Programmation événementielle et réseau avec le langage Java

Module I6

IUT d'Aix-en-Provence
Réseaux et Télécommunications
Février 2011
Ivan Madjarov

Interface graphique
Gestion des Événements
Système d'Entrée-Sortie, Flux et Fichiers
Programmation réseau, UDP, TCP, Client-Serveur

JAVA
(I6)

Le langage Java – Partie 5

JAVA
INTERFACE GRAPHIQUE

Les packages (rappel)

- Un package regroupe un ensemble de classes sous un même espace de nommage.
  - L'intérêt est de regrouper les classes par : thème, lien logique, dépendance…
  - Le package peut être représenté comme un répertoire contenant des fichiers classes
  - Permet de limiter la portée du nom des classes
- Un package peut contenir des sous-packages
  - Les noms des packages suivent le schéma : **`name.subname.subsubname ...`**
- Les API de java sont organisées en packages (ex: java.lang, java.io, java.net, javax.swing …)

Les packages (rappel)

- les noms complets des classes sont :
  nom_du_package.nom_de_la_classe
  ou encore
  package.souspackage.classe
- pour utiliser des classes sans les préfixer du nom de leur package il faut :
  - ***import package.souspackage.classe;***
  - ***import package.*; (TOUTES les classes)***
- Exemple :
  - ***import java.io.File;*** `// UNE seule classe`
  - ***import java.io.*;*** `// toutes les classes`
- le '*' n 'est pas récursif !
- Implicite dans un prog.java : ***import java.lang.*;***

Les *core* API (rappel)

- Les API les plus courants :
  - **java.lang** : Types de bases, Threads, Exception, Math, ...
  - **java.util** : Hashtable, Vector, Stack, Date, ...
  - **java.applet** : Interface vers les applications Web
  - **java.awt** : Interface graphique portable
  - **java.io** : accès aux i/o par flux (fichiers, stdin, stdout,..)
  - **java.net** : Socket (UDP, TCP, multicast), URL, ...
  - **java.lang.reflect** : introspection sur les classes et les objets
  - **java.beans** : composants logiciels
  - **java.sql** (JDBC) : accès homogène aux bases de données
  - **java.security** : signature, cryptographie, authentification
  - **java.rmi** : *Remote Method Invocation*
- Les API sont installées en version binaire
  - utilisables via la **javadoc** associée (au format HTML)
  - A télécharger en même temps que le JDK!

Interface graphique

- Deux packages permettent de gérer les interfaces graphiques : AWT et SWING.
- AWT utilise des composants "lourds", c'est à dire des ressources du système d'exploitation.
- Swing utilise des composants "légers" qui ne se réfèrent pas aux ressources système.
  - Le Swing est plus robuste que l'AWT,
  - Le Swing est plus portable, et plus facile à utiliser.
  - Le Swing ne remplace pas complètement AWT
  - Le Swing fournit des composants d'interface plus performants.

Interface graphique

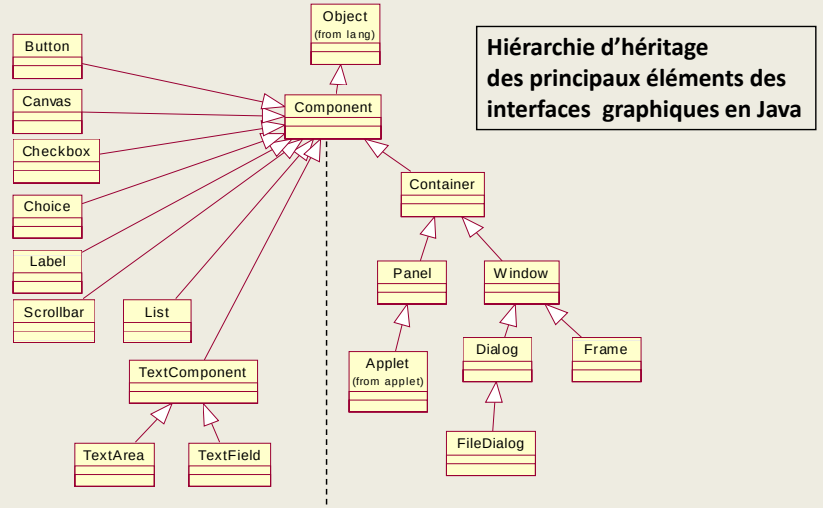
- Il y a deux principaux types de composants utilisés dans une interface graphique :
  - les *conteneurs* qui sont destinés à contenir d'autres composants (ex: *fenêtres, cadres*);
  - les *composants atomiques* qui sont des composants qui ne peuvent pas en contenir d'autres (ex: *boutons*).
- Les classes graphiques en Java à importer sont:
  - *java.awt .*;* // le package de l'AWT
  - *javax.swing.*;* // le package du Swing

Conteneurs et composants

- Les *composants* d'une interface graphique sont placés dans des *conteneurs*.
  - Un conteneur : sous-classe de la classe java.awt.Container
- Un composant est une partie "visible" de l'interface utilisateur Java (les boutons, les zones de textes).
  - une sous-classes de la classe abstraite java.awt.Component
- Chaque objet *Conteneur* utilise un gestionnaire de placement (*layout*) pour contrôler la taille de l'écran et la disposition des objets qu'il contient.
- Un *gestionnaire de placement* fait la mise en écran des composants utilisateurs.

Conteneurs et composants

- Le schéma de dépendance



Interface graphique AWT

```
Object
  |
Component
  |
  + --Button, Label, Choice, List, Checkbox, CheckboxGroup, Scrollbar
  |
  + --Canvas
  |
  + --TextComponent
  |       |
  |       + --TextField
  |       |
  |       + --TextArea
  |
  + --Container
        |
        + --Panel
        |     |
        |     + -- Applet
        |
        + --Window
              |
              + -- Frame
              |
              + -- Dialog
```


Conteneurs et composants

- Un *Frame* représente une fenêtre de haut niveau avec un titre, une bordure et des angles de redimensionnement.

```
import java.awt.*;

public class EssaiFenetre1
{
  public static void main(String[] args)
  {
    Frame f =new Frame("Ma première fenêtre");
    Button b= new Button("coucou");
    f.add(b);
    f.pack();
    f.show();
  }
}
```

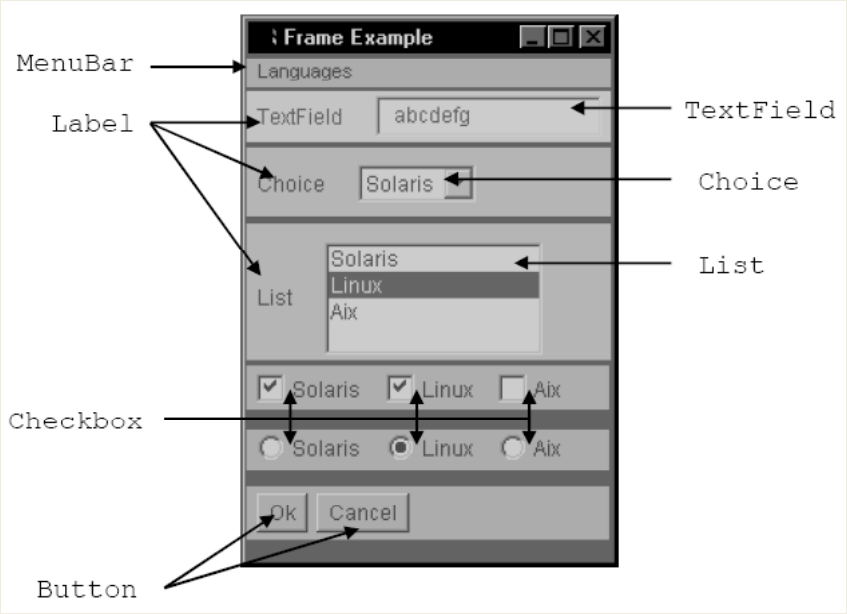
Création d'une fenêtre (un objet de la classe Frame) avec un titre

Création du bouton ayant pour label "coucou"

Ajout du bouton dans la fenêtre

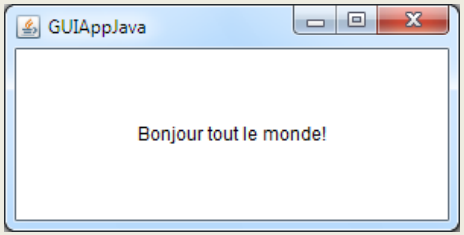
On demande à la fenêtre de choisir la taille minimum avec pack() et de se rendre visible avec show()

Interface graphique AWT



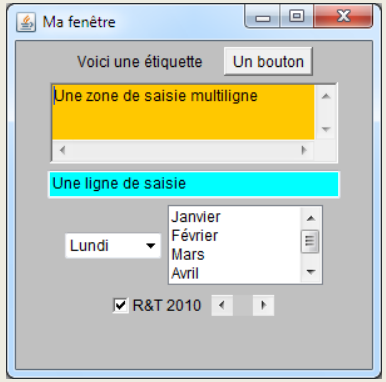
Application graphique AWT

```
import java.awt.*;  // importer le package graphique de base
class GUIApp {      // nom de la classe graphique
  public static void main(String argv[]) {    // la méthode main
    // instanciation de l'objet cadre en tant que conteneur
    Frame fn = new Frame("GUIAppJava");
    // instanciation d'un objet de type étiquette avec alignement
    Label bnj = new Label("Bonjour tout le monde!",Label.CENTER);
    // ajouter le cadre au conteneur avec alignement vertical
    fn.add("Center",bnj);
    // délimiter le champ visuel
    fn.resize(300,150);
    // faire visible à l'écran
    fn.show();    } }
```



