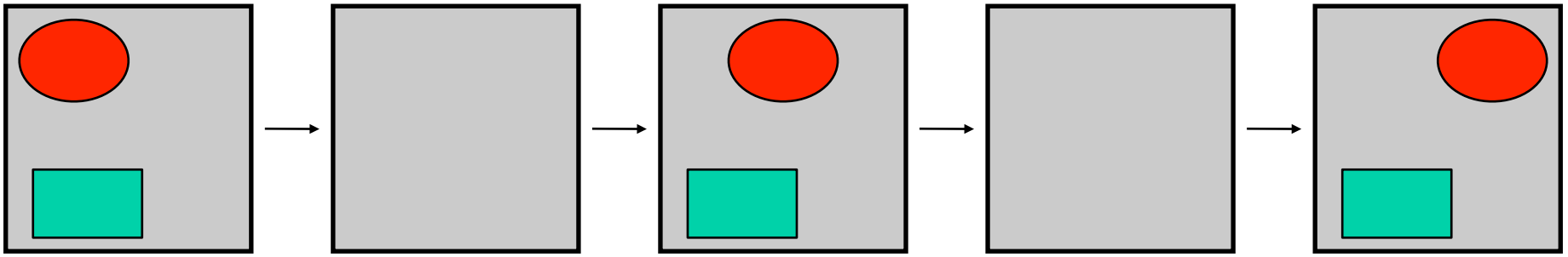
```
public class MyPanel extends JPanel {

  // like paint(Graphics g) but only interior
   public void paintComponent(Graphics g){
     super.paintComponent(g);          // erases background
     Graphics2D g2 = (Graphics2D)g;    //cast for java2

     // my graphics:
     g2.setColor(new Color(255,0,0));
     g2.fillRect(10,10,200,50);
     g2.setColor(new Color(0,0,0));
     g2.drawString("Hello World", 20, 20);
  }
}
```

Nouveau composant, un exemple

```
public class MyPanel extends JPanel {

    public void paintComponent(Graphics g) {
        super.paintComponent(g); // erases background
        Graphics2D g2 = (Graphics2D) g; // cast for java2
        // my graphics:
        g2.setColor(new Color(255, 0, 0));
        g2.fillRect(10, 10, 200, 50);
        g2.setColor(new Color(0, 0, 0));
        g2.drawString("Hello World", 20, 20);
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("my panel");

        JPanel jp = new MyPanel();
        frame.getContentPane().add(jp);

        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
        frame.setSize(250, 200);
        frame.setVisible(true);
    }
}
```

