CNAM – Aix en Provence

Valeur C - RSM

Conception d'Applications Multimedia

Programmation Multimedia

Session 2

Erwan TRANVOUEZ
Maître de Conférences

erwan.tranvouez@polytech.univ-mrs.fr
http://erwan.tranvouez.free.fr

Objectifs de la session :

- Suite des bases de programmation
- Programmation d'Interfaces Utilisateur Graphiques en Java (IHM, …)

Plan du cours

1. Suite base de programmation java

- Utilisation des exceptions Java
- Définir ses propres exceptions (héritage et polymorphisme)
- Gestion des flux

Rôle des exceptions

- Essentiellement, la gestion d'erreurs.
- Pourquoi ne pas utiliser des tests ou renvoyer un code d'erreur ?
  - le traitement est mélangé dans le code : peu explicite;
  - difficile ou lourd de renvoyer un résultat **et** un code d'erreur;
  - code d'erreur pas très explicite;
  - ne permet pas de traiter des erreurs fatales;
- **De plus, en forçant leur prise en compte avec des blocs try-catch, cela souligne l'existence de problèmes potentiels pouvant conduire à les traiter** (ex. le fichier n'existe pas)**.**
- **Une exception est un objet … avec toutes les propriétés que cela implique.**

Exceptions prédéfinies

- **Le nom d'une exception doit indiquer le type d'erreur rencontrée.**
- **Quelques exemples**
  - **ArrayOutOfBoundException** : dépassement de tableau
  - **NullPointerException** : tentative d'accès à une réference nulle.
  - **IOException** : erreur d'entrée/sortie.
  - **SQLException** : erreur d'accés à une base de donnée.
  - **RunTimeException** : erreur détectée par la machine virtuelle lors de l'exécution du programme.
  - **SecurityException** : violation de sécurité.
  - **ArithmeticException** : erreurs mathématiques (division par zéro, …)
- **Exemple de message d'erreur :**
  - **indique** : **la méthode d'origine** de l'erreur (main) **le type d'exception** (NullPointerException) et **la ligne d'origine** de l'erreur (12)**.**

```
Exception in thread "main" java.lang.NullPointerException
        at TestFlux.main(TestFlux.java:12)
```


Utilisation d'exceptions

- Le bloc try – catch - finally

```
import java.io.*;
public class Utils {
 public static String readLine(String fileName) {
  String s;
  BufferedReader in = null;
  try {
    in = new BufferedReader(new FileReader(fileName));
    s = in.readLine();
  }catch( IOException io) {
   System.out.println("Erreur avec le fichier "+fileName);
   io.printStackTrace();
   s=null;
  }finally {
     try {    in.close();   }
     catch( IOException io) {
         System.out.println("Erreur lors de la fermeture de "+fileName);
         s=null;
     } // fin bloc try-catch
  } // fin bloc try-catch-finally
  return s;
 }
}
```

Le code ci-dessous créer un gestionnaire de flux et lui demande de lire 1 ligne.

Le bloc **try** signale que l'on va exécuter du code qui peut lever une exception.

Le bloc **catch** indique que faire le cas **échéant**. Ici, cela se limite à un affichage.

printStackTrace affiche la liste des appels de méthodes ayant conduit à cette erreur.

Le bloc **finally** assure que certaines opérations seront effectuées quoi qu'il arrive. Ici, il s'agit de fermer le fichier.

Polymorphisme et Exception


```
import java.io.*;
public class Utils {
 public static int testFile(Reader f) throws NullPointerException{
  int n, test=0;

   if(f == null)
       throw new NullPointerException("Utils.testFile()");

   try {
         BufferedReader in = new BufferedReader(f);
         n = Integer.parseInt(in.readLine());
   }catch( IOException io) {1
     io.printStackTrace();
     test=20;
   }catch( NumberFormatException nfe) {2
     test=10;
   } catch ( Exception e) {3
     test=100;
   }

  return test;
 }
}
```

Cette méthode peut renvoyer une exception. L'utilisateur de la méthode doit en prendre compte.

Afin de signaler la gravité de l'erreur on **lève** une exception en "**lançant**" une **instance** de la classe **NullPointerException**

Si une exception est levée dans ce bloc elle traversera les blocs catch 1,2 et 3 jusqu'à qu'elle soit reconnue.
Le bloc 3 assure que toute exception sera reconnue, Exception étant la classe mère de toutes les exceptions.

Exemple: si la chaîne lue clavier est "ahah", **parseInt** va lever une **NumberFormatException** qui sera attrapée en **2**. Ceci fait ensuite **sortir** du bloc try-catch et exécute les instructions suivantes (ici return test;).