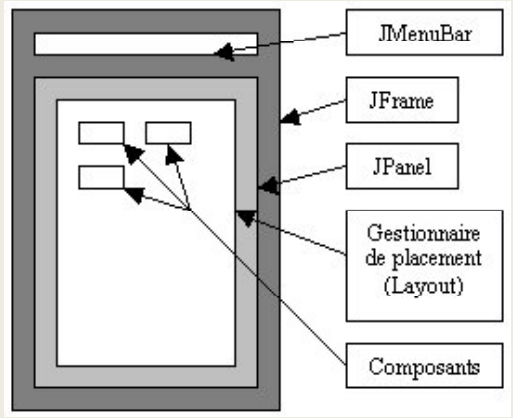

```
import java.awt.*;
public class Fenetre1 extends Frame {
  Label la;     // étiquette
  TextField tf; // champ de saisie
  TextArea ta ; // aire de saisie
  Button b ;    // bouton
  Checkbox cb;  // Case à cocher
  Choice ch;    // liste déroulante
  List li;      // Liste simple
  Scrollbar sc; // défilement
  public Fenetre1(String titre) { // Constructeur
    super(titre); // appel à la super-classe Frame
    this.setBackground(Color.lightGray); // couleur de fond
    la = new Label("Voici une étiquette");
    b = new Button("Un bouton");
    tf = new TextField("Une ligne de saisie", 30);
    tf.setBackground(Color.CYAN);
    ta = new TextArea ("Une zone de saisie multiligne", 3 ,30);
    ta.setBackground(Color.ORANGE);
    ch = new Choice();   // liste déroulante
    ch.add ("Lundi");    // ajout de composants
    ch.add ("Mardi");  ch.add ("Mercredi");
    ch.add ("Jeudi");  ch.add ("Vendredi");
    li = new List();    // Liste simple
    li.add ("Janvier"); // ajout de composants
    li.add ("Février");    li.add ("Mars");
    li.add ("Avril");      li.add ("Mai");
    cb = new Checkbox("R&T 2010", true); // case à cocher
    sc = new Scrollbar(Scrollbar.HORIZONTAL, 100 , 30 , 0, 1000);
    FlowLayout fl = new FlowLayout (FlowLayout.CENTER, 5, 5);
    this.setLayout(fl);
    add(la);     add(b);     add(ta);     add(tf);
    add(ch);     add(li);    add(cb);     add(sc);
  }
```

```
  public static void main (String args[]) {
      Fenetre1 f = new Fenetre1("Ma fenêtre");
      f.setBounds(200, 100, 300, 300);
      f.setVisible(true);
      f.ta.requestFocus();
    }
  }
```



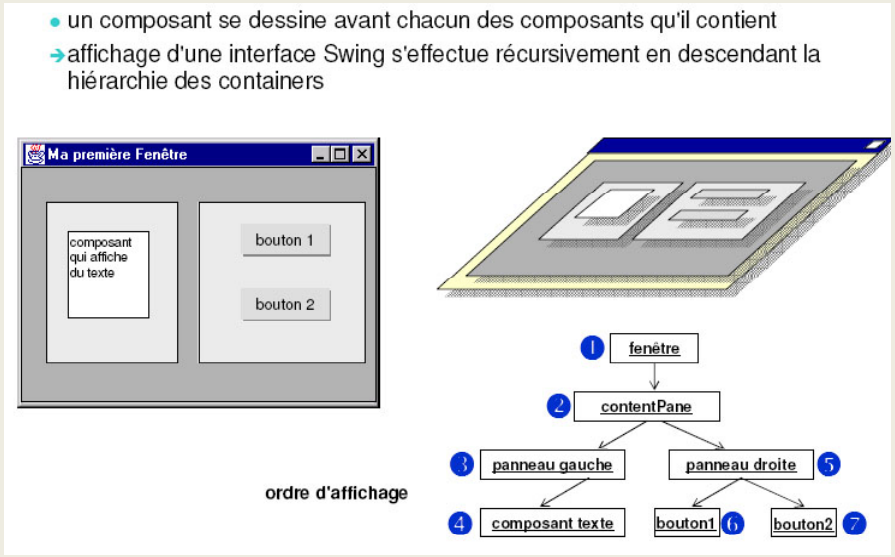
Interface graphique Swing

- Une application graphique (voir une fenêtre) peut être construite selon le principe suivant :



Interface graphique Swing

- L'ordre d'affichage des composants :



Fenêtre graphique Swing

- JWindow : C'est la fenêtre la plus basique. C'est juste un conteneur sans barre de titre, boutons de fermeture et n'est pas redimensionnable.
- JDialog : C'est une fenêtre destinée aux boîtes de dialogue. Ce type de fenêtre est modal, c.à.d. qu'elle bloque une autre fenêtre tant qu'elle est ouverte.
- JFrame : Forme la fenêtre principale d'une application. Ne dépendant d'aucune autre fenêtre et ne peut pas être modale. Elle a une barre de titre, bouton de fermeture et de redimensionnement et peut accueillir une barre de menu et peut être iconifié.

La hiérarchie des composants

- *Button* : classe des boutons à cliquer.
- *Canvas* : objet zone de dessin.
- *Checkbox* : boîte à cocher.
- *CheckboxGroup* : objet qui regroupe plusieurs objets *Checkbox* réalise un ensemble de boutons radio.
- *Choice* : objet qui permet de gérer une liste de choix.
- *Label* : affiche simplement une ligne de texte fixe (étiquette).
- *List* : objet spécialisé dans l'affichage d'une liste d'items.
- *Scrollbar* : objet pour choisir une valeur dans un intervalle.
- *TextComponent* :
  - *TextField* : une simple ligne d'édition (champ de saisie).
  - *TextArea* : zone d'édition de texte de plusieurs lignes, munie de barres de défilement.

La hiérarchie des composants

- *Container* : classe abstraite (sans objet) qui sert de support à d'autres composants. Principales méthodes : *add(), remove()*
  - *Panel* : c'est un objet conteneur qui contient lui-même d'autres conteneurs, placé souvent dans un objet *Frame*.
  - *Applet:* application interprétée par un navigateur Web.
- *Window* : fenêtre d'application sans bordure, ni barre de menus; c'est un *conteneur*, géré par défaut par un *BorderLayout*. Rarement instanciée directement, on utilise ses sous-classes *Dialog* et *Frame*.
  Méthodes : *setVisible(), dispose(), pack().*
  - *Dialog* : fenêtre de dialogue, qui dépend d'une fenêtre *Frame*, aussi conteneur
  - *Frame* : fenêtre encadrée affichable, redimensionnable : c'est la fenêtre typique d'une application graphique Java.

Interface graphique - Fenêtre

- Une fenêtre (conteneur) de type ***JFrame***

```
import javax.swing.JFrame;
public class TestFen {
   public static void main(String[] args){
     // Instancier l'objet fenêtre
     JFrame fn = new JFrame();
     // Définir un titre pour votre fenêtre
     fn.setTitle("Ma première fenêtre java");
     // Définir sa taille: 500 px de large et
     // 400 px de haut
     fn.setSize(500, 400);
     // Alignement centré
     fn.setLocationRelativeTo(null);
     //Terminer le processus au clique "Fermer"
     fn.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     // Rendre visible la fenêtre à l'écran
     fn.setVisible(true);
   }
}
```


Interface graphique - Fenêtre

- Une fenêtre (conteneur) héritée de type ***JFrame***

```
import javax.swing.Jframe;
public class Fenetre extends Jframe {
  public Fenetre() {
  // Définir un titre pour la fenêtre
  this.setTitle("Ma première fenêtre java");
  // Définir sa taille: 400 px de large et 500 px de haut
  this.setSize(400, 500);
  // positionner au centre
  this.setLocationRelativeTo(null);
  // Fermer lorsqu'on clique sur "Fermer" !
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  // Rendre visible
  this.setVisible(true);
  }
}
```


Interface graphique - Fenêtre

- Méthodes liées au ***JFrame*** *:*
  - Positionner la fenêtre à l'écran :
    <refObjet>.setLocation(int x, int y);
  - Empêcher le redimensionnement de la fenêtre :
    <refObjet>. setResizable(false);
  - Faire que la fenêtre soit toujours au premier plan :
    <refObjet>. setAlwaysOnTop(boolean b);
  - Retirer les contours et les boutons de contrôles :
    <refObjet>. setUndecorated(Boolean b);
  - Référencer le conteneur :
    <refObjet>. getContentPane();

Interface graphique - bouton

```
import java.awt.* ;
import javax.swing.* ;
class MaFenetre extends JFrame {
   private JButton MonBouton ;    // bouton
   public MaFenetre () {          // Constructeur
      super() ; // Appel au constructeur de la classe JFrame
      setTitle("Ma première fenêtre avec un bouton") ;
      // initialisation du titre de la fenêtre
      setBounds(10, 40, 300, 200) ;
   // le coin supérieur gauche de la fenêtre est placé au pixel de
   coordonnées x=10, y=40 et ses dimensions seront de 300 X
   200 pixels.
```

Interface graphique - bouton

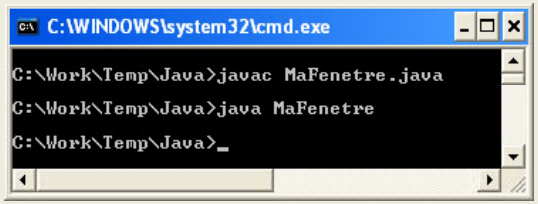
```
    MonBouton = new JButton("Un bouton") ;
    /* création d'un bouton de référence MonBouton portant
       l'étiquette "Un bouton" */
    getContentPane().add(MonBouton) ;
    /* Pour ajouter un bouton à une fenêtre, il faut incorporer
    le bouton dans le contenu de la fenêtre.
    La méthode getContentPane de la classe JFrame fournit une
     référence à ce contenu, de type Container.
    La méthode add de la classe Container permet
    d'ajouter le bouton au contenu de la fenêtre. */
    }
}
```


Interface graphique - bouton

```
public class MonProgGraphique {
    public static void main(String args[]) {
        JFrame  fen = new MaFenetre() ;
        fen.setVisible(true) ;
        // rend visible la fenêtre de référence fen
    }
}
```

Interface graphique - bouton

- Un bouton est visible par défaut.
- Une fenêtre de type *JFrame* peut être retaillée, déplacée et réduite à une icône.
- La fermeture de la fenêtre ne met pas fin au programme, mais rend simplement la fenêtre invisible.
- Pour mettre fin au programme, il faut fermer la fenêtre console ou interrompre le programme par Ctrl/c!



