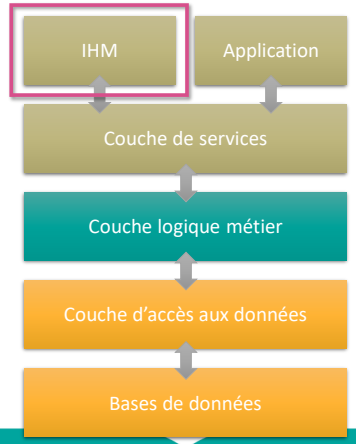
L'Interaction Homme Machine en Java

R309 - Programmation événementielle
R&T 2 – IUT de Blois
Sameh.kchaou@univ-tours.fr
Supports de cours adaptés de Arnaud Soulet

Plan

Introduction

- Architecture orientée services (SOA)



API graphique en Java

API graphique en Java

❖ **Interface avec l'utilisateur**

- La quasi-totalité des programmes informatiques nécessitent
  - **l'interaction entre la machine et l'utilisateur**
  - l'affichage de questions posées à l'utilisateur
  - l'entrée de données par l'utilisateur au moment de l'exécution
  - **l'affichage d'une partie des résultats** obtenus par le traitement informatique
- Cet **échange d'informations** peut s'effectuer avec une interface utilisateur (**UI** en anglais) en mode **texte** (ou console) ou en mode **graphique**

API graphique en Java

❖ **Interface avec l'utilisateur en Java**

- **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 (clic de souris etc…)
- **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.

API graphique en Java

❖ **Interface avec l'utilisateur en Java**

- Java : une langage indépendante de la plateforme
- **L'API Java** doit communiquer avec le **GUI** cible via des « **Adaptateurs**» (entre GUI et boîte à outils)
- Avec quelle **stratégie** ?
  - ✓Faire une utilisation **maximale** du système graphique cible (**AWT**)
  - ✓Faire une utilisation **minimale** du système graphique cible (**SWING**)

API graphique en Java

❖ **Utilisation maximale : java.awt**

- L'objet **TextField** délègue la plupart des 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 (look 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

API graphique en Java

❖ **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

API graphique en Java

❖ **Swing ou AWT ?**

- Tous les composants de **AWT** ont leurs **équivalents dans Swing**
  - ✓ En plus joli
  - ✓ Avec plus de fonctionnalités

Mais Swing est
+ lourd et + lent que AWT….

- Swing offre de nombreux composants qui n'existent pas dans AWT

➔ Il est fortement conseillé d'utiliser les composants Swing

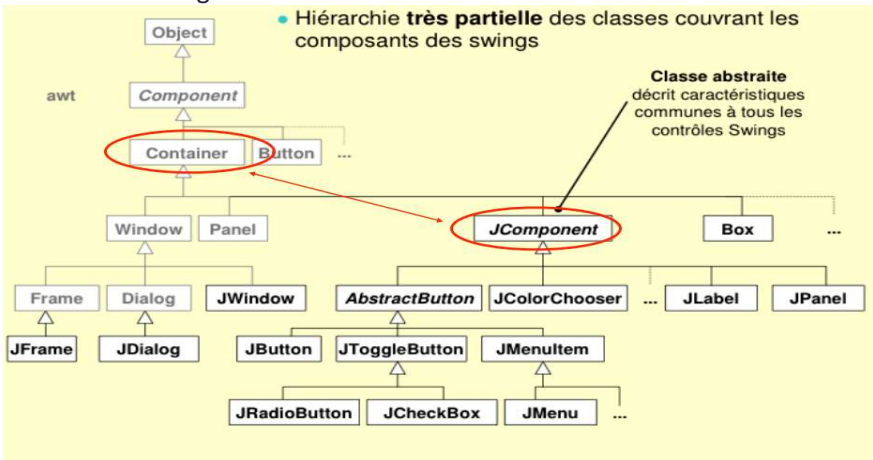
Architecture Swing

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
  - ✓ Des composants atomiques qui présentent les feuilles et
  - ✓ Une racine qui présente le conteneur (top-level).
- Un composant ne peut être contenu qu'une seule fois.

Architecture Swing

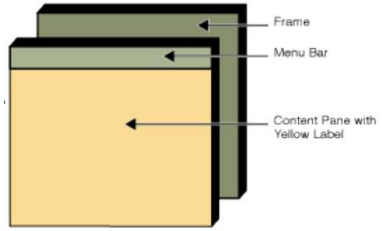
❑ Aperçu des classes Swing



Architecture Swing

❖ **Top-level container : le composant racine**

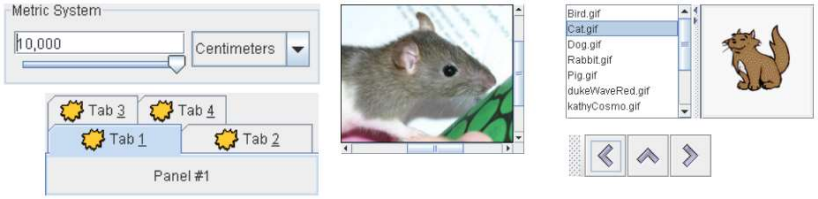
- Une application graphique doit avoir un composant **top-level** comme composant racine (composant qui inclus tous les autres composants).
- Un composant graphique doit pour apparaître faire partie d'une hiérarchie (arbre) d'un conteneur (composant top-level)
- Il en existe 3 types : **JFrame, JDialog et JApplet**
- C'est un conteneur dans lequel vont être disposés les différents éléments constitutifs (composants) de l'interface graphique de l'application (boutons, listes déroulantes, zone de saisie…).



Architecture Swing

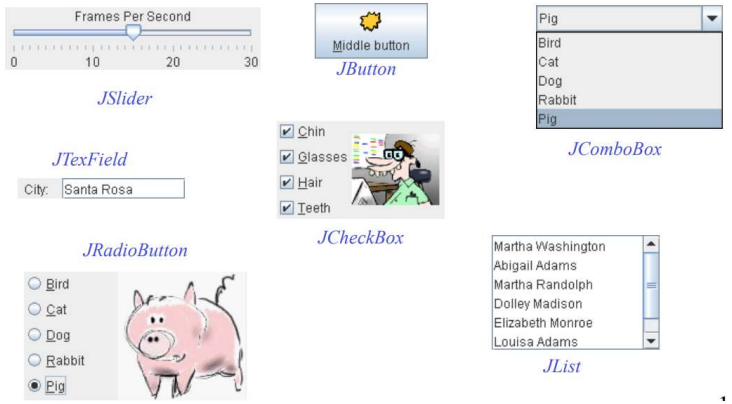
❖ **Les conteneurs intermédiaires**

- 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 ...