Créer ses propres exceptions

- Il suffit de créer une classe héritant de Exception.

```
import java.io.*;

class AimePasEpinardsException extends Exception{
  Enfant e=null;
  public AimePasEpinardsException (Enfant e) { this.e=e;}
  public String toString() { return e + " aime pas les épinards."); }
}

class Nourriture {}
class Epinard extends Nourriture {}
class Enfant{
 public void manger(Nourriture n) throws AimePasEpinardsException {
  if( n instanceOf Epinards) throw new AimePasEpinardsException(this);}
}

public class Nourrice {
 public void aTable(Enfant e, Nourriture n) {
  try {
   e.manger(n);
  }catch(AimePasEpinardsException a) {
    menacer(e);
  }
 }
}
```

Se déclare comme une classe normale.

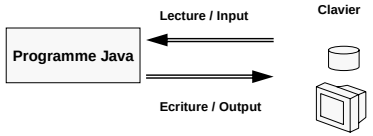
S'utilise comme une exception normale.

Exercice sur les exceptions

- Créer une classe de type Wrapper sur le type primitif integer.
  - utiliser la methode static Integer.parseInt(String) sachant qu'elle peut lever une exception NumberFormatException.
- Créer un tableau dynamique qui considère comme un erreur grave l'ajout d'éléments vide, et accepte toutes les classes sauf la classe Integer.

Principe de la gestion de flux en Java

- Il s'agit d'assurer la gestion des transmissions d'information vers ou depuis un programme java **indépendamment de la plateforme d'exécution**.
- La **lecture** et **l'écriture** d'informations sont traitées séparément au travers de **classes** dédiées (InputStream/OutputStream).

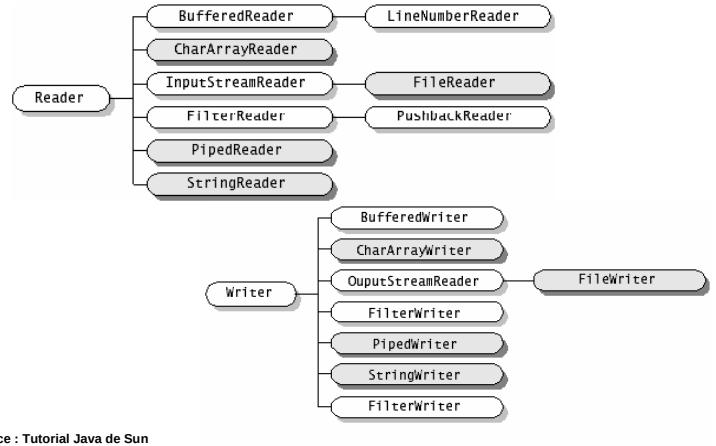


Principe de la gestion de flux en Java

- Le principe est de partir de l'information au plus bas niveau (octets) au plus haut niveau (caractères, son, video) en enveloppant ces flux avec des traitements adaptés.
- Paquetage concerné : java.io.

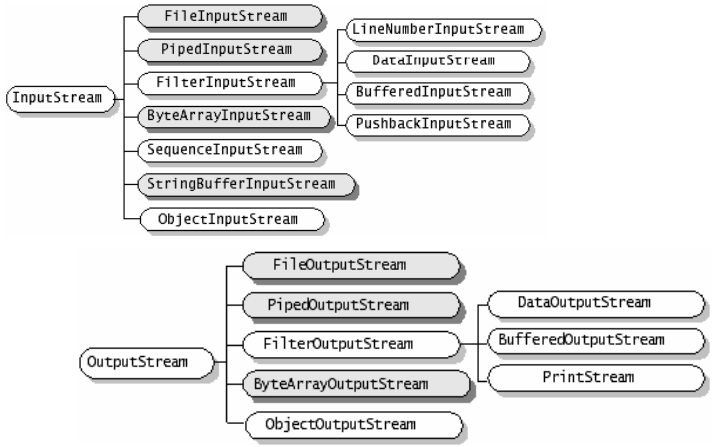
Fichier
Connection réseau

Gestionnaire de flux disponibles : Flux de caractères



Source : Tutorial Java de Sun

Gestionnaire de flux disponibles : Flux de d'octets



Source : Tutorial Java de Sun

E. Tranvouez

Exemple : les flux standards

- Flux standards :
  - **System.in** : flux de lecture standard (clavier). C'est une variable statique instanciant la classe InputStream.
  - **System.out** : flux d'écriture standard (écran). Idem mais instanciant OutputStream.
  - **System.err** : idem.
- **Exemple :**

```
package flux;
import java.io.*;

public class TestFlux {
 public static void main(String args[]) {
  int b = 0;
  try {
     b = System.in.read();
     System.out.println(b);
  }catch( IOException io) {
     S.o.p ("Erreur lors de la lecture du flux!");}
 }
}
```

Bloc **try-catch** permet d'intercepter les exceptions levées suite à des erreurs de lecture dans le flux System.in

La méthode read() renvoie le code du prochain octet présent dans le flux.