Conteneurs Swing : JFrame

- Montrer et cacher un composant
  void setVisible(boolean b)
- Activer et désactiver un composant
  void setEnabled(boolean b)
- Modifier la couleur de fond d'un composant
  void setBackground(Color c)
- Modifier la couleur du premier plan d'un composant
  void setForeground(Color c)
- Modifier la taille d'un composant
  void setSize(int largeur, int hauteur)
- Modifier la position et la taille d'un composant
  void setBounds(int x, int y, int largeur, int hauteur)

Les composants atomiques

- La classe *JComponent* est une classe abstraite dérivée de la classe Container
- La classe *JComponent* encapsule les composants atomiques d'une interface graphique, tels que:
  - boutons,
  - les cases à cocher,
  - les boutons radio,
  - les étiquettes,
  - les champs de texte,
  - les boîtes de liste.

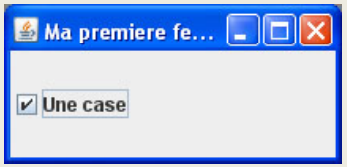
Les composants atomiques

- La méthode *setPreferredSize* de la classe *Jcomponent* permet d'imposer une taille à un composant.
- Elle prend en argument un objet de type *Dimension*.
- Exemple:

```
JButton bouton = new JButton("Un bouton") ;
bouton.setPreferredSize(new Dimension(10, 20)) ;
// bouton de largeur 10 et de hauteur 20 pixels
```


Les cases à cocher

- La case à cocher de type JCheckBox permet à l'utilisateur d'effectuer un choix de type oui/non. Exemple:




```
class MaFenetre extends JFrame  {
    private JCheckBox MaCase;
    public MaFenetre ()  {
        super("Ma première fenêtre") ;
        setBounds(10, 40, 300, 200) ;
        MaCase = new JCheckBox("Une case") ;
        // la référence MaCase portant l'étiquette Une case
        getContentPane().add(MaCase) ;
    }  }
```


Les cases à cocher

- Par défaut, une case à cocher est construite dans l'état non coché (false).
- La méthode *isSelected* de la classe *AbstractButton* permet de connaître l'état (true ou false).
- La méthode *setSelected* de la classe *AbstractButton* permet de modifier l'état d'une case à cocher.
- Exemple:

```
MaCase.setSelected(true) ;
// coche la case de référence MaCase
```

Les boutons radio


```
class MaFenetre extends JFrame {
    private JRadioButton bRouge;        private JRadioButton bVert;
    public MaFenetre() {
      super("Une fenetre avec des boutons radio") ;      setBounds(10,40,300,200) ;
      bRouge = new JRadioButton("Rouge") ;  // bouton radio de référence bRouge
      bVert = new JRadioButton("Vert") ;  // bouton radio de référence bVert
      ButtonGroup groupe = new ButtonGroup() ;
      groupe.add(bRouge) ;      groupe.add(bVert) ;
      /* Un objet de type ButtonGroup sert à désactiver un bouton dans le groupe
    activé.  Sinon le bouton radio se comporte comme une case à cocher */
      Container contenu = getContentPane() ;  contenu.setLayout(new FlowLayout()) ;
      /* Un objet de type FlowLayout est un gestionnaire de mise en forme qui dispose
    les composants les uns à la  suite des autres */
      contenu.add(bRouge) ;       contenu.add(bVert) ;
     /* Ajout de chaque bouton radio dans la fenêtre. Un objet de type ButtonGroup
    n'est pas un composant et ne peut pas être ajouté à un conteneur */
    }
}
```


Les étiquettes

- Une étiquette de type *JLabel* permet d'afficher dans un conteneur un texte (d'une seule ligne) non modifiable, mais que le programme peut faire évoluer.


```
...
private JLabel MonTexte;
...
MonTexte = new JLabel ("texte initial") ;
//Une étiquette de référence MonTexte avec  "texte initial"
getContentPane().add(MonTexte) ;
MonTexte.setText("nouveau texte") ;
//modification du texte de l'étiquette de référence MonTexte
```


Les champs de texte

- Un champ de texte de type JTextField est une zone rectangulaire dans laquelle l'utilisateur peut entrer ou modifier un texte (d'une seule ligne).


```
...
private JTextField MonChamp1 ;
...
MonChamp1 = new JTextField("texte initial", 20) ;
//champ de taille 20 caractères avec un texte initial
getContentPane().add(MonChamp1) ;
```

- La méthode getText permet de connaître à tout moment le contenu d'un champ de texte. Exemple

```
String ch= Monchamp1.getText() ;
```


Les boîtes de liste

- Le type JList permet de choisir une ou plusieurs valeurs dans une liste prédéfinie. Initialement, aucune valeur n'est sélectionnée dans la liste.
- Exemple:

```
String[] couleurs =
   {"rouge", "bleu", "gris", "vert", "jaune", "noir"};
JList MaListe = new Jlist(couleurs) ;
MaListe.setSelectedIndex(2) ;
//sélection préalable de l'élément de rang 2
```

Les boîtes de liste combinée

- Le type JComboBox associe un champs de texte et une boîte de liste à sélection simple. Tant que le composant n'est pas sélectionné, seul le champ de texte s'affiche :

- Lorsque l'utilisateur sélectionne le champ de texte, la boîte de liste s'affiche :

- L'utilisateur peut choisir une valeur dans la boîte de liste qui s'affiche alors dans le champ de texte.

Les boîtes de liste combinée

```
String[] couleurs =
   {"rouge", "bleu", "gris", "vert", "jaune", "noir"};
JComboBox MaCombo = new JComboBox(couleurs) ;
MaCombo.setSelectedIndex(2) ;
//sélection préalable de l'élément de rang 2
```

- La boîte combo est dotée d'une barre de défilement dès que son nombre de valeurs est supérieur à 8. On peut modifier le nombre de valeurs visibles à 4:

```
MaCombo.setMaximumRowCount(4) ;
```

- Pour récupérer l'élément choisi on utilise la méthode:

```
String ch = (String) MaCombo. getSelectedItem() ;
```


Les dessins avec Java

- Pour dessiner, on utilise, en général, un conteneur de type *JPanel* qui est un composant intermédiaire (*panneau*).
  - Il peut être contenu dans un conteneur et peut contenir d'autres composants.
- Un panneau est une sorte de "sous-fenêtre", sans titre ni bordure, qui doit être associé par la méthode *add* (de la classe *Container*) à un autre conteneur, généralement une fenêtre.
  - *getContentPane().add(panneau) ;*
- Le gestionnaire de mise en forme *FlowLayout* est le gestionnaire par défaut des panneaux.

Dessin dans un panneau

- Pour obtenir un dessin permanent dans un composant on appel la méthode :

void paintComponent (Graphics g)

- L'argument *g* de type *Graphics* est le contexte graphique du composant qui a appelé la méthode. Il sert d'intermédiaire entre les demandes de dessin et leur réalisation effective sur le composant.

Dessin dans un panneau

- Les couleurs sont représentés par un objet de type *Color*
- Les constantes prédéfinies de la classe Color sont :
  - Color.black, Color.white, Color.blue, Color.green, Color.red, Color.yellow.
- On peut récupérer ou modifier la couleur de d'avant plan ou de l'arrière plan par:
  - *setForeground* ou *setBackground*
  - *getColor()* et *setColor(Color c)*
- Les méthodes getColor et setColor sont des méthodes de la classe Graphics.

La police du contexte graphique

- Une police est définie par le type *Font*
  - *Font f = new Font("Serif", Font.ITALIC, 18);*
  - un nom de famille de police (*SansSerif, Serif, Monospaced*)
  - un style : style normal (*Font.PLAIN*), style gras (*Font.BOLD*), style italique (*Font.ITALIC*)
- On peut récupérer la police du contexte graphique:
  - *public abstract Font getFont()*
- modifier la police du contexte graphique
  - *public abstract setFont(Font f)*
- Les méthodes *getFont* et *setFont* sont des méthodes de la classe *Graphics*.

Méthodes de dessin (Graphics)

public abstract void drawLine(int x1, int y1, int x2, int y2)

Trace un trait du point de coordonnées (x1, y1) au point decoordonnées (x2, y2)

public void drawRect(int x, int y, int largeur, int hauteur)

Dessine un rectangle de largeur largeur et de hauteur hauteur au point de coordonnées (x, y)

public void drawOval(int x, int y, int largeur, int hauteur)

Dessine un ovale de largeur largeur et de hauteur hauteur au point de coordonnées (x, y)

public abstract void drawstring(String str, int x, int y)

Dessine le texte str au point de coordonnées (x, y)

Exemple

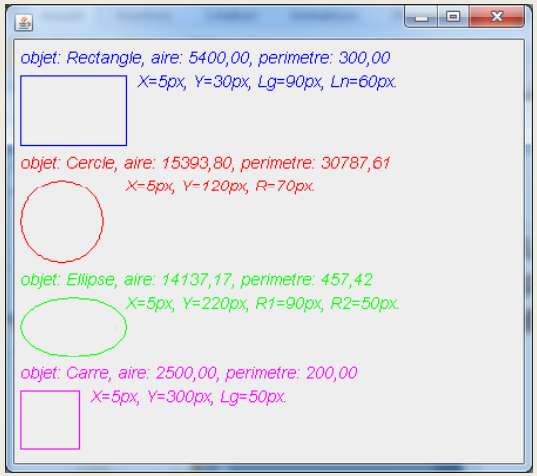
```
import java.awt.*;
import javax.swing.*;
class MaFenetre extends JFrame {
    private JPanel panneau;
    public MaFenetre () {
        super("Une fenetre avec un panneau jaune");
        setSize(200, 150);  panneau = new Panneau();
        panneau.setBackground(Color.yellow);
        getContentPane().add(panneau);
    }
    public static void main(String args[]) {
        JFrame fen = new MaFenetre11() ;
        fen.setVisible(true) ;
    } }
```

Exemple

```
class Panneau extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g) ;
        Font f = new Font("Serif", Font.ITALIC, 18);
        g.setColor(Color.black);  g.setFont(f);
        g.drawString("Rectangle", 50, 20);
        g.setColor(Color.red);
        g.drawRect(50, 30, 80, 50);
    }
}
```

Exemple

- La classe des figures géométriques développée déjà peut servir de base pour un exercice de dessin graphique en Java. Voici le résultat souhaité :



La gestion de la mise en forme

- Pour chaque conteneur (fenêtre, boîte de dialogue, …), Java permet de choisir un gestionnaire de mise en forme ("*Layout manager*") responsable de la disposition des composants. (*en colonnes, les uns à la suite des autres, en cercle*, ...).
  - *BorderLayout, FlowLayout, CardLayout, GridLayout, BoxLayout, GridBagLayout*.
- La méthode *setLayout* de la classe *Container* permet d'associer un gestionnaire de mise en forme à un conteneur.
- Le gestionnaire *BorderLayout* est le gestionnaire par défaut des fenêtres et des boîtes de dialogue.