Architecture Swing

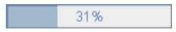
❖ **Les contrôles de bases (interactives, widgets)**



Architecture Swing

❖ **Les composants non-editables**

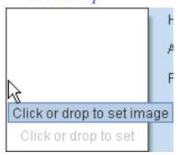
JLabel



JProgressBar

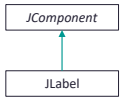
JSeparator

JToolTip

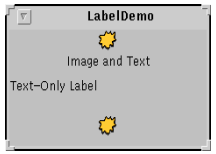


Composants constitutifs

JLabel

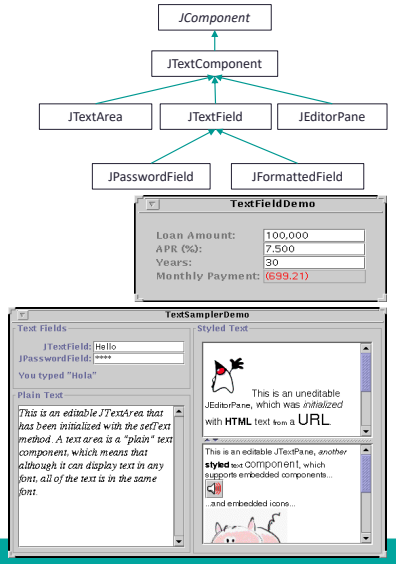


- **Usage : afficher du texte statique et/ou une image**
  - Ne réagit pas aux interactions de l'utilisateur
- **Définition du texte du label**
  - Dans le constructeur : JLabel lb = new JLabel("un label") ;
  - Par la méthode setText : lb.setText("un autre texte pour le label") ;
  - Méthode symétrique : lb.getText() ;
- **Définition d'une icône (Par défaut pas d'image)**
  - Dans le constructeur
  - Par la méthode setIcon : lb.setIcon(new ImageIcon("info.gif") ;
  - Spécification de la position du texte par rapport à l'icône :
    - lb.setVerticalTextPosition(SwingConstants.BOTTOM);
    - lb.setHorizontalTextPosition(SwingConstants.CENTER);
- **Mise en forme du texte**
  - setText supporte du code HTML : lb.setText("<html>ce texte est en <B>gras</B></HTML>")

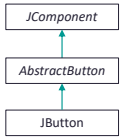


JTextField

- **Usage : afficher un champ de saisie de texte**
- **Création d'un JTextField :**
  - JTextField jtf = new JTextField() ;
  - JTextField jtf = new JTextField("un texte") ;
  - JTextField jtf = new JTextField(20) ;
- **Modifier du texte :**
  - Par interaction de l'utilisateur
  - Par la méthode setText : jtf.setText("le texte") ;
- **Récupérer le texte :**
  - jtf.getText() ;
- **Copier / Coller :**
  - jtf.copy() ou jtf.cut()
  - jtf.paste() ;
- **Interdire la saisie :**
  - jtf.setEditable(false) ;

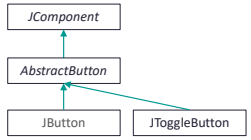


JButton



- **Usage : afficher un bouton permettant de déclencher une action**
- **Création d'un Jbutton :**
  - JButton jbt = new JButton("Titre") ;
  - JButton jbt = new JButton("Titre", icon) ;
- **Association d'une icône**
  - à l'instanciation
  - méthodes setIcon(), setRollOverIcon(), setPressedIcon(), setDisabledIcon()
- **Association d'un raccourci clavier :**
  - jbt.setMnemonic('b') ; // Alt + b
- **Enlever / ajouter la bordure :**
  - jbt.setBorder(false) ;
- **Enlever le cadre indiquant le focus :**
  - jbt.setFocusPainted(false) ;
- **Simuler un clic utilisateur :**
  - jbt.doClick() ;

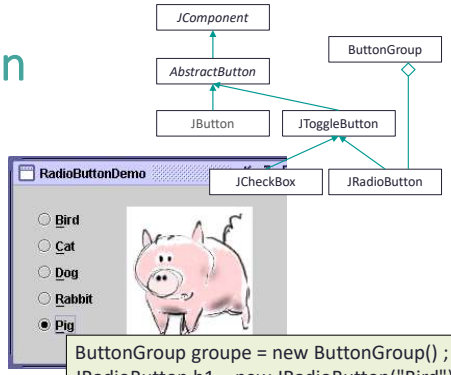
JToggleButton



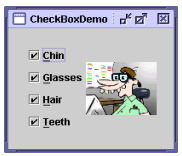
- **Usage : un bouton à deux états**

- **Association d'une icône aux états :**
  - jtbt.setIcon(icon1) <= état « non sélectionné »
  - jtbt.setSelectedIcon(icon2) <= état « sélectionné »
- **Forcer l'état :**
  - jtbt.setSelected(true)
- **Consulter l'état :**
  - jtbt.isSelected() => true ou false

JCheckBox et JRadioButton



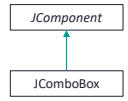
- **Usage :**
  - cases à cocher (états indépendants)
  - boutons radio(états dépendant au sein d'un groupe de boutons)
- **La méthode isSelected() permet de savoir si le bouton est coché ou non**




```
JCheckBox cb1 = new JCheckBox("Chin") ;
JCheckBox cb2 = new JCheckBox("Glasses") ;
JCheckBox cb3 = new JCheckBox("Hair") ;
JCheckBox cb4 = new JCheckBox("Teeth") ;
```

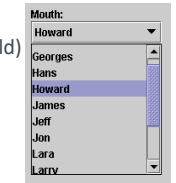
```
ButtonGroup groupe = new ButtonGroup() ;
JRadioButton b1 = new JRadioButton("Bird") ;
JRadioButton b2 = new JRadioButton("Cat") ;
...
b5.setSelected(true) ;
groupe.add(b1) ;
groupe.add(b2) ;
...
```


JComboBox



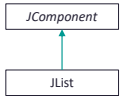
- **Usage : liste déroulante dans laquelle l'utilisateur peut choisir un item**
- **indiquer les items à afficher :**
  - en passant en paramètre du constructeur un tableau d'objets ou un java.util.Vector
    Object[] items = { "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi" } ;
    JComboBox cb = new JComboBox(items) ;
  - en passant par addItem(Object) : cb.addItem("dimanche") ;

- **combo box éditable :**
  - le champ qui représente le texte sélectionné peut être éditable (c'est un JTextField)
    Vector pays = new Vector() ;
    pays.add("Angleterre") ;
    ...
    JComboBox cb = new JComboBox(pays);
    cb.setEditable(true) ;

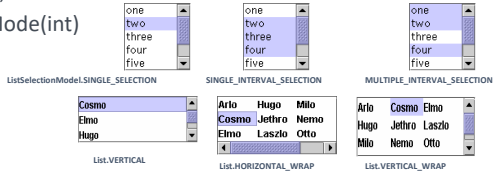


- **récupération de l'item sélectionné :** int getSelectedIndex() ou Object getSelectedItem()

JList



- **Usage : liste déroulante dans laquelle l'utilisateur peut choisir un ou plusieurs item(s)**
- **Indiquer les éléments à afficher**
  - en passant en paramètre dans le constructeur un tableau d'objets ou un java.util.Vector :
    Object[] data = { "lundi", "mardi", "mercredi", "jeudi","vendredi", "samedi" } ;
    JList jl = new JList(data) ;
  - par la méthode setData : jl.SetData(data);
- **Définir le mode de sélection :** setSelectionMode(int)
- **Définir l'arrangement des éléments :**
  setLayoutOrientation(int)
- **Récupérer la sélection**
  int getSelectedIndex()
  int[] getSelectedIndices()
  Object getSelectedValue()
  Object[] getSelectedValues()



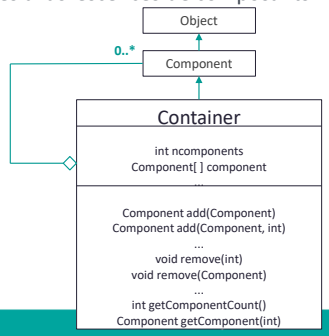
Exemples de capacités communes à tous les composants

- **Activation / désactivation du composant (le composant réagit ou non aux interactions avec l'utilisateur) :**
  - setEnabled(boolean);
  - boolean isEnabled();
- **Visibilité / invisibilité d'un composant :**
  - setVisible(boolean);
  - boolean isVisible();
- **Apparence du curseur (changement de l'apparence du curseur en fonction du composant qu'il survole) :**
  - setCursor(java.awt.Cursor) ; // exemple : bt.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR)) ;
- **Bulles d'aide ("tooltips") :**
  - setToolTopText(String) ;
  - possibilité d'utiliser HTML pour formater le texte d'aide
- **Couleur :**
  - setBackGroundColor(java.awt.Color) ;
  - setForeGroundColor(java.awt.Color) ;

Conteneurs

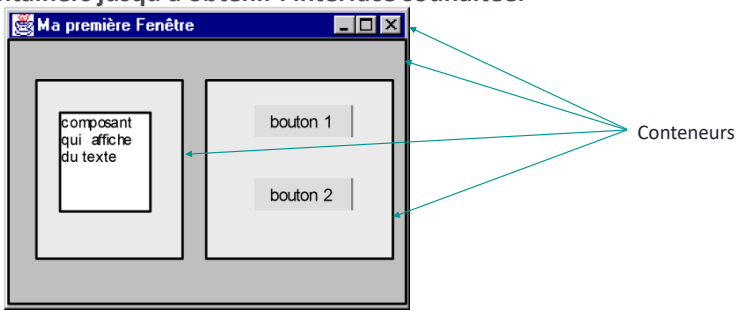
Container

- **Un Container est un composant qui peut contenir d'autres composants.**
  - Un container peut donc contenir d'autres containers.
  - Il permet de créer des arborescences de composants.