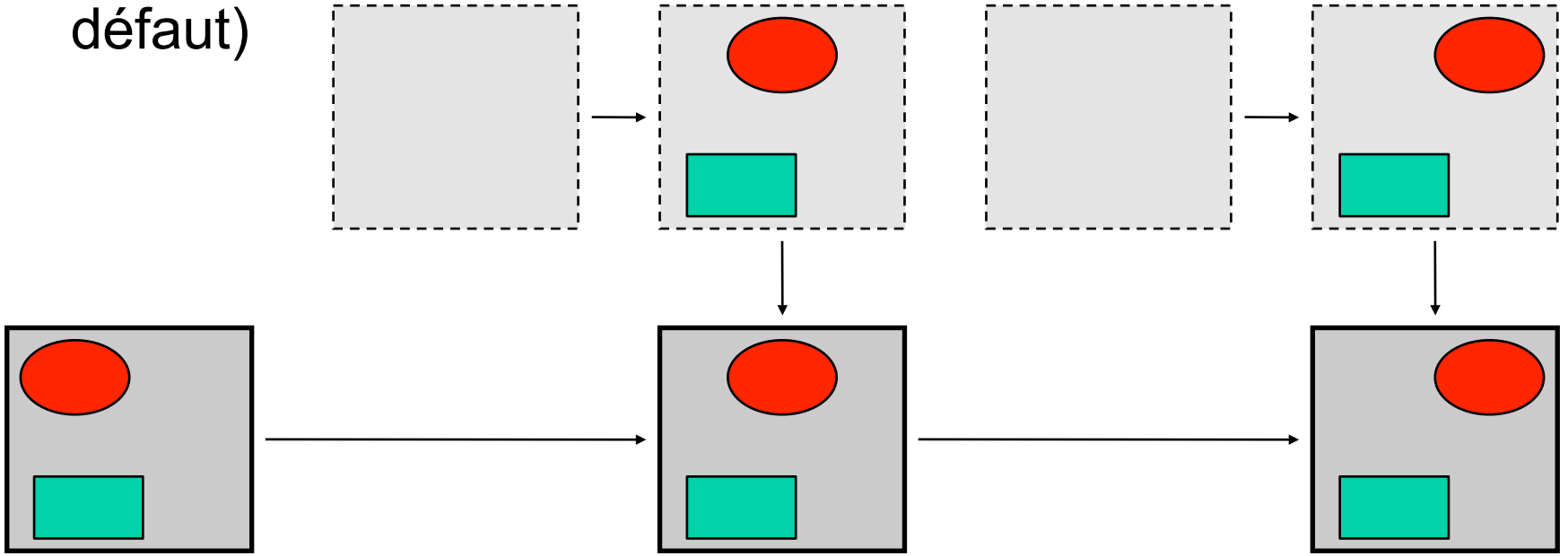
Flashing, un problème

- “Ugly flashing” pour le *repaint*:
  - Paint background
  - Redraw shapes
- Cette approche génère un effet visuel désagréable !



DoubleBufferring

- Dessiner tous un composant sur une image hors-écran :
  - Paint background color
  - Paint shapes
- Puis, dessiner l'image résultante dans le *JPanel*
- Swing le fait pour vous !! (avec setDoubleBuffered(b) activé par défaut)



drawString et Antialiasing

- drawString() dessine chaque glyphe dans une chaîne de texte avec une couleur pour chaque pixel qui est «on» dans ce glyphe.
- Anticrénelage du texte (antialiasing) est une technique pour lisser les contours du texte sur un écran.

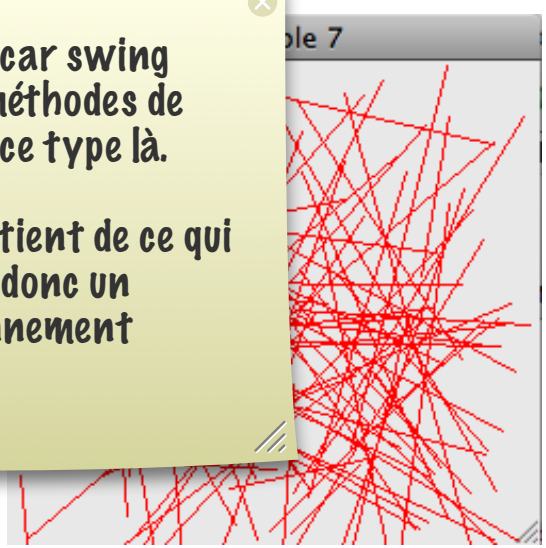
```
public void paintComponent(Graphics g){
  super.paintComponent(g);          // erases background
  Graphics2D g2 = (Graphics2D)g;    //cast for java2
  g2.setRenderingHint(
          RenderingHints.KEY_TEXT_ANTIALIASING,
          RenderingHints.VALUE_TEXT_ANTIALIAS_LCD_HRGB);
  // my graphics:
  g2.setColor(new Color(255,0,0));
  g2.fillRect(10,10,200,50); // left, top, width, height
  g2.setColor(new Color(0,0,0));
  g2.drawString("Hello World", 20, 20);// S, left, BOTTOM
}
```

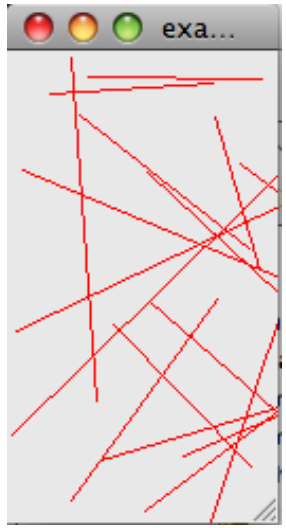
Quelques Exemples Pratiques

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

public class SwingDemo7 extends JFrame {

    public JPanel panel;

    public void init() {
           Container cp = getContentPane();
           this.setTitle("example 7");
           this.setDefaultCloseOperation(EXIT_ON_CLOSE);
           panel = new JPanel();
           cp.add(panel);
    }
    public static void main(String[] args)
    {
            SwingDemo7 frame = new SwingDemo7();
            frame.init();
            frame.setSize(250,250);
            frame.setVisible(true);