La gestion de la mise en forme

- Le gestionnaire de type *FlowLayout* organise les objets graphiques dans la fenêtre de gauche à droite et de haut en bas suivant
  - Les objets sont insérés dans le conteneur à l'aide de la méthode *add*.
- Les gestionnaire de type *GridLayout* organise les objets graphiques dans la fenêtre suivant un quadrillage en lignes et colonnes dont les dimensions sont spécifiées par l'utilisateur au moment de la création du gestionnaire.
  - Les objets sont insérés séquentiellement à partir de la case en haut à gauche de la fenêtre, de gauche à droite et de haut en bas, suivant l'ordre des appels à la méthode *add*.

Les gestionnaires

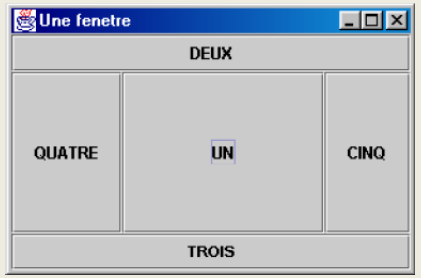
- Le gestionnaire de mise en forme *CardLayout* permet de disposer les composants suivant une pile.
  - Seul le composant supérieur est visible à un moment donné. Par défaut, c'est le premier ajouté au conteneur.
- Le gestionnaire de mise en forme *GridLayout* permet de disposer les composants les uns à la suite des autres sur une grille régulière (*tableau*).

  public GridLayout(rows, cols, hgap, vgap);
  - *rows* et *cols* définissent respectivement le nombre de lignes et de colonnes de la grille.
  - *hgap, vgap* définissent respectivement l'espacement entre les composants placés en écran.

Le gestionnaire *BorderLayout*

- Ce gestionnaire permet de placer chaque composant dans une zone géographique.

Constante symbolique	Valeur
BorderLayout.NORTH	"North"
BorderLayout.SOUTH	"South"
BorderLayout.EAST	"East"
BorderLayout.WEST	"West"
BorderLayout.CENTER	"Center"



Le gestionnaire *BorderLayout*

```
class MaFenetre extends JFrame  {
   public MaFenetre ()  {
        super("Une fenetre") ;     setSize(300, 200) ;
        Container contenu = getContentPane() ;
        contenu.setLayout(new BorderLayout()) ;
        contenu.add(new JButton("UN")) ;
        // bouton placé au centre par défaut
        contenu.add(new JButton("DEUX"), "North") ;
        contenu.add(new JButton("TROIS"), "South") ;
        contenu.add(new JButton("QUATRE"), "West") ;
        contenu.add(new JButton("CINQ"), "East") ;
   }
}
```


Le gestionnaire *BorderLayout*

- La classe dispose de deux constructeurs :

public BorderLayout() ;

public BorderLayout(int hgap, int vgap) ;

- *hgap* et *vgap* définissent l'espace horizontal et vertical (en pixels) entre les composants d'un conteneur. (5 pixels par défaut).
- *BorderLayout* ne tient pas compte de la taille souhaitée des composants.
- Leur taille peut être définie par la méthode *setPreferredSize* de la classe *JComponent*

Le gestionnaire FlowLayout

- La classe *FlowLayout* dispose de trois constructeurs :

1. *public FlowLayout() ;*
2. *public FlowLayout(int align) ;*

- *align* est un paramètre d'alignement d'une ligne de composants par rapport aux bords verticaux de la fenêtre.

  FlowLayout.LEFT("Left"),
  FlowLayout.RIGHT("Right")
  FlowLayout.CENTER("Center").

- Par défaut les composants sont alignés à gauche.

3. *public FlowLayout(int align, int hgap, int vgap) ;*

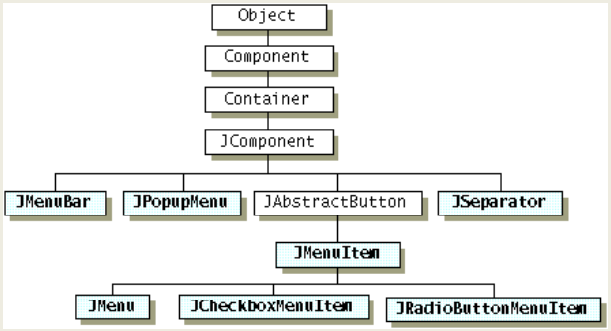
- *hgap* et *vgap* définissent les espaces entre les composants.

Le gestionnaire FlowLayout

```
class MaFenetre extends JFrame {
    public MaFenetre () {
        super("Une fenetre") ; setSize(300, 100) ;
        Container contenu = getContentPane() ;
        contenu.setLayout(new FlowLayout()) ;
        contenu.add(new JButton("UN")) ;
        contenu.add(new JButton("DEUX")) ;
    }
```

Les menus déroulants

- Les menus sont des composants d'un conteneur JFrame.
- Les menus déroulants font intervenir trois sortes d'objets:
  - un objet barre de menu de type JMenuBar ;
  - différents objets menu (Jmenu), visibles sur la barre de menu
  - pour chaque menu, les différentes options, de type JMenuItem, qui le constituent.



Les menus déroulants

```
private JMenuBar barreMenus ;
private JMenu couleur, dimensions ;
private JMenuItem rouge, vert, hauteur, largeur ;
....
// création d'une barre de menu
barreMenus = new JMenuBar() ;
setJMenuBar(barreMenus) ;
// ajouter la barre de menu dans la fenêtre
// création d'un menu Couleur et de ses options Rouge
couleur = new JMenu("Couleur") ;
barreMenus.add(couleur) ;
rouge = new JMenuItem("Rouge") ;
couleur.add(rouge) ;
couleur.addSeparator() ;
// ajouter une barre séparatrice avant l'option suivante
```

Les menus déroulants

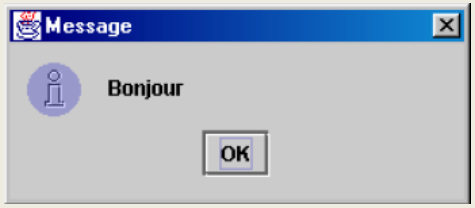
```
vert = new JMenuItem("Vert");  couleur.add(vert) ;
// création d'un menu Dimensions et de ses options "Hauteur" et "Largeur"
dimensions = new JMenu("Dimensions");
barreMenus.add(dimensions);
hauteur = new JMenuItem("Hauteur");
dimensions.add(hauteur); dimensions.addSeparator() ;
largeur = new JMenuItem("Largeur") ;
dimensions.add(largeur) ;
```

Les boîtes de dialogue

- La boîte de dialogue est un conteneur qui permet de regrouper des composants dans une fenêtre qu'on fait apparaître ou disparaître.
- Java propose un certain nombre de boîtes de dialogue standard obtenues à l'aide de méthodes de la classe *JOptionPane* :
  - boîtes de message,
  - boîtes de confirmation,
  - boîtes de saisie et boîtes d'options.
  - La classe *JDialog* permet de construire des boîtes de dialogue personnalisées.

Les boîtes de message

- Une boîte de message fournit à l'utilisateur un message qui reste affiché tant que l'utilisateur n'agit pas sur le bouton OK.
- Elle est construite à l'aide de la méthode de classe *showMessageDialog* de la classe *JOptionPane*.

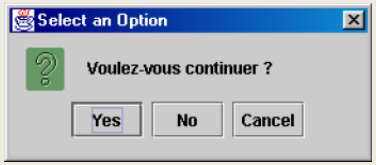


Les boîtes de message

```
import java.awt.* ;
import javax.swing.* ;
class MaFenetre extends JFrame {
  public MaFenetre () {
    super("Une fenêtre") ; setSize(400, 300) ;   }
  public static void main(String args[ ]) {
    JFrame fen = new MaFenetre() ;  fen.setVisible(true) ;
    JOptionPane.showMessageDialog(fen, "Bonjour") ;
    /* le premier argument correspond à la fenêtre parent de la
    boîte de message. Cet argument peut prendre la valeur
    null. */
    }
}
```

Les boîtes de confirmation

- La boîte offre un choix de type oui/non à l'aide de la méthode de classe showConfirmDialog de la classe JOptionPane.

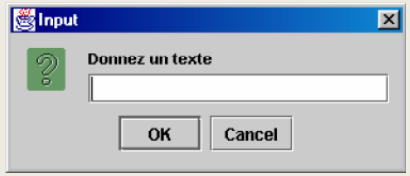



```
public class BoiteConf  {
  public static void main(String args[ ]) {
     JFrame fen = new MaFenetre() ;
     fen.setVisible(true) ;
     JOptionPane.showConfirmDialog(fen, "Voulez-vous
    continuer ?") ;
  }
}
```


Les boîtes de saisie

- La boîte permet l'acquisition d'une chaîne de caractères.
- La boîte de saisie est construite à l'aide de la méthode showInputDialog de la classe JOptionPane.

```
public class BoiteSaisie {
  public static void main(String args[ ]) {
     JFrame fen = new MaFenetre() ;
     fen.setVisible(true) ;
     JOptionPane.showInputDialog (fen, "Donnez un texte") ;
  }
}
```



Affichage d'images

- Java traite deux formats d'images :
  - le format GIF (*Graphical Interface Format*) avec 256 couleurs disponibles.
  - le format JPEG (*Joint Photographic Expert Group*) avec plus de 16 millions de couleurs disponibles.
- Trois étapes de chargement d'une image:
  - le constructeur de la classe *ImageIcon* permet de charger une image.
  - la méthode *getImage* permet d'obtenir un objet de type Image depuis un objet de type URL ou depuis un fichier local.
  - la méthode *drawImage* permet d'afficher une image de type Image.