Container

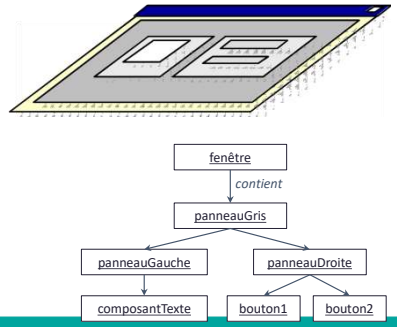
- construction d'une interface : élaborer une arborescence de composants à l'aide de containers jusqu'à obtenir l'interface souhaitée.



Container

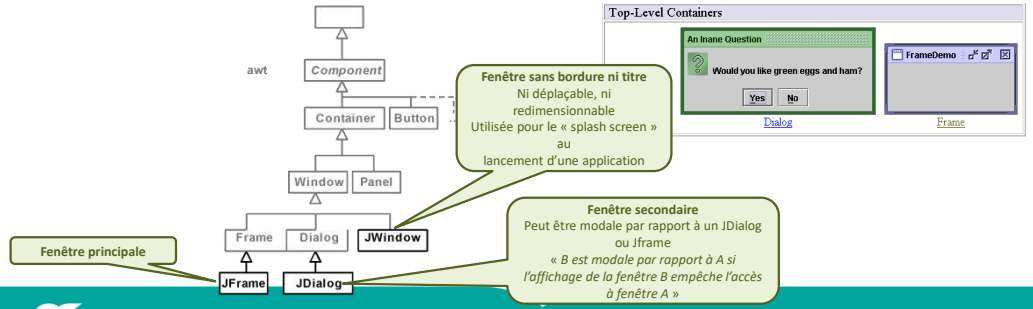
- construction d'une interface : élaborer une arborescence de composants à l'aide de containers jusqu'à obtenir l'interface souhaitée.


```
panneauGauche.add(composantTexte) ;

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

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

fenêtre.add(panneauGris) ;
```



Les fenêtres (Top Level Container)

- Pour apparaître sur l'écran, tout composant doit se trouver dans une hiérarchie de « conteneurs »
- A la racine de cette hiérarchie se trouvent les fenêtres (top level container)



Comportement des fenêtres

- new JFrame(...) ou new JDialog(...) : créer un objet fenêtre mais ne provoque pas son affichage.
- setVisible(true) : afficher la fenêtre, obligatoire pour afficher la fenêtre
- setVisible(false) : cacher la fenêtre , mais ne la détruit pas.
  - l'objet fenêtre et son contenu existent toujours en mémoire,
  - la fenêtre peut être réaffichée ultérieurement.
- dispose() : libérer les ressources utilisées par le système pour afficher la fenêtre et les composants qu'elle contient.
- Attention par défaut à la fermeture d'une fenêtre, elle est simplement rendue invisible :
  - Il est possible d'associer un comportement choisi parmi un ensemble de comportements prédéfinis :

```
setDefaultCloseOperation(int operation)
        WindowConstants.HIDE_ON_CLOSE
        WindowConstants.DO_NOTHING_ON_CLOSE
        WindowConstants.EXIT_ON_CLOSE
```

Comportement des fenêtres

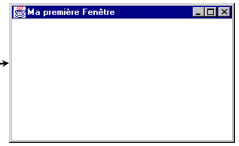
- **setSize(int largeur, int hauteur) :** fixer la taille de la fenêtre
- **setLocation(int x, int y) :** fixer la position de la fenêtre (son coin supérieur gauche) sur l'écran
  - java.awt.Toolkit permet d'obtenir la taille de l'écran :
  - Toolkit tk = Toolkit.getDefaultToolKit() ;
  - Dimension dimEcran = tk.getScreenSize() ;
- **setResizable(boolean) :** autoriser ou non le redimensionnement de la fenêtre par l'utilisateur
- **setTitle(String) :** définir le contenu de la barre de titre de la fenêtre
- **toFront() toBack() :** faire passer la fenêtre au premier plan ou à l'arrière plan

Fenêtre d'application (Jframe)

- **Toute fenêtre d'application est représentée par une classe dérivant ou utilisant la classe JFrame du package javax.swing.**

```
import javax.swing.Jframe ;
public class MyFrame extends JFrame {
   final static int HAUTEUR = 200 ;
   final static int LARGEUR = 300 ;

   public MyFrame() {
      setTitle("Ma première Fenêtre") ;
      setSize(LARGEUR,HAUTEUR) ;
      setVisible(true) ;
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   }
```

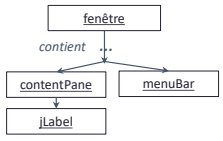
affiche la fenêtre à l'écran et lance un « thread » d'exécution pour gérer interactions sur cette fenêtre

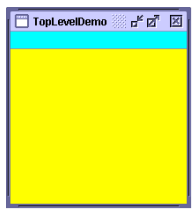


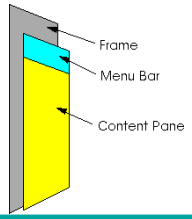
- se comporte comme toute fenêtre du système d'exploitation : peut être redimensionnée, déplacée, …
- La ligne setDefaultCloseOperation permet la fermeture de la fenêtre!
- L'exécution du programme principal ne se termine que lorsque le « thread » lié à la fenêtre se termine lui aussi

Comportement des fenêtres

- **les composants qui seront visibles dans la fenêtre sont placés dans un conteneur particulier associé à celle-ci : Content Pane,**
  - pour récupérer ce conteneur :
    getContentPane()<= Container
- **la fenêtre peut contenir de manière optionnelle une barre de menus (qui n'est pas dans le content pane).**





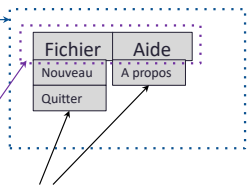


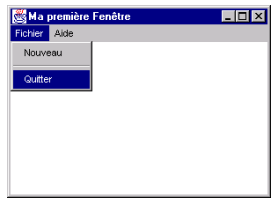
Ajout d'un menu à une fenêtre

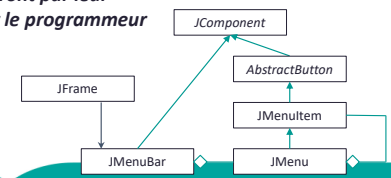
JMenuBar : représente la barre de menu d'une fenêtre

JMenu : options visibles dans la barre de menu

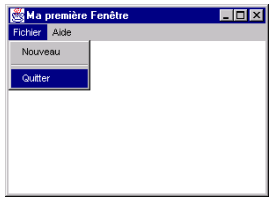
JMenuItem : Items qui déclencheront par leur sélection des actions définies par le programmeur