            Graphics g = frame.panel.getGraphics();
            Graphics2D g2 = (Graphics2D) g;

            g2.setColor( Color.RED );

            for (int i = 0; i <100 ;++i) {
               g2.drawLine(
                   (int)(250*Math.random()), (int)(250*Math.random()),
                   (int)(250*Math.random()), (int)(250*Math.random()) );
               }
    }
}
```

Graphics2D car swing définit les méthodes de dessin avec ce type là.

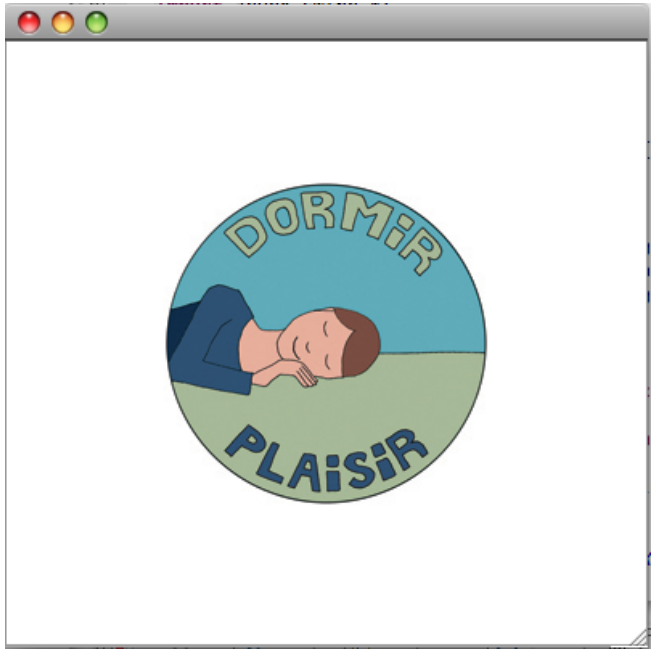
Pas de maintient de ce qui est dessiné, donc un redimensionnement efface tout





après redimensionnement (efface le contenu)

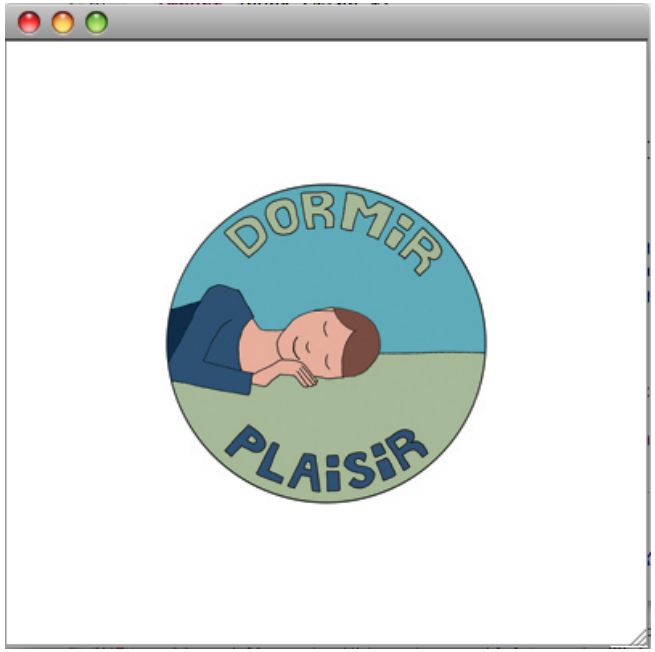
```
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.*;
import javax.imageio.ImageIO;
import javax.swing.*;

public class SwingDemo8 extends JPanel {

        BufferedImage image;

        public SwingDemo8(BufferedImage image) {
                this.image = image;
        }

        protected void paintComponent(Graphics g) {
                int x = (getWidth() - image.getWidth()) / 2;
                int y = (getHeight() - image.getHeight()) / 2;
                g.drawImage(image, x, y, this);
        }

        public static void main(String[] args) throws IOException {
                BufferedImage image = ImageIO.read(new File("image.jpg"));

                SwingDemo8 myDemo = new SwingDemo8(image);
                JFrame f = new JFrame();
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.add(new JScrollPane(myDemo));

                f.setSize(400, 400);
                f.setLocation(200, 200);
                f.setVisible(true);
        }
}
```



après redimensionnement (conserve le contenu)

intérêt de la méthode paintComponent

```
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.*;
import javax.imageio.ImageIO;
import javax.swing.*;

public class SwingDemo9 extends JPanel {

            BufferedImage image;

            public SwingDemo8(BufferedImage image) {
                        this.image = image;
            }

            protected void paintComponent(Graphics g) {
                        g.drawImage(image,
                        0, 0, getWidth(), getHeight(),
                        0, 0, image.getWidth(), image.getHeight(), this);
            // drawImage (image, dst, src, null/this) :
            // dst how many pixels we'll draw,
            // src part of the original image to draw
            }

            public static void main(String[] args) throws IOException {

                        BufferedImage image = ImageIO.read(new File("image.jpg"));