Affichage d'images (exemple)

```
import java.awt.*;
import javax.swing.*;
class MaFenetre extends JFrame {
    private JPanel pan;
    public MaFenetre() {
        super("Une fenetre avec une image");
        setSize(300, 245);
        pan = new Panneau();
        getContentPane().add(pan);
    }
    public static void main(String args[]) {
        JFrame fen = new MaFenetre() ;
        fen.setVisible(true) ;
    }
}
```

Affichage d'images (exemple)

```
class Panneau extends JPanel  {
    private ImageIcon rouge;
    public Panneau() {
        img = new ImageIcon("IMG_1683.jpg") ;
    // chargement d'une image dans l'objet de référence img, à partir d'un fichier
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Image imG = img.getImage();
    // la méthode getImage retourne une référence à un objet de type Image  à partir d'un objet de type ImageIcon
        g.drawImage(imG, 5, 5, this);
    // affiche l'image imG au point de coordonnées (5, 5)
    // this pour appeler la méthode repaint()
    }
}
```


Affichage d'images (exemple)

Programmation événementielle et réseau avec le langage Java

Module I6

IUT d'Aix-en-Provence
Réseaux et Télécommunications
Février 2011
Ivan Madjarov

Le langage Java – Partie 6

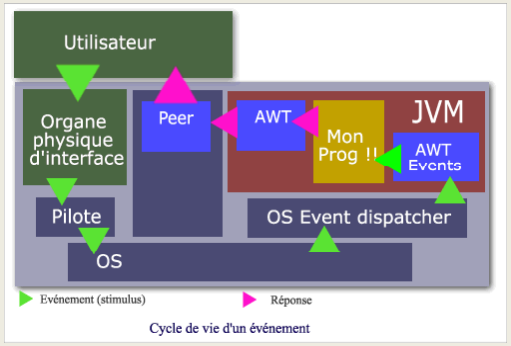
INTERFACE GRAPHIQUE ET LA GESTION DES ÉVÉNEMENTS

Le concept d'événement

- La programmation événementielle se base sur la conception du comportement d'un programme :
  - Un programme a un déroulement permanent, en interaction et au service d'un utilisateur.
  - Le programme ne "*fait rien*" tant que l'utilisateur ne lui demande "*rien*".
- Dans la programmation événementielle existe une "*boucle d'attente*", qui permet au processeur d'attribuer du temps à d'autres programmes (*fonction d'interruption du système d'exploitation*).
- Lorsqu'un utilisateur intervient, il envoie un "*événement*" au programme.
- L'événement est un "*objet*" passé à une fonction particulière du programme, chargée de son acheminement vers le logiciel qui est sensé le traiter.

Le concept d'événement

- Chaîne d'événements dans une application Java
  - L'événement est une action d'un utilisateur sur un dispositif physique d'interface. (une souris, un clavier,...)
  - L'événement est traduit en un "code d'événement matériel" par le dispositif d'analyse et de contrôle de l'interface. (un contrôleur appropriée qui reconnait son "*interruption code*")
  - Ce code d'événement matériel provoque une "interruption" du système d'exploitation.
  - Le système d'exploitation communique le code à un service de distribution d'événements.
  - Ce service est sensé savoir à quelle application est destiné cet événement.
  - L'événement "système" est fourni à la MVJ qui l'a distribue à l'objet concerné.



Cycle de vie d'un événement

La gestion des événements

- Exemples d'événements:
  - Un clic souris,
  - La frappe d'une touche au clavier,
  - Le changement de la taille d'une fenêtre.
- En Java, les événements n'ont pas une valeur physique, mais logique.
- Un événement dépend du composant qui l'a généré.
- On appelle source d'un événement l'objet qui l'a généré.
  - *MouseEvent* : un clic souris dans une fenêtre
  - *ActionEvent* : un clic souris sur un bouton

Traiter un événement

- En général tout événement qui peut se produire dans une interface graphique est de type *XXXEvent*
- Le traitement des événements (*Event*) est délégué à des objets particuliers appelés *écouteurs* (*Listener*)
- En fonction des événements qu'ils traitent, un écouteur doit implémenter une interface particulière, dérivée de l'interface *EventListener*, qui correspond à une catégorie d'événements.
  - Pour traiter un événement de type *XXXEvent*, un écouteur doit implémenter l'interface *XXXListener*.

Traiter un événement

- Interfaces à implémenter pour le traitement des événements

interface	type d'événement	méthodes
`ActionListener`	évènement correspondant à une action telle que: enfoncer un bouton, sélection d'un item dans un menu	`void actionPerformed(ActionEvent e)`

`ComponentListener`	composant rendu invisible,
	composant déplacé,
	changement de taille du composant,
	composant rendu visible
		`void componentHidden(ComponentEvent e)`
		`void componentMoved(ComponentEvent e)`
		`void componentResized(ComponentEvent e)`
		`void componentShown(ComponentEvent e)`

`FocusListener`	focus clavier du composant, père du focus clavier,	`void focusGained(FocusEvent e)` `void focusLost(FocusEvent e)`

`MouseListener`	click souris dans un composant,
	entrée dans un composant,
	sortie d'un composant,
	bouton pressé dans un composant,
	bouton relâché dans un composant
		`void mouseClicked(MouseEvent e)`
		`void mouseEntered(MouseEvent e)`
		`void mouseEntered(MouseEvent e)`
		`void mousePressed(MouseEvent e)`
		`void mouseReleased(MouseEvent e)`

`WindowListener`	évènement sur une fenêtre	`void windowActivated(WindowEvent e)`
		`void windowClosed(WindowEvent e)`
		`void windowDeiconified(WindowEvent e)`
		`void windowIconified(WindowEvent e)`
		`void windowOpened(WindowEvent e)`

Traiter un événement

- En général l'écouteur d'événement d'un composant d'interface, est souvent le conteneur de ce composant.
- Un objet écouteur (JPanel) contient un à plusieurs objets source d'événements (par exemple des JButton)
- L'écouteur s'enregistre auprès des sources d'événements afin de pouvoir les écouter
  - JPanel doit implémenter l'interface **ActionListener** possédant une seule méthode **actionPerformed()**
- Lorsqu'un événement se produit (clic de souris), l'écouteur reçoit un objet **ActionEvent**.
- La méthode **getSource()** (héritée de **EventObject** superclasse des classes événements) permet de déterminer quel est l'objet source de l'événement, ou la méthode **getActionCommand()** renvoie la chaine de commande associée à l'action.

Traiter un événement

- Exemple:
  L'interface *MouseListener* correspond à une catégorie d'événements souris de type *MouseEvent*.

```
public interface MouseListener extends EventListener  {
    public void mousePressed(MouseEvent e) ;
```

/* appelé lorsqu'un bouton de la souris est pressé sur un composant l'argument **e** de type **MouseEvent** correspond à l'objet événement généré */

```
public void mouseReleased(MouseEvent e) ;
```

/* appelé lorsqu'un bouton de la souris est relâché sur un composant */

Traiter un événement

public void mouseClicked(MouseEvent e) ;

/* appelé lors d'un clic souris sur un composant (la souris n'a pas été déplacée entre l'appui et le relâchement du bouton) */

public void mouseEntered(MouseEvent e) ;

/* appelé lorsque la souris passe de l'extérieur à l'intérieur d'un composant */

public void mouseExited(MouseEvent e) ; }

/*appelé lorsque la souris sort d'un composant (la souris passe de l'intérieur à l'extérieur du composant) */

Intercepter un événement

- Lorsqu'on souhaite intercepter un événement, on doit le préciser dans son constructeur en appelant la méthode dans le style:
  addXXXListener(XXXListener objetEcouteur);
  - où l'argument *objetEcouteur* correspond à l'objet écouteur chargé de traiter l'événement.
- Exemple: La classe Component définit les méthodes suivantes :

public void addFocusListener(FocusListener l) ; // prise et perte du focus d'un champs
public void addKeyListener(KeyListener l) ; // événements clavier
public void addMouseListener(MouseListener l) ; // événements souris
public void addMouseMotionListener (MouseMotionListener l) ;
// événements liés au déplacement de la souris

Gérer un événement

- Pour gérer la fermeture de la fenêtre d'une application Java il nous faut créer un objet qui "*écoute*" les événements qui se produisent sur la fenêtre et détecte l'événement "fermeture de la fenêtre".
- C'est un objet "listener" ou *gestionnaire d'événements*.
- Il existe différents types de gestionnaires pour les différents événements qui peuvent se produire sur les composants d'une interface graphique.
  - Pour le composant *JFrame*, le *listener* s'appelle *WindowListener* qui définie les méthodes portant à la gestion des sept événements qui peuvent être gérés.

Gérer un événement

```
WindowListener win = new WindowListener() {
    public void windowActivated(WindowEvent e) {  /* fenêtre activée */  }
    public void windowClosed(WindowEvent e) {     /* fenêtre fermée */  }
    public void windowClosing(WindowEvent e) {
            System.exit(0);                       /* fermeture de la fenêtre */
    }
    public void windowDeactivated(WindowEvent e)  {
      /* fenêtre désactivée */  }
    public void windowDeiconified(WindowEvent e)  {
      /* fenêtre à l'état normal*/  }
    public void windowIconified(WindowEvent e)  {
      /* fenêtre à l'état réduit */  }
    public void windowOpened(WindowEvent e) {
      /* fenêtre visible */ }
};
```


Gérer un événement

- L'objet chargé de gérer les événements de la fenêtre est lourde
- On est obligé de définir des méthodes même pour des événements qu'on ne veut pas gérer.
- On utilise l'interface *WindowAdapter* à la place de *WindowListener.*
- La classe *WindowAdapter* implémente l'interface *WindowListener*, avec les sept méthodes vides.

Ainsi :

– définition du gestionnaire d'événements

– association du gestionnaire à la fenêtre .