Création d'un menu




```
import java.swing.*;

public class MenuEditeur extends JMenuBar {
   JMenuItem quitter, nouveau, aPropos;
   JMenu menuFichier, menuAide;
   public MenuEditeur() {

     menuFichier = new JMenu("Fichier");
      nouveau = new JMenuItem("Nouveau");
      quitter = new JMenuItem("Quitter");

      menuFichier.add(nouveau);
      menuFichier.addSeparator();
      menuFichier.add(quitter);

     menuAide = new JMenu("Aide");
      aPropos = new JMenuItem("A propos");
      menuAide.add(aPropos);

      this.add(menuFichier);
      this.add(menuAide);
   }
} // MenuEditeur
```

1) création des éléments du menu

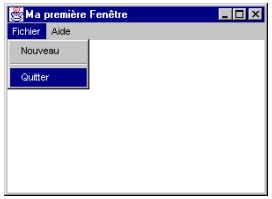
2) association des objets JMenuItem dans un objet JMenu

3) ajout des objets JMenu dans l'objet JMenuBar (this)

Création d'un menu

- **Seule une instance de la classe JFrame (ou JDialog) peut héberger un menu :**

```
import java.swing.* ;

public class MyFrame extends JFrame {

   final static int HAUTEUR = 200 ;
   final static int LARGEUR = 300 ;

   public MyFrame() {
      setTitle("Ma première Fenêtre") ;
      setSize(LARGEUR,HAUTEUR) ;

      setJMenuBar( new MenuEditeur() ) ;

      setVisible(true);
   }

   public static void main(String[] args) {
      new MyFrame() ;
   }
}
```



setJMenuBar prend en paramètre un JMenuBar :

- soit une instance directe de JMenuBar qui aura été modifiée grâce aux méthodes add(…),
- soit une instance d'une classe dérivée de JMenuBar comme dans le cas présent.

Ajout d'un composant à une fenêtre

```
import java.awt.* ;

public class MyFrame extends Frame {

   final static int HAUTEUR = 200 ;
   final static int LARGEUR = 300 ;

   public MyFrame() {
      setTitle("Ma première Fenêtre") ;
      setSize(LARGEUR,HAUTEUR) ;

      Button b = new Button("Mon 1er composant") ;
      this.add(b) ;

      this.setVisible(true) ;
   }
}
```

1) Création du composant
2) ajout au « conteneur »

Un composant ne devrait pas être directement inséré dans une JFrame, mais dans un autre conteneur

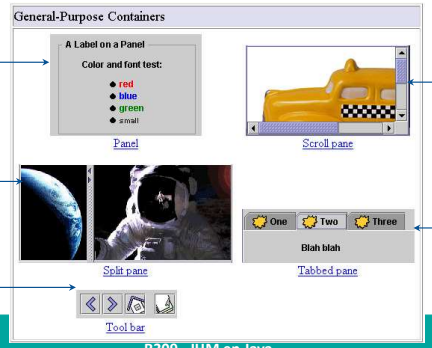
Autres types de containers

- **JFrame et JDialog sont des containers racines (top level containers)**
- **On met jamais des composants dans JFrame ou Jdialog directement**
- **Il existe aussi des containers de type nœud, par exemple :**

JPanel : Aspect réduit au minimum : rectangle invisible dont on peut fixer la couleur de fond utilisé pour regrouper des composants dans une fenêtre

JSplitPane : permet de séparer son contenu en deux zones distinctes dont les surfaces respectives peuvent varier dynamiquement

JToolBar : barre d'outils (regroupe des boutons)



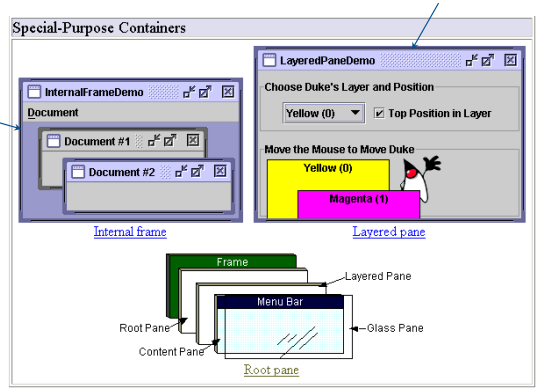
JScrollPane : permet de gérer des ascenceurs (scrollbar) lorsque le composant qu'il contient n'est pas affichable dans sa totalité

JTablePane : tableau à onglets

Autres types de containers

JLayeredPane : fournit une troisième dimension (z : profondeur) pour positionner les composants qu'il contient.

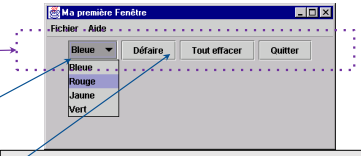
JDesktopPane : permet de définir des fenêtres internes dans une fenêtre.



Imbrication de composants

- **Pour structurer l'interface graphique : utilisation de JPanel,**
  - exemple : ajout d 'une barre d'outils à l'éditeur graphique :

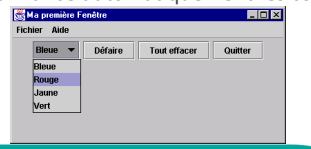
```
import java.awt.* ;
import javax.swing.* ;
public class BarreOutils extends JPanel {

  public BarreOutils() {
    String[] libelleCouleurs = { "Bleue","Rouge","Jaune","Vert" } ;

    this.setBackground(Color.lightGray) ;

    JComboBox listeCouleurs = new JComboBox() ;
    for (int i = 0; i < libelleCouleurs.length; i++)
      listeCouleurs.addItem(libelleCouleurs[i]) ;

    this.add(listeCouleurs) ;
    this.add(new JButton("Défaire")) ;
    this.add(new JButton("Tout effacer")) ;
    this.add(new JButton("Quitter")) ;
  }
} // BarreOutils
```




```
import javax.swing.* ;
public class MyFrame extends JFrame {
  final static int HAUTEUR = 200 ;
  final static int LARGEUR = 300 ;

  public MyFrame() {
    setTitle("Ma première Fenêtre") ;
    setSize(LARGEUR,HAUTEUR) ;
    setJMenuBar(new MenuEditeur());
    this.add(new BarreOutils());
    setVisible(true) ;
  }
}
```

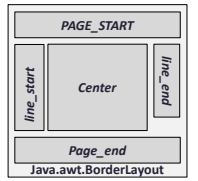

Gestion du placement des composants

- **Par défaut, les composants sont placés automatiquement.**
- **Il est possible d'adapter, le placement aux besoins particulier de l'application**
- **Un LayoutManager est objet associé à un Container qui :**
  - se charge de gérer la disposition des composant appartenant à celui-ci :
    - par exemple le Layout par défaut pour les JPanel fait que :
      - les composants sont placés les uns après les autres dans leur ordre d'ajout,
      - le tout est centré.
  - réordonnance automatiquement les composants lorsque la fenêtre est redimensionnée.



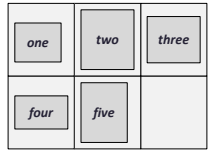
Gestion du placement des composants

- **Principaux gestionnaires de mise en forme implémentant l'interface LayoutManager dans AWT :**
  - BorderLayout : organise la page en 5 parties
  - FlowLayout : comportement par défaut des Jpanels
  - GridLayout : placement dans une grille
  - CardLayout : gère les composants dans des panneaux empilés (un seul est visible)
  - GridBagLayout : placement dans une grille, les cellules peuvent être fusionnées
- **Il est aussi possible de définir ses propres gestionnaires.**



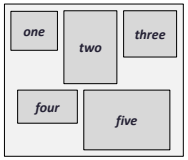
Java.awt.BorderLayout

add(Component, String)



Java.awt.GridLayout

add(Component, index)



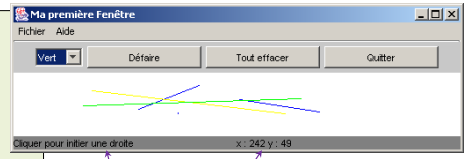
Java.awt.FlowLayout

add(Component)