⇒

```
C:\>java flux.TestFlux
a↵
97
C:\>
```

E. Tranvouez

Utilisation des flux (JDK 1.1)

- Le traitement de flux d'octets ou de caractère en soit est fastidieux, lorsqu'une phase de reconnaissance est nécessaire (chiffres, mots,…).
- Il est possible d'ajouter une ou plusieurs classes intermédiaires chargées d'effectuer un pré-traitements des données lues ou écrites.
- Exemple pour la lecture du flux standard:
  - **InputStreamReader** : convertit le flux d'octets (InputStream – représentation 8-bits) en flux de caractères unicodes (représentation 16-bits).
  - **BufferedReader** : ajoute la gestion d'un tampon améliorant les performances de lecture. Ajoute une méthode readLine() renvoyant un String.
- Exemple pour l'écriture dans un fichier :
  - **FileWriter** : permet d'écrire dans des fichiers texte.
  - **BufferedWriter** : ajoute la gestion d'un tampon.
  - **PrintWriter** : permet une écriture formatée de caractères.

E. Tranvouez

Exemple de lecture/écriture complet (JDK 1.1)


```
package Flux;
import java.io.*;

public class Test {

 public static void main(String arg[]) {
    int nbLignes=0;
    String s1=null, s2 = "/home/chezmoi/test.txt";
    BufferedReader in = new BufferedReader( new InputStreamReader(System.in
    PrintWriter out = null;
    try {
      out = new PrintWriter( new BufferedWriter( new FileWriter(s1,true) ) );
    }catch(IOException e) {
         System.out.println("Erreur lors de l'ouverture du fichier " +s1);   }

    try {
      do {
        s2 = in.readLine();
        out.write(s2+"\n");
        nbLignes++;
    } while (s2.length() != 0);
      out.print("Nombre de lignes écrites : "); out.println(nbLignes);
      out.close();
    } catch ( IOException e )  {
      System.out.println("Erreur lors en manipulant le fichier " + s1);
    }
}
}
```

Nom et chemin du fichier. Description plus complète possible avec File

2nd «empaquetage»

1er «empaquetage»

Le 1er paramètre contient le nom du fichier. Le 2nd indique si le fichier est en mode **ajout** ou non.

Lit une ligne au clavier.

Ecrit la ligne dans le fichier

Fermeture du fichier

Une exception peut être levée : par in.readLine(), par out.write() ou close().

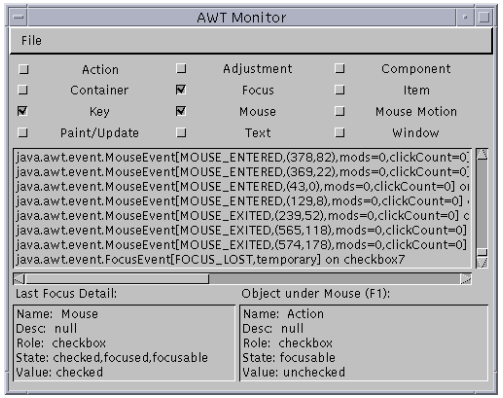
2. Programmation d'IHM

- Composants graphiques
- Gestion des évenements

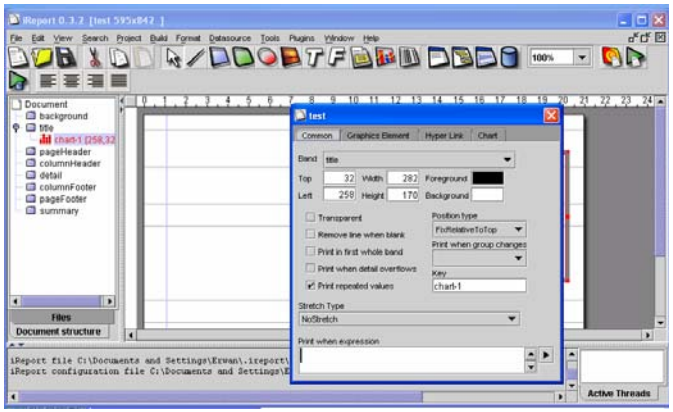
Architecture d'une application graphique