Gérer un événement

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class MaFenetre extends JFrame {
    public MaFenetre() {  // le constructeur
            this.setTitle("Gestion de la fenêtre");
            this.setSize(new Dimension(300, 100));
            // création d'un gestionnaire d'événements
            WindowAdapter win = new WindowAdapter()  {
                    public void windowClosing(WindowEvent e) { System.exit(0); }
            };  // fin gestionnaire fermeture de la fenêtre
            this.addWindowListener(win);  // on ajoute le GdE à la fenêtre (conteneur)
    }
    public static void main(String[] args)  {
            MaFenetre fn = new MaFenetre();
            fn.setVisible(true);
    }
}
```


Gérer des événements

- La gestion de la souris avec rendue des coordonnées du clic et la fermeture de la fenêtre principale avec l'arrêt de l'application

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class MaFenetre extends JFrame {
    public MaFenetre() {
            super("Une fenêtre qui traite les clics souris");
            setSize(250, 150);
            addMouseListener( new EcouteurSouris() );
            addWindowListener( new EcouteurFenetre() );          }
    public static void main(String[] args)  {
            new MaFenetre().setVisible(true);
    }
}
```

Gérer des événements

```
class EcouteurSouris extends MouseAdapter {
    // Redéfinition de la méthode appelée lors d'un clic souris
    public void mouseClicked(MouseEvent e) {
      int x = e.getX();
      int y = e.getY();
      // Coordonnées du curseur de la souris au moment du clic
      System.out.println("clic au point de coordonnées " + x + ", " + y);
    }   }
// Méthode assurant la fermeture de la fenêtre
class EcouteurFenetre extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
          System.out.println("fermeture de la fenêtre");
          System.exit(0);
    }
}
```

(démo)

Les adaptateurs

- Pour chaque interface XXXListener possédant plusieurs méthodes, Java fournit une classe particulière XXXAdapter, appelée adaptateur, qui implémente toutes les méthodes de l'interface avec un corps vide.
- Pour définir un écouteur d'événements de type XXXEvent, il suffit alors de dériver l'écouteur de la classe XXXAdapter et de redéfinir uniquement les méthodes voulues.

Les interfaces

- Le mot clé "implements" permet de spécifier une ou des interfaces que la classe implémente.
- Cela permet de récupérer quelques avantages de l'héritage multiple.
- Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super classes. Ce mécanisme n'existe pas en Java à l'état pur.
- Les interfaces permettent de mettre en œuvre un mécanisme de remplacement.
- Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite.

Les interfaces

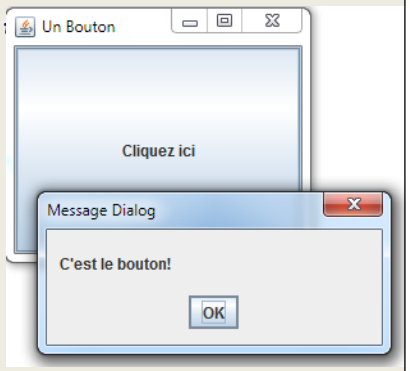
- C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (*qui implémentent cette interface*) possèdent les méthodes et les constantes déclarée dans celle-ci.
- Plusieurs interfaces peuvent être implémentées dans une même classe. Les interfaces se déclarent avec le mot clé *interface* et sont intégrées aux autres classes avec le mot clé *implements*.
- Exemple :
  public class MonApplet extends Applet implements ActionListener, MouseListener { ….. }
  // implémente deux interfaces

Les interfaces

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Frame01 extends JFrame implements ActionListener {
  JButton pressme = new JButton("Cliquez ici");
  public Frame01()   {  // le constructeur de la fenêtre
    super("Un Bouton"); // appel du constructeur parent
    setBounds(50,50,240,200);
    // opération de fermeture de la fenêtre
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // prend la référence du conteneur par défaut
    Container con = this.getContentPane();
    con.add(pressme);
    // met en place l'événement du bouton
    pressme.addActionListener(this);
    setVisible(true);
}
```


Les interfaces

```
public void actionPerformed(ActionEvent ae) {
   Object source = ae.getSource();
   if (source == pressme) //on a cliqué sur le bouton
      JOptionPane.showMessageDialog(null,"C'est le bouton!",
      "Message Dialog",JOptionPane.PLAIN_MESSAGE);
   // un message dans un box de dialogue
  }
  // La méthode principale
  public static void main(String args[]) {
     new Frame01();
  }
}
```



Les interfaces

- Un autre moyen de récupérer l'action d'un bouton dans la méthode *actionPerformed(ActionEvent evt)* est de tester le retour de la méthode *getActionCommand* (*classe ActionEvent*) qui renvoie la chaîne de caractères qui caractérise l'action à réaliser.

```
public void actionPerformed(ActionEvent evt)  {
   if (evt.getActionCommand().equals("Ajouter")) {
      JOptionPane.showMessageDialog(null,
      "bouton en marche") ;
   }
}
```


classe interne

- Une classe est dite interne lorsque sa définition est située à l'intérieure d'une autre classe (ou d'une méthode).
- Un objet d'une classe interne est toujours associé, au moment de son instanciation, à un objet d'une classe externe dont on dit qu'il lui a donné naissance.
- Un objet d'une classe interne a toujours accès aux champs et méthodes (même privés) de l'objet externe lui ayant donné naissance.
- Un objet d'une classe externe a toujours accès aux champs et méthodes (même privés) de l'objet interne auquel il a donné naissance.

classe interne

- Exemple:

```
class MaFenetre extends JFrame {
    public MaFenetre () {
        super("Une fenêtre qui gère sa fermeture") ;
        setSize(300, 200) ;
        addWindowListener( new EcouteurFermer() );
        }
        // classe interne à la classe MaFenetre
        class EcouteurFermer extends WindowAdapter {
            public void windowClosing(WindowEvent e) {
                System.exit(0);  }
    }  // fin classe interne
}
```


classe anonyme

- C'est une classe imbriquée sans nom.
- A la compilation elle aura le nom: *Factive$1.class*
- Exemple :

```
public class Factive {
...
    JButton b = new JButton("cliquez ici");
    b.addActionListener ( new ActionListener ()   {
    public void actionPerformed ( ActionEvent e )  {
    System.out.println("un clic");
    } } );
    ...
}
```


Gestion des menus

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
import javax.swing.event.*;
public class monMenu extends JFrame {
   public monMenu() {
      super("Menu");
      this.setBounds(100,100,640,480);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      this.getContentPane().setBackground(Color.lightGray);
      // Menubar pour JFrame
      JMenuBar menuBar = new JMenuBar();
      // Ajouter menubar au frame
      this.setJMenuBar(menuBar);
      // Définir deux drop down menu pour le menubar
      JMenu fileMenu = new JMenu("Fichier");
      JMenu editMenu = new JMenu("Edition");
      menuBar.add(fileMenu);
      menuBar.add(editMenu);
     // Ajouter simple menu item to the drop down menu
      JMenuItem newAction = new JMenuItem("Nouveau");
      JMenuItem openAction = new JMenuItem("Ouvrir");
      JMenuItem exitAction = new JMenuItem("Sortie");
      JMenuItem cutAction = new JMenuItem("Couper");
      JMenuItem copyAction = new JMenuItem("Copier");
      JMenuItem pasteAction = new JMenuItem("Coller");
      fileMenu.add(newAction);
      fileMenu.add(openAction);
      fileMenu.addSeparator();
      fileMenu.add(exitAction);
      editMenu.add(cutAction);
      editMenu.add(copyAction);
      editMenu.add(pasteAction);
      // Ajouter un 'listener ' au 'Nouveau' menu item
      newAction.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {
            System.out.println("Nouveau");
         }      });
      openAction.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {
            System.out.println("Ouvrir");
         }      });
      // Ajouter un 'listener' au 'Sortie' menu item
      exitAction.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {
            System.out.println("Fermer");
            System.exit(0);
         }      });
   }
   public static void main(String[] args) {
      monMenu me = new monMenu();
      me.setVisible(true);
   }
}
```


Méthodes dynamiques

- Normalement les composants d'un conteneur sont créés en même temps que le conteneur et restent affichés en permanence.
- Cependant, on peut ajouter un nouveau composant à un conteneur (méthode *add* de la classe *Container*) ou supprimer un composant d'un conteneur (méthode *remove* de la classe *Container*).
- Si l'une de ses opérations est effectuée après l'affichage du conteneur, il faut forcer le gestionnaire de mise en forme à recalculer les positions des composants (1) soit en appelant la méthode *validate* (classe *Component*) du conteneur, (2) soit en appelant la méthode *revalidate* (de la classe *JComponent*) des composants du conteneur.

Méthodes dynamiques

```
import java.awt.* ;  import java.awt.event.* ;
import javax.swing.* ;  import javax.swing.event.* ;
class MaFenetre extends JFrame {
    private JButton MonBouton ;
    public MaFenetre() {
        super("Une fenêtre avec des boutons dynamiques");
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contenu = getContentPane();
        contenu.setLayout(new FlowLayout());
        MonBouton = new JButton("Creation de boutons");
        contenu.add(MonBouton);
        MonBouton.addActionListener(new EcouteurBouton(contenu));
    }
```

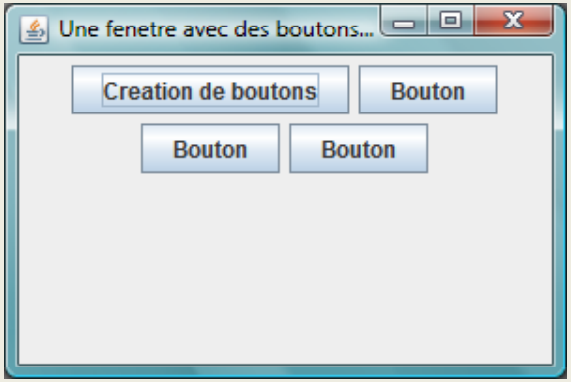

Méthodes dynamiques

```
class EcouteurBouton implements ActionListener {
        private Container contenu ;
        public EcouteurBouton (Container contenu) {
                this.contenu = contenu ;  }
        public void actionPerformed(ActionEvent e) {
                JButton NvBouton = new JButton("Bouton") ;
                contenu.add(NvBouton) ;   contenu.validate() ;
                //recalcul les positions des composants du conteneur
        }       }

public class MonProgEvtBouton2 {
   public static void main(String args[]) {
        JFrame fen = new MaFenetre() ;  fen.setVisible(true) ;
   }  }
```


Méthodes dynamiques

- Le bouton "Création de bouton" provoque la création et par la suite l'apparition du nouveau bouton à l'écran



Programmation événementielle et réseau avec le langage Java

Module I6

IUT d'Aix-en-Provence
Réseaux et Télécommunications
Février 2011
Ivan Madjarov

Le langage Java – Partie 7

Système Entrée-Sortie, Flux et Fichiers

Les flux d'entrée-sortie

- La communication d'une application Java avec le monde extérieur s'effectue par des Stream (flots):
  - Flots de données binaires (bytes)
  - Flots de caractères (char)
- Quel que soit la communication établie (fichiers ou réseau) et le type de données transportée (Unicode ou binaire), la procédure générale consiste à :
  - ouvrir un flot ("Stream") de communication ;
  - utiliser le flot pour lire ou écrire des données ;
  - fermer le flot de communication.