Exemple de placement avec un GridLayout

- **Définition de la barre d'état de l'application**

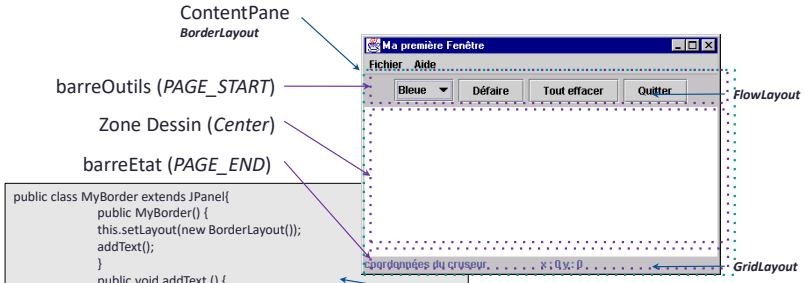
```
import javax.swing.* ;
import java.awt.* ;

public class BarreEtat extends JPanel {
   private JLabel coord, info ;

   public BarreEtat() {
      this.setBackground(Color.darkGray) ;
      this.setLayout(new GridLayout(1,2)) ;
      this.add( new JLabel()) ;
      this.add( new JLabel()) ;
   }

   public void afficheCoord(int x, int y) {
      coord.setText("x : " + x + " y : " + y) ;
   }

   public void afficheInfo(String message) {
      info.setText(message) ;
   }
}
```



Associe un GridLayout en spécifiant le nombre de lignes et colonnes.

Ajout des composants dans les différentes cellules définies par le layout.

Exemple de placement avec un BorderLayout




```
public class MyBorder extends JPanel{
                public MyBorder() {
                this.setLayout(new BorderLayout());
                addText();
                }
                public void addText () {
                JPanel center = new JPanel() ;
                                ...
                this.add(top, BorderLayout.PAGE_START);
                this.add(left, BorderLayout.LINE_START);
                this.add(right, BorderLayout.LINE_END);
                this.add(bottom, BorderLayout.PAGE_END);
                this.add(center, BorderLayout.CENTER);
}
```

Associe un BorderLayout en spécifiant l'espacement entre les composants

Ajout des composants dans les différentes régions définies par le layout

Dimensionnement des composants

- **Dans la barre d'outils les boutons n'ont pas tous la même taille (fixée automatiquement par le layout manager).**
- **Il est possible d 'indiquer une taille avec les méthodes setPreferredSize, setMinimumSize.**

```
public BarreOutils() {
   JComboBox listeCouleurs ;
   String[] libelleCouleurs = {"Bleue", "Rouge", "Jaune", "Vert"} ;
   Color[] couleurs = { Color.blue, Color.red, Color.yellow, Color.green } ;
   this.setBackground(Color.lightGray) ;
   listeCouleurs = new JComboBox() ;
   for (int i = 0; i < libelleCouleurs.length; i++) listeCouleurs.addItem(libelleCouleurs[i]) ;
   this.add(listeCouleurs) ;
   JButton b ;
   this.add(b= new JButton("Défaire")) ;
   b.setPreferredSize(new Dimension(130,25)) ;
   this.add(b = new JButton("Tout effacer")) ;
   b.setPreferredSize(new Dimension(130,25)) ;
   this.add(b = new JButton("Quitter")) ;
   b.setPreferredSize(new Dimension(130,25)) ;
}
```

Indique les dimensions souhaitées, elles ne pourront pas toujours être respectées selon les contraintes et la politique de placement du LayoutManager

Dimensionnement des composants

- **Prise en compte des dimensions souhaitées selon le layout :**

Layout	Hauteur	Largeur
FlowLayout	oui	oui
BorderLayout (line_start,line_end)	non	oui
BorderLayout (page_start, page_end)	oui	non
BorderLayout (Center)	non	non
GridLayout	non	non

- **Il est possible de se passer des services des LayoutManager et de placer les composants « à la main » en indiquant des positions et dimensions exprimées en pixels (setBounds, setSize)**
  - plus de souplesse
  - mais attention lors du redimensionnement des conteneurs (attention à la portabilité de l'interface graphique)

Interface graphique en Swing : résumé

- **Interface graphique = hiérarchie de composants**

- **2 types de composants :**
  - Composant constitutif (ne contient pas d'autres composants)
  - Composant conteneur (contient d'autres composants)

Evénements

Gestion des événements

- **Par défaut, les interactions de l'utilisateur avec les différents composants de l'interface graphique ne provoquent aucune action !**
- **Les applications comportant une interface graphique sont dirigées par les événements (event-driven)**
  - elles ne font rien jusqu'à ce que l'utilisateur bouge la souris, clique un bouton ou appuie sur une touche…
- **Le cœur de toute application comportant une interface graphique est le code de traitement des événements (souvent appelé moteur).**
  - un programme dirigé par les événements est structuré autour d'un modèle de traitement des événements. Bien comprendre ce modèle est important pour une bonne programmation.

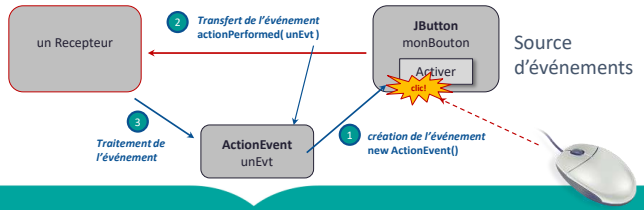
Modèle de gestion des événements

- **Objectifs de conception :**
  - simple et facile à apprendre,
  - séparation nette entre code applicatif et code de l'interface utilisateur,
  - faciliter l'écriture de code robuste pour la gestion des événements ("strong compile time checking"),
- **Ce modèle est utilisé par AWT, Swing et sert également dans de nombreuses API Java.**

Modèle de gestion des événements

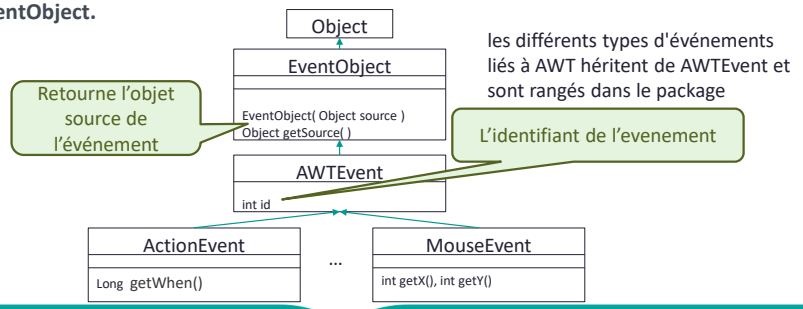
- **Le modèle événementiel se compose de 3 objets:**
  - objets événements
  - objets récepteurs d'événements (« écouteurs »)
  - objets sources d'événements
- **Ces objets interagissent de façon standard en invoquant des méthodes pour permettre le déclenchement et la gestion des événements.**

Récepteur d'événements qui va capter l'événement si celui ci doit être traité

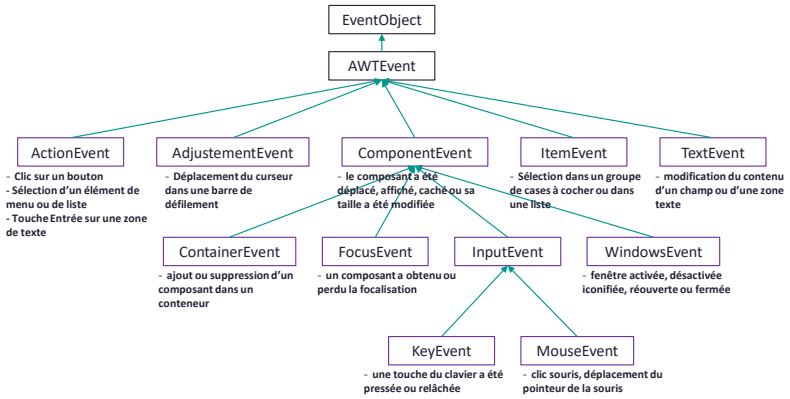


Objets « événement »