                        SwingDemo9 myDemo = new SwingDemo9(image);
                        JFrame f = new JFrame();
                        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                        f.add(new JScrollPane(myDemo));

                        f.setSize(400, 400);
                        f.setLocation(200, 200);
                        f.setVisible(true);
            }
```



après redimensionnement
change la taille

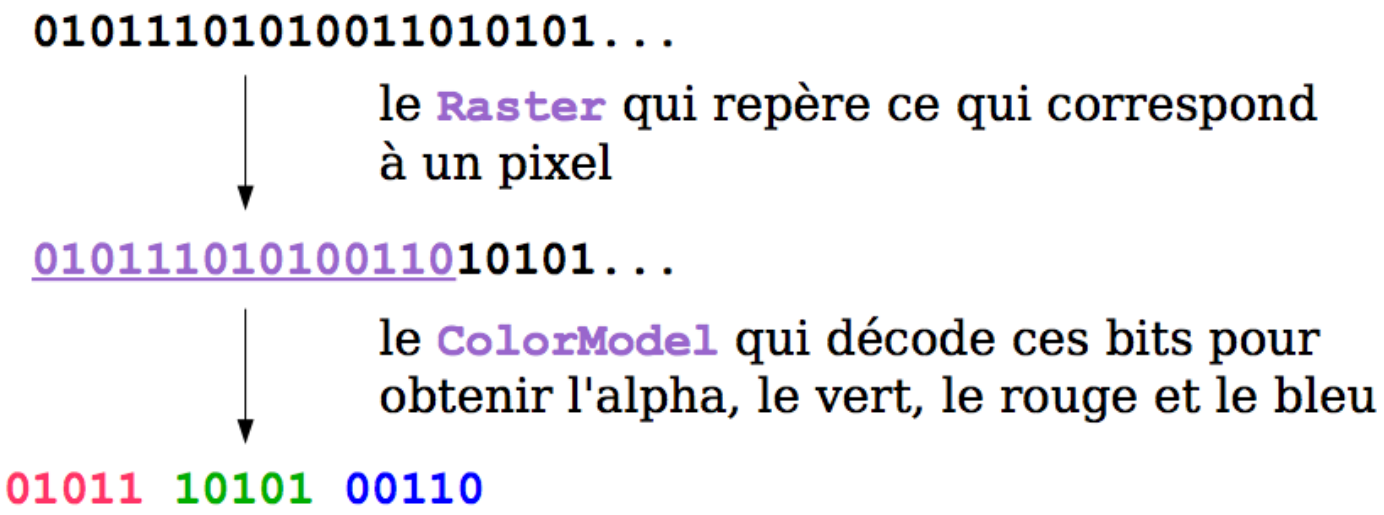
Un mot sur les images

- Source: Sébastien Paumier – C12 – ImageIO

- la classe `java.awt.Image` possède 3 implémentations:
  - `BufferedImage`: tableau de pixels stocké en mémoire
  - `VolatileImage`: image stockée dans la carte graphique
  - `ToolkitImage`: image chargée de façon paresseuse

BufferedImage (1)

- pour passer d'une séquence d'octets représentant une image à un tableau de pixels, il y a 2 intermédiaires:



BufferedImage (2)

- on accède aux pixels de façon générique via le `WritableRaster`, avec les méthodes:
  - `Object getDataElements(int x,int y, Object outData)`
    - obtient un objet représentant un pixel; si `outData` est fourni, il est modifié
  - `void setDataElements(int x, int y, Object inData)`
    - `inData` est censé représenter un pixel
- il existe aussi des méthodes non génériques `setPixel`

BufferedImage (3)

- exemple: récupération des valeurs alpha, rouge, vert et bleu, quel que soit le type de l'image

```
public String getPixelDescription(int x, int y) {
    Object pixel=img.getRaster().getDataElements(x,y,null);
    ColorModel model=img.getColorModel();
    return "alpha="+model.getAlpha(pixel)+" red="+model.getRed(pixel)
           +" green="+model.getGreen(pixel)
           +" blue="+model.getBlue(pixel);
}
```

- il y a 2 façons d'obtenir des images depuis un nom de fichier, une URL ou un flux (`InputStream`):
  - `Toolkit`: permet d'obtenir des images JPG, GIF et PNG (au moins), **en lecture seule**, et au format préféré de l'écran
  - `ImageIO`: permet la lecture et l'écriture, possède un mécanisme de SPI qui permet d'ajouter des jars contenant de nouveaux codecs

Plus sur la manipulation d’images
http://www.javalobby.org/articles/ultimate-image/