Les flots (Stream)

- *java.io* est le package des entrées/sorties.
- Le package intègre des classes de gestion :
  - fichier classique,
  - des classes dédiées à la gestion des flux de données.
- Les classes de base de *java.io* sont :
  - *InputStream,* gère des données en entrée (lecture),
  - *OutputStream,* gère des données en sortie (écriture).
- Ces deux classes abstraites ne sont pas utilisées directement mais plutôt comme intermédiaires vers des sous-classes mieux adaptées.

Les flots (Stream)

- Le flot de communication établi est unidirectionnel. On utilisera les classes suivantes pour :
- Les opérations d'écriture :
  - *InputStream* pour les flots de données binaires ;
  - *Writer* pour les flots de caractères ;
- Les opérations de lecture :
  - *OutputStream* pour les flots de données binaires ;
  - *Reader* pour les flots de caractères;

Les flots (Stream)

- Exemples de sous-classes de *InputStream* et *OutputStream* :
  - *BufferedReade*r : Lecture de texte depuis un flux d'entrée (caractères, tableaux et chaînes de caractères).
  - *BufferedWriter* : Écriture de texte dans un flux de sortie (caractères, tableaux et chaînes de caractères).
  - *DataInputStream* : Lecture de données Java primitives.
  - *DataOutputStream* : Écriture de données Java primitives.
  - *FileInputStream* : Lecture d'octets depuis un fichier.
  - *FileOutputStream* : Écriture d'octets dans un fichier.

Les flots (Stream)

- *java.io.File* les méthodes des fichiers.

String getName();	Retourne le nom du fichier.
String getPath();	Retourne la localisation du fichier en relatif.
String getAbsolutePath();	Idem mais en absolu.
String getParent();	Retourne le nom du répertoire parent.
boolean renameTo(File newFile);	Permet de renommer un fichier.
boolean exists() ;	Est-ce que le fichier existe ?
boolean canRead();	Le fichier est t-il lisible ?
boolean canWrite();	Le fichier est t-il modifiable ?
boolean isDirectory();	Permet de savoir si c'est un répertoire.
boolean isFile();	Permet de savoir si c'est un fichier.
long length();	Quelle est sa longueur (en octets) ?
boolean delete();	Permet d'effacer le fichier.
boolean mkdir();	Permet de créer un répertoire.
String[] list();	On demande la liste des fichiers localisés dans le répertoire.

Les flots (Stream)

- Exemples (*java.io.File*):

```
import java.io.File;
public class monFile {
  public static void main(String argv[]) {
   File f = new File("java/h.class");
   System.out.println(f.exists());   // --> true
   System.out.println(f.canRead());  // --> true
   System.out.println(f.canWrite()); // --> false
   System.out.println(f.length());   // --> 11345
   File d = new File("java/");
   System.out.println(d.isDirectory()); // --> true
   String[] files = d.list();
   for (int i=0; i < files.length; i++)
       System.out.println(files[i]);
  }
}
```

```
true
true
true
2514
true
Cercle$Centre.class
Cercle.class
Cercle.java
ClientTCP.class
ClientTCP.java
```

Les flots (Stream)

- *java.io.File* permet d'accéder en lecture et en écriture à un fichier.

```
… … …
FileInputStream fin = new FileInputStream("source.txt");
byte[] data = new byte[ fin.available() ];
// available() retourne le nombre d'octets accessibles à lire
fin.read(data);
fin.close();
… … …
FileOutputStream fout = new FileOutputStream("cible.txt");
fout.write(data);
fout.close();
```


Les flots (Stream)

- *java.io.Data* permet de lire et d'écrire des types primitifs et des lignes sur des flux.

```
FileInputStream fis = new FileInputStream("source.dat");
DataInputStream dis = new DataInputStream(fis);
int i = dis.readInt();
double d = dis.readDouble();
String s = dis.readLine();
FileOutputStream fos = new FileOutputStream("cible.dat");
DataOutputStream dos = new DataOutputStream(fos);
dos.writeInt(123);
dos.writeDouble(123.456);
dos.writeChars("Une chaine");
```


Les flots (Stream)

- *java.io.PrintStream* permet de manipuler un *OutputStream* au travers des méthode *print()* et *println()*.
  - *PrintStream ps = new PrintStream(new FileOutputStream("cible.txt", true));*
- Le paramètre "*true*" autorise l'ajout de nouvelles lignes à la fin du fichier "*cible.txt*".
  - *ps.println("Une ligne");*
  - *ps.println("Une deuxième);*
  - *ps.print("Une autre ligne");*
  - *ps.print("123 et fin");*
  - *ps.flush(); ps.close();*

Les flots (Stream)

- *Les exceptions* (rappel): Elles permettent de séparer un bloc d'instructions de la gestion des erreurs pouvant survenir dans ce bloc. Format:

```
try {
// Code pouvant lever des Exception: ouverture de fichier, écriture
   dans un fichier
}
catch (Exception e) {
// Gestion des Exceptions : ici le code pour récupérer le programme
   ou se contenter d'afficher un message et poursuivre
}
```

Le principe des exceptions

- *Emetteur*: code où il faut anticiper un problème:
  - détecter l'erreur, probablement avec un *if* … créer une nouvelle Exception et la lancer (*throw*)
  - laisser la JVM détecter l'erreur, créer et lancer une Exception
- *Recepteur*: code dans l'appelant (quelque part dans la pile d'appel)
  - tenter un "*try*", en espérant que ca marche comme prévu (écrire le code comme si tout marchait)
  - se préparer à attraper (*catch*) *une Exception* et effectuer des opérations de récupération ou afficher un message

Lire un fichier texte

```
String chaine="";
String fichier="source.txt";
try {
   // Ouvrir le fichier
  InputStream ips = new FileInputStream(fichier);
   // Initialiser le flot de lecture
  InputStreamReader ipsr = new InputStreamReader(ips);
  // Unité de lecture
  BufferedReader br = new BufferedReader(ipsr);
  String ligne;
  // Lire les lignes du fichier
  while ((ligne = br.readLine()) != null) {
      chaine += ligne+"\n";
  }
  br.close();
} catch (Exception e){ }
```


Le langage Java – Partie 8

Programmation réseau avec Java

Introduction

- Tout langage de programmation "moderne" offre une interface de programmation réseau
  - Java, C#, Python, etc.
- Java propose le paquetage *java.net*. Il fournit des facilités pour la programmation réseau par des sockets en implémentant les protocoles TCP/IP et UDP.
  - Le protocole UDP permet d'établir une connexion, sur une socket, en mode non connecté
    - Transmission de données en mode datagramme.
  - Le protocole TCP/IP permet d'établir une connexion en mode connecté.
    - Transmission de données en mode connecté

La Socket

- La Socket (*connecteurs réseau*) représente une interface de programmation pour les communications entre processus.
- Il existe généralement quatre types de sockets :
  - Une *socket datagram* permet une communication bidirectionnelle qui n'est pas séquencée. Un processus utilisant ce type de socket peut donc recevoir les données dans un ordre différent de l'ordre de départ. il s'agit du protocole UDP.
  - Une *socket stream* permet une communication bidirectionnelle, sûre, séquencée et un flux de données sans duplication pouvant entraîner une fragmentation des paquets transmis. Il s'agit du protocole TCP.
  - Une *socket raw* et une *socket sequenced packet*.

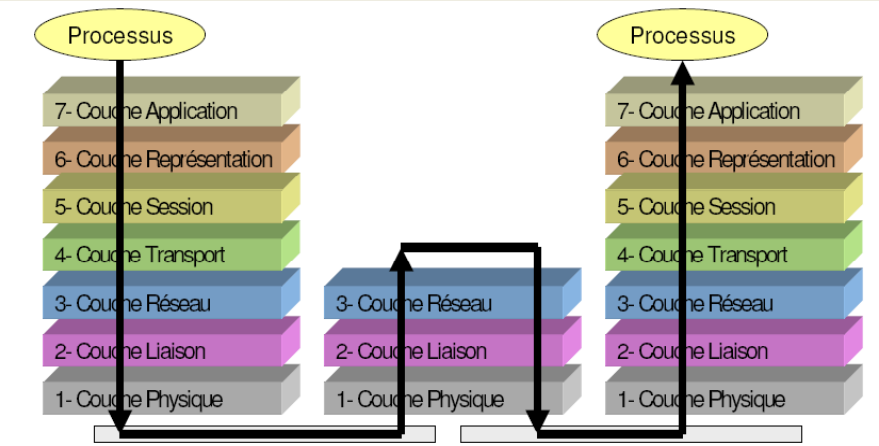
Introduction

- Le modèle OSI (*Open System Interconnexion*)

7- Couche Application	Un ensemble de services standard : terminal virtuel, transfert de fichiers, ...
6- Couche Représentation	Codage des informations sous une forme standard
5- Couche Session	Gestion complète d 'une session de communications entre 2 utilisateurs.
4- Couche Transport	Transport de l 'information entre deux processus (Multiplexage)
3- Couche Réseau	Transport de l 'information entre deux points du réseau (Routage)
2- Couche Liaison	Transport, sans erreur entre deux points, des blocs de données (trames)
1- Couche Physique	Transport de l 'information comme une suite de bits