- **Un objet événement encapsule l'information spécifique à l'événement.**
  - Exemple : un événement représentant un clic souris contient la position du pointeur souris.
- **Toute classe d'événement (ActionEvent, MouseEvent,…) est une sous classe de la classe java.util.EventObject.**



Extraits de la hiérarchie des événements



Récapitulatifs des principaux événements

Source	Evénements générés	Description
Button	ActionEvent	User clicked the button
Checkbox	ItemEvent	User selected or deselected an item
CheckboxMenuItem	ItemEvent	User selected or deselected an item
Choice	ItemEvent	User selected or deselected an item
Component	ComponentEvent	Component moved, resized, hidden, or shown
	FocusEvent	Component gained or loast focus
	KeyEvent	User pressed or released a key
	MouseEvent	User pressed or released a mouse button, mouse entered or exited a component, or user moved or dragged mouse.
Container	ContainerEvent	Component added to or removed from Container
List	ActionEvent	User double-clicked on a List item
	ItemEvent	User selected or deselected an item
MenuItem	ActionEvent	User selected a menu item
ScrollBar	AdjustmentEvent	User moved the scrollbar
TextComponent	TextEvent	User changed the text
TextField	ActionEvent	User pressed Enter (finished editing text)
Window	WindowEvent	Window opened, closed, iconified, deiconified, or close requested

Objets « Récepteurs d'événements » (écouteurs)

- **Un récepteur d'événements est un objet qui doit être prévenu ("notified") par la source lorsque certains événements se produisent. Le récepteur doit donc être lié à une source qu'il « écoute ».**
- **Les notifications d'événements se font en invoquant des méthodes du récepteur, l'objet événement étant transmis en paramètre.**
- **La source d'événements doit savoir quelle méthode du récepteur doit être appelée :**
  - Pour chaque classe d'événements, une interface spécifique définit les méthodes à appeler pour notifier les événements de cette classe.
    - Exemple : les ActionEvent : méthodes de l'interface ActionListener
  - Toute classe désirant recevoir des notifications d'un événement donné devra implémenter cette interface :
    - Un récepteur d'ActionEvent doit implémenter l'interface ActionListener.

Objets « Récepteurs d'événements » (écouteurs)

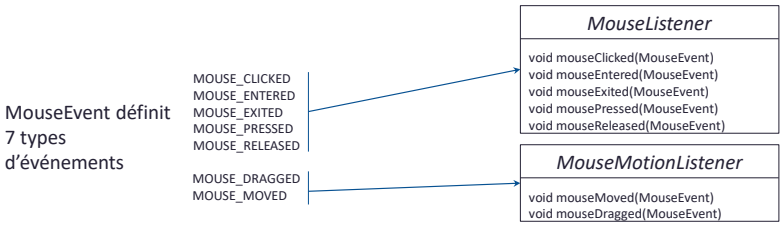
- **Par convention, toutes les interfaces des récepteurs d'événements ont des noms de la forme: <EventType>Listener.**

Classe d'événement	Interface d'écoute
ActionEvent	ActionListener
AdjustmentEvent	AdjustmentListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener
ItemEvent	ItemListener
KeyEvent	KeyListener
MouseEvent	MouseListener/MouseMotionListener
TextEvent	TextListener
WindowEvent	WindowListener

- **Toutes les interfaces d'écoute d'événements héritent de java.util.EventListener.**

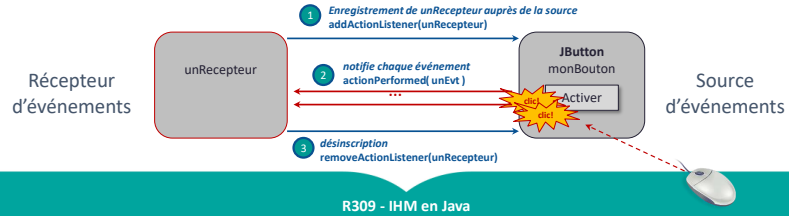
Objets « Récepteurs d'événements » (écouteurs)

- **Une interface d'écoute d'événements peut contenir un nombre quelconque de méthodes, chacune correspondant à un événement différent.**



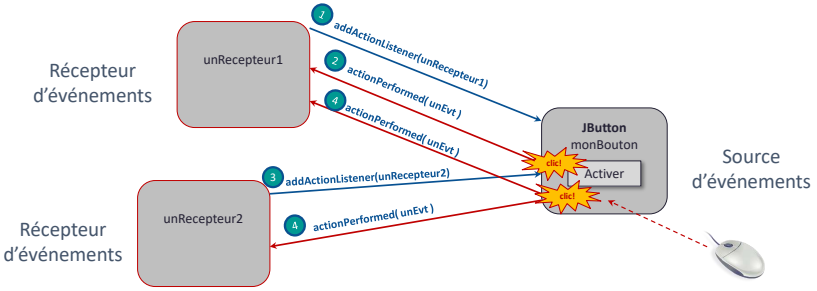
Sources d'événements

- **Les événements sont générés par des sources d'événements (event sources).**
- **Une source d'événements est un objet capable de :**
  - déterminer quand un événement s'est produit
  - d'avertir (notify) des objets récepteurs (event listeners) de l'occurrence de cet événement
- **Pour être averti des événements produits par une source, un récepteur doit préalablement se faire enregistrer auprès de la source :**



Sources d'événements

- Plusieurs récepteurs peuvent être à l'écoute d 'une même source d'événements
- Lorsqu'un événement se produit, il se déclenche vers tous les récepteurs d'événements (multicast)



- Un récepteur peut également être à l 'écoute de plusieurs sources différentes

Sources d'événements

- **Une source d'événements pour une interface d'écoute d'événements propose une méthode d'enregistrement :**
  - public void add<ListenerType>(<ListenerType> listener)
  - Exemple : bouton.addActionListener(MonActionListener)

- **A tout moment, un objet récepteur d'événements peut annuler sa demande de notification**
  - Une source d'événements pour une interface d'écoute d'événements propose une méthode d'annulation de notification :

  public void remove<ListenerType>(<ListenerType> listener)

Interface du récepteur	*Méthodes de l'interface (à définir pour créer un nouveau récepteur)*	*Méthode permettant l'ajout dans l'objet source*	*Type d'événement*
ActionListener	actionPerformed (ActionEvent)	addActionListener()	ActionEvent
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)	addAdjustmentListener()	AdjustmentEvent
ComponentListener	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)	addComponentListener()	ComponentEvent
ContainerListener	componentAdded(ComponentEvent) componentRemoved(ComponentEvent)	addContainerListener()	ContainerEvent
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)	addFocusListener()	FocusEvent
ItemListener	itemStateChanged(ItemEvent)	addItemListener()	ItemEvent
KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)	addKeyListener()	KeyEvent
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	addMouseListener()	MouseEvent
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)	addMouseMotionListener()	MouseEvent
TextListener	textValueChanged(TextEvent)	addTextListener()	TextEvent
WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)	addWindowListener()	WindowEvent

Exemple 1 de gestion des événements

```
public class MyFrame extends JFrame {
     public MyFrame() {
          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          this.setTitle("Test de JFrame");
          this.setSize(1280,900);
     }

     /**
      * Add my JPanel
      * Add a mouse event
      * JPanel is the source
      */
     public void addPanel() {
          JPanel jpl = new JPanel();
          jpl.setBackground(Color.WHITE);
          MyMouseListener ml = new MyMouseListener();
          jpl.addMouseMotionListener(ml);
          this.add(jpl);
     }
}
```