- Evolutions de la conception de GUI (Graphical User Interface) en java :
  - **AWT (Abstract Window ToolKit)** : Première librairie de composants graphiques. Présentait l'avantage d'une exécution rapide mais d'une esthétique spartiate et parfois non uniforme selon les plateforme. Liés au système de fenêtrage de la plateforme cible (composants lourds – *heavyweight component*).
  - **SWING** : Ajoute des composants plus complexes (composants léger – lightweight component) tenant la comparaison avec Windows.
  - Depuis : **SWT** … suit le principe d'AWT cad composant lourd (code java + dll windows par exemple).
- Respect de la philosophie « Compile Once, Run Everywhere » avec en plus « look the same ».

Exemple AWT

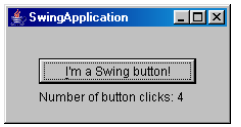
- 

Exemple SWING



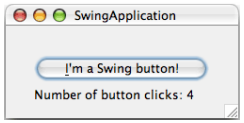
Exemple de fenêtre Swing multiPlateforme

Java Look&Feel



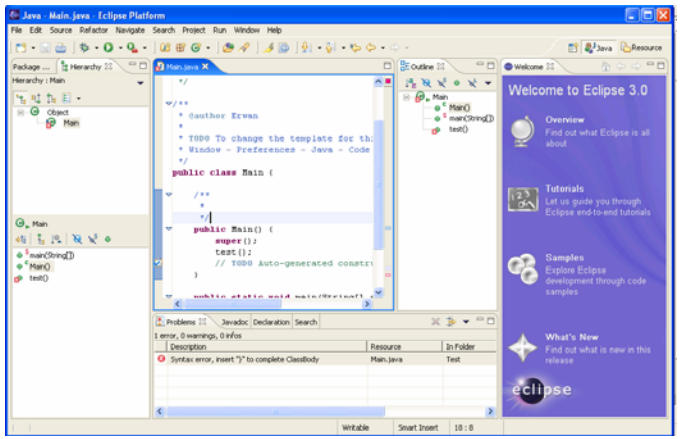
Windows L&F

GTK L&F



Mac OS L&F

Exemple SWT

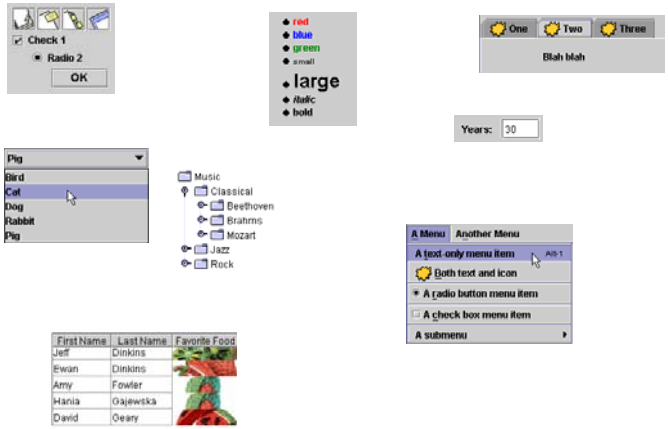


http://www.eclipse.org

Composants graphiques Swing 1/2

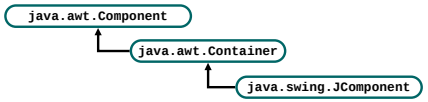
- Les composants graphiques héritent de JComponent
  - **JButton : bouton**
  - **JLabel : zone de texte ou d'icône**
  - **JTextField : champ de saisie**
  - **JTextArea : champ de saisie multiligne**
  - **JCheckBox : Case à cocher**
  - **JComboBox : Liste déroulante d'éléments**
  - **JList: zone de liste de choix (déroulant ou non)**
  - **JFrame : Conteneur de composants graphiques de plus haut niveau**
  - **JPanel : Contener graphique intermédiaire.**
  - **JMenu : Barre de menu et éléments du menu**
  - **JScrollBar : Barre de défilement**

Composants graphiques Swing 2/2

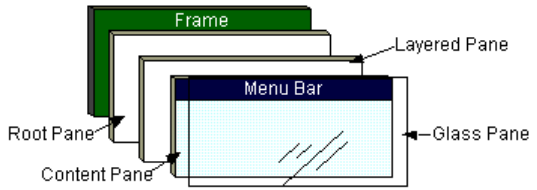


Architecture une application Swing 1/2

- **Schéma général :** Swing définit une hiérarchie des différents composants selon leur role :
- un ***top level container*** (une frame ou une applet) : i.e. un composant graphique dans lequels s'inscriront tous composants graphique.
- les ***containers*** : comme leur nom l'indique destiné a contenir des composants graphiques
- ***les composants graphiques*** (boutons, zones de saisie, liste de choix, ascenseurs, ...).