Source → jpl

ml → Listener

Exemple 1 de gestion des événements

```
public class MyMouseListener implements MouseMotionListener {
     @Override
     public void mouseDragged(MouseEvent e) {
          //System.out.println("Mouse dragged");
     }

     @Override
     public void mouseMoved(MouseEvent e) {
          System.out.println("Mouse moved"+e.getX()+" "+e.getY());
     }
}
```

Evénement

Exemple 2 de gestion des événements

```
public class MyFrame extends JFrame {
     public MyFrame() {
          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          this.setTitle("Test de JFrame");
          this.setSize(1280,900);
     }

     public void addPanel() {
          JPanel jpl = new JPanel();
          jpl.setBackground(Color.WHITE);
          jpl.addMouseMotionListener(new MouseMotionListener() {
               @Override
               public void mouseMoved(MouseEvent e) {
               System.out.println("mouse move"+e.getX()+" "+e.getY());
               }
               @Override
               public void mouseDragged(MouseEvent e) {
               }
          });
          this.add(jpl);
     }
}
```

Classe interne anonyme

Gestion des événements : résumé

- **Identifier l'objet à l'origine des événements** (souvent un composant)
- **Identifier le type de l 'événement à intercepter** (en regardant les méthodes du type addXXXListener)
- **Créer une classe implémentant l'interface associée à l'événement à gérer :**
  - Soit celle du composant (ou du conteneur du composant) à l'origine de l 'événement (« facilité » d 'implémentation)
  - Soit une classe indépendante qui détermine la frontière entre l 'interface graphique (émission des événements) et ce qui représente la logique de l 'application (traitement des événements)

  ➔ Séparer = faciliter l'évolution du logiciel
- **Implémenter dans cette classe les méthodes associées à l'événement**
- **Utiliser l'événement passé en paramètre pour accéder aux informations utiles sur l'événement**

Adaptateurs d'événements

Adapateurs d'événements

- **Exemples introductifs**
  - Fermeture de fenêtre
  - Dessin d'un segment
  - Barre d'outils
- **Notion d'adaptateur**
- **Classes internes**
  - Classes locales
  - Classes anonymes

Exemple introductif : Fermeture de fenêtre

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

public class EditeurGraphique extends JFrame {
   final static int HAUTEUR = 450;
   final static int LARGEUR = 750;

   public EditeurGraphique {
      BarreEtat barreEtat = new BarreEtat() ;
      setTitle("Ma première Fenêtre") ;
      setSize(LARGEUR,HAUTEUR) ;
      setJMenuBar(new MenuEditeur()) ;
      this.getContentPane().add(new BarreOutils(),"North") ;
      this.getContentPane().add(new ZoneGraphique(barreEtat),"Center") ;
      this.getContentPane().add(barreEtat,"South") ;
      barreEtat.afficheInfo("coordonnées du cruseur") ;
      MyClosingListener mcl = new MyClosingListener(this);
      this.addWindowListener(mcl);
      setVisible(true) ;
   }


   public static void main(String[ ] args) { new EditeurGraphique() ; }

} // MyFrame
```

L'objet MyClosingListener est récepteur des WindowEvent que l'EditeurGraphique est susceptible de générer,

Maintenant il faut implémenter les méthodes de l'interface WindowListener.

Exemple introductif : Fermeture de fenêtre

WindowEvent définit 7 types d'événements :

WINDOW_ACTIVATED
WINDOW_CLOSED
WINDOW_CLOSING
WINDOW_DEACTIVATED
WINDOW_DEICONIFIED
WINDOW_ICONIFIED
WINDOW_OPENED

WindowListener définit donc sept méthodes

WindowListener

windowActivated(WindowEvent)
  Invoked when a window is activated
windowClosed(WindowEvent)
  Invoked when a window has been closed.
windowClosing(WindowEvent)
  Invoked when a window is in the process of being closed.
windowDeactivated(WindowEvent)
  Invoked when a window is de-activated.
windowDeiconified(WindowEvent)
  Invoked when a window is de-iconified.
windowIconified(WindowEvent)
  Invoked when a window is iconified.
windowOpened(WindowEvent)
  Invoked when a window has been opened.

```
public class MyClosingListener implements  WindowListener{
                private JFrame myMainFrame;
                public MyClosingListener(final JFrame jf) {
                                this.myMainFrame = jf;
                }
                @Override
                public void windowActivated(WindowEvent e) {
                                // TODO Auto-generated method stub

                }
                @Override
                public void windowClosed(WindowEvent e) {
                                // TODO Auto-generated method stub
                }
                @Override
                public void windowClosing(WindowEvent e) {
                                if(JOptionPane.showConfirmDialog(myMainFrame,"Voulez-vous vraimment
quitter ?",      null, JOptionPane.YES_NO_OPTION) == 0)
                                                System.exit(0);
                }
```

Seul windowClosing nous intéresse, mais quand on implémente une interface il faut

```
public void windowActivated(WindowEvent e){ }
..
public void windowOpened(WindowEvent e){ }
```


Exemple introductif : Fermeture de fenêtre

- **Solution : Utiliser un adaptateur d'événements de la classe WindowAdapter du package java.awt.event :**

Il faut étendre la classe WindowAdapter, pour permettre de n'implémenter que la méthode qui nous concerne :

Java(TM) 2 Platform, Standard Edition, v1.2.2 API Specification: Class WindowAdapter ...

Fichier Edition Afficher Aller Communicator Aide

public abstract class **WindowAdapter**
extends Object
implements WindowListener

An abstract adapter class for receiving window events. The methods in this class are empty. This class exists as convenience for creating listener objects.

Extend this class to create a `WindowEvent` listener and override the methods for the events of interest. (If you implement the `WindowListener` interface, you have to define all of the methods in it. This abstract class defines null methods for them all, so you can only have to define methods for events you care about.)

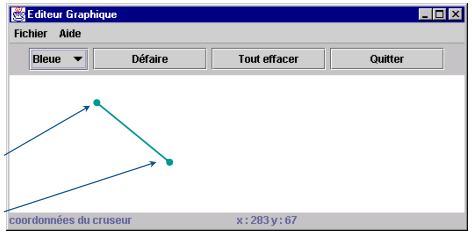
```
public class ClosingAdapter extends WindowAdapter {

   public void windowClosing(WindowEvent e) {
      if(JOptionPane.showConfirmDialog(this,
                  "Voulez-vous vraimment quitter ?",
                  null, JOptionPane.YES_NO_OPTION) == 0)
         System.exit(0);
   }
}
```

```
public class EditeurGraphique extends JFrame {

public EditeurGraphique() {
      ...
      this.addWindowListener( new ClosingAdapter( ) ; ) ;
      this.setVisible(true);
   }

   public static void main(String[] args){ new EditeurGraphique(); }
}
```

Exemple introductif : Dessin de segments

❑ **Gestion des événements souris pour tracer des segments :**

Le déplacement de la souris sur la zone de dessin met déjà à jour les coordonnées du curseur dans la barre d'état grâce à notre gestion des événements : MOUSE_MOVED et MOUSE_DRAGGED.



Nouvel Objectif :

pouvoir appuyer sur un bouton de la souris (MOUSE_PRESSED) pour définir le début d'une droite

et relâcher ce bouton (MOUSE_RELEASED) pour en définir la fin.

- Type d 'événement : MouseEvent
- Source : la zone de dessin
- Récepteur : la zone de dessin
  ➔ doit implanter MouseListener

MouseListener
void mouseClicked(MouseEvent)
void mouseEntered(MouseEvent)
void mouseExited(MouseEvent)
void mousePressed(MouseEvent)
void mouseReleased(MouseEvent)

Mais seules deux des méthodes nous intéresse :

Exemple introductif : Dessin de segments

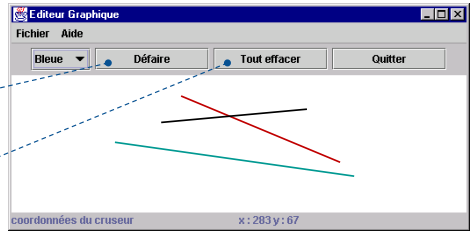
```
import java.awt.* ;
import javax.swing.* ;
import java.awt.event.* ;

public class ZoneDessin extends JPanel implements MouseMotionListener {
   private BarreEtat be ;

   public ZoneDessin(BarreEtat be) {
      setBackground(Color.white) ;
      setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR)) ;
      this.be = be ;
      addMouseMotionListener(this) ;
      addMouseListener(new GestionnaireClic(this)) ;
   }

   public void mouseMoved(MouseEvent e) { be.afficheCoord(e.getX(),e.getY()); }

   public void mouseDragged(MouseEvent e) { be.afficheCoord(e.getX(),e.getY()); }

   public void initieDroite(int x, int y) {
      be.afficheMessage(" Relacher pour dessiner la droite ");
   }
   public void termineDroite(int x, int y) {
      be.afficheMessage(" Cliquer pour initier une droite ");
   }
} // ZoneDessin
```

```
import java.awt.event.*;
public class GestionnaireClic extends MouseAdapter {
   private ZoneDessin zone ;

   public GestionnaireClic(ZoneDessin z) { zone = z ; }

   public void mousePressed(MouseEvent e) {
      zone.initieDroite(e.getX(),e.getY()) ;
   }

   public void mouseReleased(MouseEvent e) {
      zone.termineDroite(e.getX(),e.getY()) ;
   }
}
```

Pour ne pas avoir à définir des méthodes inutiles il est possible d'utiliser un adaptateur d'événements : MouseAdapter

Exemple introductif : Barre d'outils

❑ **Gestion des boutons de la barre d'outil**



Annule le dernier tracé

Efface toute la zone de dessin

- Type d 'événement : ActionEvent
- Source : les boutons (Jbutton - classe de la barre d'état)
- Récepteur : ????
  ➔ doit implanter ActionListener

L'activation des boutons doit se traduire par un changement d'état de la zone de dessin.
Il semble donc naturel que cela soit la zone dessin qui récupère les événements et réalise les traitement adéquats.
Mais la zone dessin et les boutons ne sont pas instanciés dans la même classe..

Exemple introductif : Barre d'outils

1) C'est la zone dessin qui effectue les traitements déclenchés par les boutons efface et annule

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

public class ZoneDessin extends JPanel
                implements MouseMotionListener, ActionListener {
   private BarreEtat be ;

   public ZoneDessin(BarreEtat be) { ....}
   public void mouseMoved(MouseEvent e) { ... }
   public void mouseDragged(MouseEvent e) { ... }
   public void initieDroite(int x, int y) { ... }
   public void termineDroite(int x, int y) { ... }

   public void efface() { ... }
   public void annule() { ... }

   public void actionPerformed(ActionEvent e) {
      selon l 'origine de l'événement cette méthode
      appelle : efface() ou annule()
   }
} // ZoneGraphique
```

2) la zone dessin reçoit les événements issus les boutons

Mais le code zone dessin dépend du code de la barre d'outils.

Lourdeur du code si on décide que le listener des boutons est la zoneDessin qui est déjà un panel :

- si beaucoup de boutons à écouter
- si plusieurs moyens de déclencher ces actions (boutons, menus…)

Adaptateurs : intérêt

- **Le modèle événementiel de JAVA peut rapidement devenir difficile à gérer :**
  - Si un objet doit être à l'écoute d'un grand nombre d'objets sources (car implementer des interfaces pour chaque objet).
  - Si un objet est à l'écoute d'événements issus de deux objets (ou d'avantage) sources d'événements du même type (car découvrir qui a émis l'événement).
- **Cela peut conduire à du code difficile à lire et/ou difficile à écrire.**

➔ **Introduire un objet "médiateur" entre la source d'événements et le récepteur d'événements** (adapter la source aux besoins spécifiques ➔ le nom d'adaptateur d'événements)

Adaptateurs : difficulté

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

public class AdaptateurEfface implements ActionListener {

   ZoneDessin zone ;

   public adaptateurEfface(ZoneDessin z) { zone = z ; }

   public void actionPerformed(ActionEvent e) { zone.efface() ; }
}
```

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

public class AdaptateurAnnule implements ActionListener {

   ZoneDessin zone ;

   public adaptateurAnnule(ZoneDessin z) { zone = z ; }

   public void actionPerformed(ActionEvent e) { zone.annule() ; }
}
```

Utilisation de classes adaptateurs personnalisées dans la classe BarreOutils :

```
...
public BarreOutils(ZoneDessin zd) {
   ...
   JButton b ;
   this.add(b= new JButton("Défaire")) ;
   b.addActionListener( new AdaptateurAnnule(zd) ) ;
   this.add(b = new JButton("Tout effacer")) ;
   b.addActionListener( new AdaptateurEfface(zd) ) ;
   ...
}
...
```

Intérêts :
- Simplification de ZoneDessin,
- Meilleur découplage des différents éléments de l'interface utilisateur.

Mais :
- multiplication du nombre de classes et donc de fichiers,
- classes pas vraiment réutilisables.

Classes internes

- **Solution : utiliser des classes imbriquées et internes**

- **2 types de classes internes :**
  - **Classe interne locale** : définition de la classe à l'intérieur d'une classe, au même niveau que les attributs et méthodes :

```
class X {
    class Inner extends Superclass { … }
}
```

  - **Classe interne anonyme :** définition de la classe à l'intérieur d'une expression en créant directement un objet de cette classe et sans la nommer :

```
myclass.work(new AnonymClass{ … } );
```


Classes internes locales

- **Classes inner:**

```
class X {
   class Inner extends Superclass {…}
   …
}
```

- **Toute instance d'une classe interne est associée de manière interne à l'instance de la classe englobante dans laquelle elle est créée**
- **Accès implicite de cette instance aux membres (attributs/méthodes) définis dans l'instance de la classe englobante (y compris les membres privés).**

Classes anonymes

- **Classe anonyme : définition de la classe à l'intérieur d'une expression en créant directement un objet de cette classe et sans la nommer :**

```
myclass.work(new AnonymClass{
} );
```

- **Elle permet de regrouper définition et instanciation d'une classe member.**
- **Les classes anonymes sont très couramment utilisées en tant qu'adaptateurs d'événements.**
- **Une classe anonyme n'ayant pas de nom : il n 'est pas possible de définir de constructeur dans son corps.**

Classe classique vs classe interne

- **Comment choisir entre écrire une classe classique ou écrire une classe interne :**
  - ➔ choisir ce qui rend le code plus clair
- **Choisir une classe interne si :**
  - La classe doit avoir un corps petit.
  - Une seule instance de la classe est necessaire.
  - La classe anonyme est utilisée/instanciée juste après qu'elle soit définie.
  - Vous avez besoin d'hériter de plusieurs classes.
- **Choisir une classe classique si :**
  - La classe doit avoir plusieurs instances.
  - Votre architecture est amenée à évoluer et doit être facile à maintenir.

Quizz!

Quizz!

- **Que se passe t'il si on clique sur la croix d'une Jframe ?**

Quizz!

- Qu'est ce qu'un JPanel ? A quoi cela sert ?

Quizz!

- Comment ajouter et organiser le placement des composants dans un conteneur ?

Quizz!

- Quelles sont les notions essentielles de la gestion des événements ?

Quizz!

❑ L'interface ActionListener :

A- Sert à gérer les événements relatifs à la souris et au clavier.
B- Sert à gérer des événements comme choix dans un menu, clic sur un bouton, appui sur le bouton « Entrée » dans un champ de texte.
C- Doit implémenter la méthode actionPerformed.
D- Doit implémenter les méthodes mouseClicked et keyTyped.

Quizz!

- **Qu'est ce qu'un adpatateur ?**