Architecture une application Swing 2/2



Fonctions communes aux Jcomponent et héritiers

- API JComponent:
  - **Apparence :**
    - **setForeground(Color), setBackground(Color)** : gestion des couleurs du composant.
    - **setFont(Font), getFont()**
    - **setCursor(Cursor) , getCursor()**
  - **Etat du composant :**
    - **setToolTipText(String) :** bulle d'aide
    - **setEnabled(boolean) :** active ou desactive le composant
    - **setVisible(boolean) :** no comment
  - **Gestion d'évenement :**
    - **Ajout d'écouteurs**
    - **contains(int, int) :** determine si un point est dans le compsant
    - **getComponentAt (int, int) :** inverse
  - **Assemblage de composants**
    - **getContentPane() :** retourne le panneau capable d'accueillir d'autres composants (JFrame)
    - **add(Component) :** ajout d'un composant à l'interieur d'un autre.
    - **getRootPane() :** retourne le panneau d'affichage principal
    - **getTopLevelAncestor() :** plus haut composant (ex: fenetre principale)
  - **Taille et position**
    - **setSize(int, int), getBounds(), …**

Interfaces graphiques préprogrammées

- Il s'agit de fournir par défaut des fenêtres prédéfinies souvent utilisées. S'utilise via des fonctions statiques parametrables
  - **JOptionPane : boîte de dialogue simple (méthodes statiques)**
    - JOptionPane.showMessageDialog (frame, « Text »)
    - JOptionPane.showMessageDialog (frame, « Text », Type_message)
    - Type_message peut valoir : JOptionPane.WARNING_MESSAGE, JOptionPane.ERROR_MESSAGE , JOptionPane.PLAIN_MESSAGE …
  - **JFileChooser : permet de sélectionner un fichier**
  - **JColorChooser : permet de sélectionner visuellement une couleur.**

Gestionnaire d'agencement : XXXLayout

- **Principe** : permet de définir des positionnement relatif afin de gérer les modifications de taille de la fenêtre ou d'un conteneur.
- **Utilisation générale** :
  - **conteneur.setLayout(unLayout)**
  - **conteneur.add(unComposant)**
  - **conteneur.add(unComposant, unePosition)**
  - **conteneur.validate()**
- **Ex** :
  - **FlowLayout :**
    - **gestionnaire par défaut.**
    - **Affiche les composants au fur et à mesure de gauche à droite.**

Gestionnaire d'agencement (suite)

- **BorderLayout :**
  - **division en 5 zones (points cardinaux).**
  - **Utilisation de constante pour préciser la zone**

Nord BorderLayout.North		
Ouest BL.West	**Centre**	**Est BL.East**
Sud BorderLayout.South		

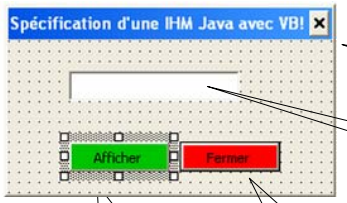
- **GridLayout : format grille**
  - **définie en nb de ligne et de colonne puis 1 composant par ligne**
  - **Ajout pour chaque ligne de gauche à droite puis descend à la ligne suivante**

Exercice sur SWING 1.

- Concevoir graphiquement une fenetre qui permet de saisir une information au clavier, de l'afficher dans une boîte de dialogue avec la possibilité d'abandonner l'opération.
- Ecrire la structure du code capable de montrer cette interface.

Solution : Spécification

- **Considérons une interface graphique composée de 2 boutons :**



Jframe qui contient les composants graphiques.

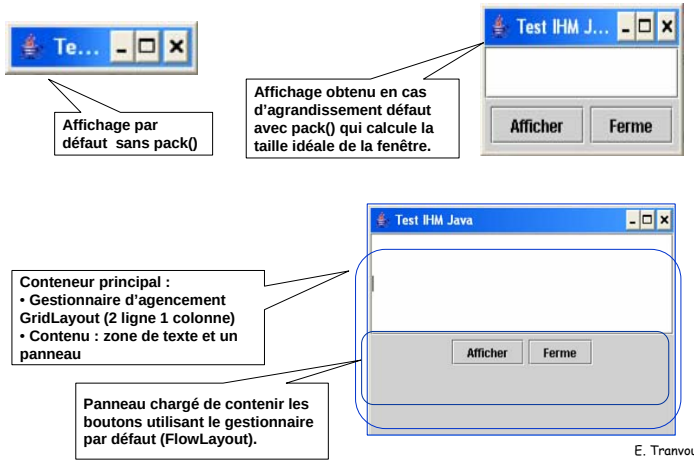
JTtextField qui permet de saisir un texte

JButton qui permet d'ouvrir une autre fenêtre avec le texte saisi.

JButton qui ferme la fenêtre

Remarque : ici ne sont pas abordés les concepts spécifiques Java comme les gestionnaires d'agencement

Solution : Résultat



Affichage par défaut sans pack()

Affichage obtenu en cas d'agrandissement défaut avec pack() qui calcule la taille idéale de la fenêtre.

Conteneur principal :
- Gestionnaire d'agencement GridLayout (2 ligne 1 colonne)
- Contenu : zone de texte et un panneau

Panneau chargé de contenir les boutons utilisant le gestionnaire par défaut (FlowLayout).

Solution : Code

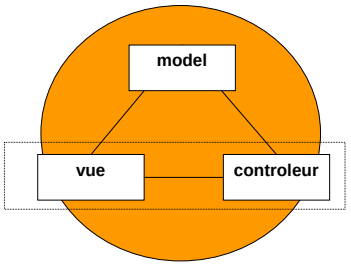
- **Voici le code correspondant *stricto senso* cad qui ne fait rien puisqu'aucun évenement n'est géré.**

```
public class Appli {
  JFrame fenetre ;
  JButton jbAffiche;
  JButton jbFerme;
  JTextField jtfSaisie;
  public Appli(String titre) {
    fenetre = new JFrame(titre);
    Container contenu =
                    fenetre.getContentPane();
    JPanel panneauBtn = new JPanel();

    jbAffiche = new JButton("Afficher");
    jbFerme = new JButton("Ferme");
    jtfSaisie = new JTextField();
    panneauBtn.add(jbAffiche);
    panneauBtn.add(jbFerme);
    contenu.setLayout(new GridLayout(2,1));
    contenu.add(jtfSaisie);
    contenu.add(panneauBtn);
    fenetre.pack();
    fenetre.setVisible(true);
  }

  public static void main(String[] args) {
    Appli appli= new  Appli("Test IHM Java");
    System.out.println("Hello");
  }
}
```

Remarque : une fois ouverte, la fenêtre s'execute dans son propre thread d'exécution. Ainsi le programme affichera « Hello » puis attendra la fermeture de la fenêtre pour se fermer.

Le Pattern Modèle - Vue - Contrôleur

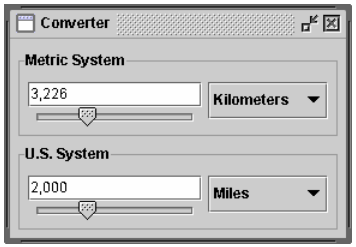
- Répartir sur plusieurs composants l'activité d'un interface.
  - **Modèle :** État et comportements devant être mis en œuvre par le composant.
  - **Vue :** Apparence extérieure du composant
  - **Contrôleur :** intermédiaire entre le modèle et la vue. Informe la vue du changement d'état du modèle (ex. inactif), et met à jour l'état du **modèle** selon les interactions d'un utilisateur avec la **vue**.



Un composant graphique actif (ex. un bouton)

Pattern MVC : exemple

- Convertisseur
  - **Modèle :** Unités de mesure, fonctions de calcul de conversion
  - **Vue :** Interface graphique
  - **Contrôleur :** liens entre composants, gestion des evenements.



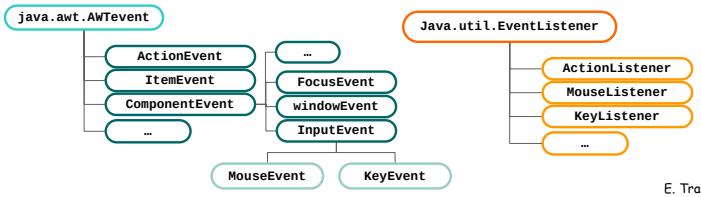
Gestion des événements : principe

- Evenement : action de l'utilisateur devant être prise en compte par l'application (mouvement souris, clic bouton, pression touche, …).

- Gérer les évenements en Java implique :
  - **définir les sources d'évenements. Cad l'interaction de l'utilisateur avec l'application java** (ex: évenement clic souris -> bouton Ok)
  - **caractériser les évenements eux-même : origine** (souris, clavier, application), **source de l'évenement, type** (cf. hiérarchie) **et contexte de l'évenement** (position souris, touche actionnée, …).
  - **définir des écouteurs d'évenements (EventListener) des composants: chargé de traiter ces évenements**

Gestion des événements : cycle

1. Utilisateur agit sur l'application ce qui génère un ou plusieurs évenements
2. Chaque évenement est identifié, caractérisé et distribué aux écouteurs d'évenements correspondant.
3. Chaque écouteur concerné effectue le traitement correspondant au type d'évenement.
4. L'action de l'évenement peut à son tour modifier l'état du composant graphique (ex. survol bouton change sa couleur)



Description d'un événement Java

Les classes xxxEvent encapsule les informations liées à un évenement :

- AWTEvent :
  - **getID : type de l'évenement**
  - **getSource : renvoie l'objet à l'orgine de l'évenement.**
- ActionEvent
  - **getActionCommand : desc. de la commande associée à l'action**
  - **getWhen : datation de l'heure de l'évenement**

- MouseEvent :
  - **getX, getY, getPoint : position du curseur dans la fenêtre**
  - **getButton : identification du bouton cliqué (BUTTON1, …)**
  - **getClickCount : nb de clic bouton**
- KeyEvent :
  - **getKeyChar : caractère associé (ou CHAR_UNDEFINED)**
  - **isActionKey : touche de contrôle**

Écouteurs

- **Interfaces spécifiant les comportements nécessaires selon le type d'évenement écouté.**
  - **xxxListener** : **yyyed()** avec yyy le type d'action ayant généré l'évenement pour lequel le listener est dédié. Toute classe implémentant l'interface xxxListener, précisera dans la méthode yyyed() le code chargé d'indiquer les conséquences de l'évenement.
  - **Exemple :**
    - **ActionListener** : actionPerformed()
    - **MouseListener**: mouseClicked(), mouseEntered (), mouseExited, mousePressed (), mouseReleased ().
    - **MouseMotionListener** : mouseDragged(), mouseMoved ().
    - **KeyListener :** keyPressed(), keyReleased(), keyTyped()
    - **TextListener** : textValueChanged()
    - **…**

Ajout d'écouteur

- **Dépends du type de composant.**
  - **addActionListener()** : JButton, JCheckBox, JComboBox, JTextField, JRadioButton
  - **addAdjustementListener()** : JScrollBar
  - **addFocusListener()** : JComponent
  - **addItemListener()** : JButton, JCheckBox, JComboBox, JRadioButton
  - **addKeyListener()** : JComponent
  - **addMouseListener()** : JComponent
  - **addMouseMotionListener()** : JComponent
  - **addWindowListener()** : JFrame et JWindows

Gérer les événements : solution 1

- **Créer une classe implémentant l'écouteur désiré.**
- **Ex :**
  - **public class MonBoutonListener implements ActionListener {**

```
public void actionPerformed(ActionEvent e) {
        ( (JButton) e.getSource() ).setText("CLIC !");
        }
}
```

Création d'un classe dédiée

Cette méthode sera appelée lorsque l'évenement ActionEvent sera crée. Elle ne fait que changer le texte du bouton à l'origine de l'évenement. Il s'agit ici d'un code générique valide pour tout objet qui s'abonne …

- **Ex d'utilisation**

```
/* .. */
mBList = new MonBoutonListener();
jbAffiche.addActionListener(mBList );
jbFerme.addActionListener(mBList );
```

Code à ajouter à la fin de l'initialisation des objets et leurs rattachement aux conteneurs. La classe MonBoutonListener peut être déclaré 'package' (ie sans public) si son utilisation est limitée à cette classe. (fin constructeur p. 35)

Gérer les événements : solution 1 - bis

- **L'«ihm» implémente l'écouteur désiré.**
- **Le principe est le même : c'est la classe à l'origine de la création du/es composant/s graphique/s qui gère également les événements**
  - Avantage/Inconvénient : tout est géré au même endroit. Donc plus simple car centralisé mais pas de séparation contrôle / vue …
- **Ex (code précédent):**
  - **public class Appli implements ActionListener { /* .. */**

```
public Appli(String titre) {
…
   jbAffiche.addActionListener(this);
   jbFerme.addActionListener(this);
}
public void actionPerformed(ActionEvent e) {
//… code gérant tous les évenements de types actions
// pour tous les composants de la classe Appli.
}
```

Gérer les événements : solution 2

- **Utilisation de classes anonymes**
  - Consiste à instancier et définir une classe en même en temps pour réduire le code à écrire.
  - Spécifiquement employé avec la gestion d'évenement
  - Ne permet d'instancier qu'un seul objet.
- **Utilisée pour faciliter l'ajout de gestionnaire d'évenement simple avec la notion de classe anonyme :**
- **Ex :**
  - **jbFerme.addActionListener(**

```
new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
                fenetre.dispose();  }
        }  )
```

On instancie ici une interface !

Remarque : la classe anonyme peut accéder aux attributs de la classe. Dispose() libere les ressources de la fenêtre cad qu'elle la tue.

Tout se passe comme si on créait une classe (sans lui donner de nom d'ailleurs) avec l'implémentation de la méthode actionP..ed() issue de l'interface et qu'en même temps on en créait une instance

Gérer les événements : solution 3

- **Utilisation d'adapteurs**
  - Consiste en des classes qui implémentent des interfaces de type listener mais ne contiennent que des méthodes vides.
- **Utilisée pour faciliter l'ajout de gestionnaire d'évenement simple de manière simillaire à la notion de classe anonyme :**
  - On instancie une classe en redéfinissant que ce qui nous intéresse (ex: simulation souris a 1 bouton).
  - On évite de redéfinir toutes les méthodes dont on n'a pas besoin.
  - Ne permet d'instancier qu'un seul objet.
- **Ex :**
  - **jbFerme.addMouseListener(**

```
new MouseAdapter() {
        public void mouseClicked(MouseEvent me)
        {       fenetre.dispose();      }
}
)
```

Nom des classes adapteurs : xxxListener -> xxxAdapter

Si on n'est interessé que par ce seul type d'évenement, cela évite de devoir redéfinir les 4 autres méthodes de l'interface MouseListener (mouseEntered(), MouseExited(), mousePressed() et mouseReleased())

Gérer les événements : solution 4

- **Utilisation de classe interne**
  - Consiste à définir une classe dans une classe. Voir apparté transparent suivant.

Aparté : principe des classes internes

- **Principe :**
  - définir une classe à l'intérieur d'une autre => concept d'encapsulation poussé à l'extrême.
- **Propriétés :**
  - A la différence d'une classe 'normale', elle peut être protected ou private (visible alors seulement depuis la classe qui la contient).
  - 2 fichiers sont alors générés par le compilateur : Externe.class et Externe$Interne.class.
  - Elle peut également accéder directement (ie par leur nom) aux attributs de la classe qui la contient voire à son instance (MaClasse.this).
  - pas **de code statique dans une classe interne *non-statique*.**
  - **Création d'une instance d'une classe interne : MaClass mc = new MaClass … suivie en suite de MaClass.MCInterne mi = mc.new MCInterne() …**
  - Peut être statique ! L'idée est la même que pour les méthodes, si le code de la classe est indépendant d'une instance on peut la faire « remonter » au niveau de la classe. => perd l'accès a l'instance de la classe. *Par contre rien n'empêche de créer plusieurs instances de la classe statique ! (sic)*

Gérer les évenements : solution 4 (reprise)

- **Utilisation de classe interne**
  - Consiste à définir une classe dans une classe chargée de centraliser la gestion des événements.
  - On reprend l'idée de la solution 1 (1 classe gère les événements) mais on la factorise en y traitant tous les évenements pouvant être traitée dans la classe « externe » ...
  - … notamment quand la classe nécessite plusieurs écouteurs (ex. MouseListener et KeyListener) ou lorsque elle-même contient plusieurs composants nécessitant le même listener (ex : 2 boutons => même comportement mais actions différentes)
  - On profite des propriétés des classes internes qui permettent d'accéder au attributs.
  - … mais on s'éloigne de la simplicité du code objet tel que préconisé au début du cours …

Gérer les événements : solution 4 (exemple)

- **Exemple de classe interne correspondant a la description p. 34.**
  - **A ajouter dans la classe Appli**

```
private class MonListener implements ActionListener {
   public void actionPerformed(ActionEvent ae) {
    if ( ae.getSource() == jbFerme)
            fenetre.dispose();
    else
      if (ae.getSource() == jbAffiche)
            JOptionPane.showMessageDialog(null,"Texte Saisis :" + jtfSaisie.getText() );
        else if (ae.getSource() == jtfSaisie)
            JOptionPane.showMessageDialog(null,"Texte Saisis 2:" + jtfSaisie.getText() );
   }
```

- **Code à ajouter dans le constructeur.**

```
MonListener m = new MonListener();          // s'utilise comme une classe normale
jbAffiche.addActionListener(m);             // la même instance gère tous les évenements
jbFerme.addActionListener(m);
jtfSaisie.addActionListener(m);
```


Exercice sur SWING 2

- Gérer les événements de la classe construite précédemment.