Introduction

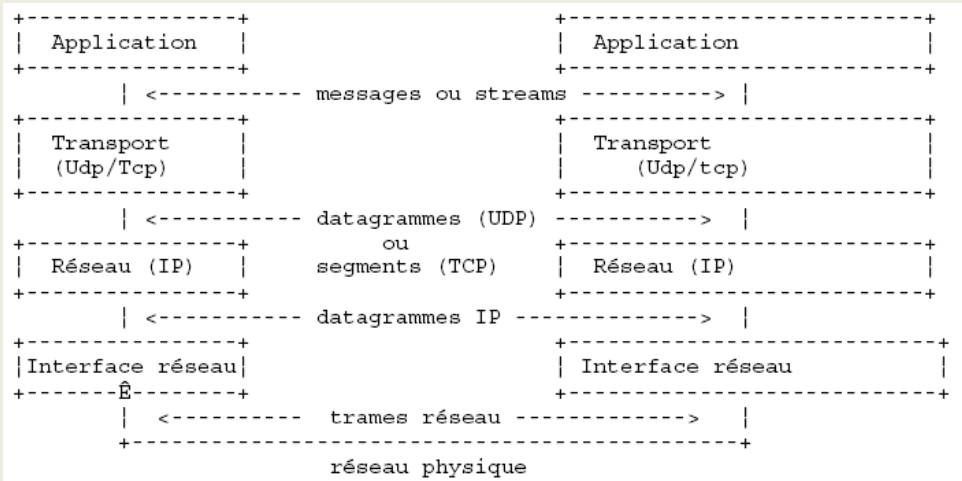
- Routage et Passerelle



Routage entre plusieurs supports de communication

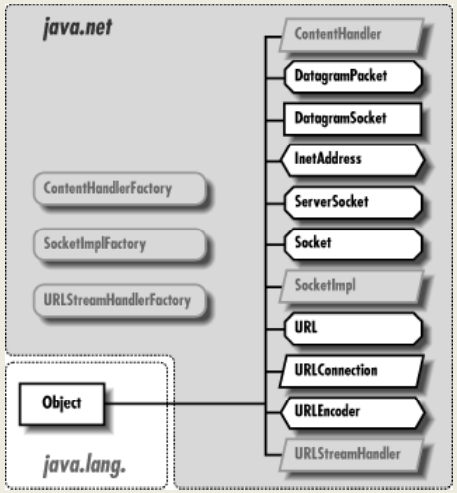
Introduction

- Le schéma montre le parcours de l'information



Introduction

- Le package *java.net* contient toutes les classes orientées réseau bas niveau de l'API JAVA.



Mode datagramme (UDP)

- Le service offert par la couche IP est l'envoi (et surtout le routage) de datagrammes de l'émetteur vers le destinataire, *en mode non connecté*
  - les datagrammes sont indépendants entre eux, même lorsqu'ils font partie d'un même message.
  - la remise du datagramme au destinataire n'est pas garantie!
- Dans certains cas de programmation réseau il est demandé d'envoyer des données d'un ordinateur à un autre sans avoir à se soucier de la bonne réception de ces données par le destinataire.

Mode datagramme (UDP)

- Cette fonctionnalité est assurée par le protocole UDP (*User Datagram Protocol*).
- Le paquetage *java.net* fourni deux classes pour faciliter la programmation réseau en mode datagramme :
  - *DatagramPacket*
  - *DatagramSocket*.
- Les données sont mises dans un objet de type *DatagramPacket* qui est envoyé sur le réseau par le biais d'un objet de type *DatagramSocket*.

Mode datagramme (UDP)

- La classe *DatagramPacket* fournit deux constructeurs : un pour les paquets à recevoir, l'autre pour les paquets à envoyer.
  - *public DatagramPacket(byte buffer[], int taille)*

Construit un objet pour recevoir un datagramme.
- buffer correspond à la zone de réception
- taille maximale des datagrammes à recevoir.

  - *public DatagramPacket(byte buffer[], int taille, InetAddress adresse, int port)*

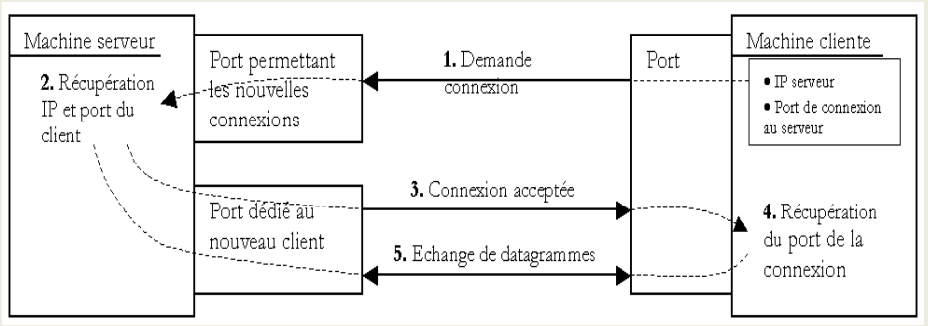
Construit un objet pour envoyer un datagramme.
- buffer correspond à la zone d'envoie
- taille correspond à la taille du datagramme à envoyer
- adresse destinataire de la datagramme
- port du UDP

Mode datagramme (UDP)

- Le principe de la programmation réseau en mode datagramme en Java se base (1) sur l'envoie des datagrammes et (2) recevoir des datagrammes.
- Pour envoyer des datagrammes en Java nous devons suivre les étapes suivantes :
  - Obtenir l'adresse du destinataire et la mettre dans une instance de la classe *InetAddress*.
  - Mettre les données et l'adresse dans une instance de la classe *DatagramPacket*.
  - Créer une instance de la classe *DatagramSocket* et lui confier l'envoi du datagramme.

Mode datagramme (UDP)

- Pour recevoir des datagrammes en Java il faut suivre les étapes suivantes :
  - Créer une instance de la classe *DatagramSocket* qui attend l'arrivée de données à travers le réseau.
  - Créer une instance de la classe *DatagramPacket* qui reçoit les données par l'instance de *DatagramSocket*.



Mode datagramme (UDP)

- Exemple:
  - trouver l'adresse IP correspondant à un nom de machine

```
C:\Work\Java>javac ResoudreNom.java

C:\Work\Java>java ResoudreNom www.yahoo.fr
www.yahoo.fr -> 217.146.186.51

C:\Work\Java>
```

```
import java.net.* ;
import java.io.* ;
public class ResoudreNom {
    public static void main(String[] args) {
      InetAddress adresse;
      try {  adresse = InetAddress.getByName(args[0]);
             System.out.println("Nom : "+args[0]+"
  IP : "+adresse.getHostAddress());
} catch(UnknownHostException e)  {
  System.err.println(args[0]+" est inconnu\n");
} } }
```


Mode datagramme (UDP)

- Deux programmes pour illustrer le mode datagramme:
  - Le programme "*Emetteur*" permet l'envoi vers le programme "*Recepteur*" des messages à travers le réseau.
  - Le programme "*Recepteur*" se charge d'afficher le message envoyé.

```
C:\Work\Java>java CoteEnvoi localhost

C:\Work\Java>
```

```
C:\Work\Java>java CoteReception
Recepteur installe
Demonstration du mode datagramme


C:\Work\Java>
```

Émetteur

- La première chose est de résoudre le nom de la machine destinatrice (passé en argument) en adresse réseau.
  - *adr = InetAddress.getByName(args[0]);*
- Ensuite une chaîne de caractères est transformée en suite d'octets pour de les transmettre sur le réseau.
  - *String message = "Démonstration du mode datagramme";*
  - *byte[] tampon = new byte[message.length()];*
- La transformation se fait par la méthode *getBytes()* de la classe *String*.
  - *tampon = message.getBytes();*

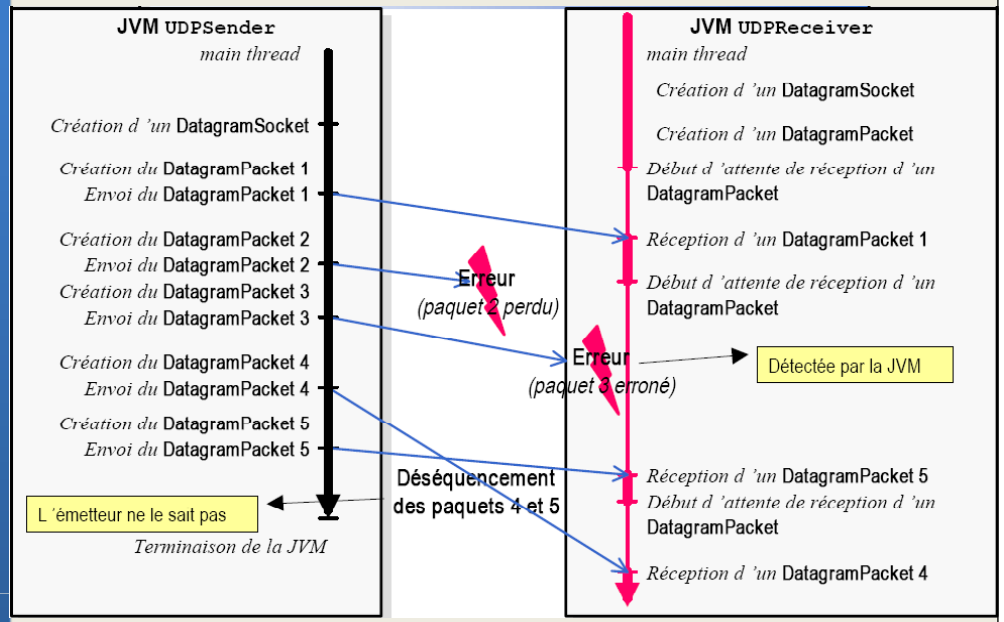
Émetteur

- Ensuite, Il nous faut, construire un datagramme en indiquant l'emplacement des données à transmettre (*tempon*), leur longueur (*tampon.length*) et enfin l'adresse du destinataire (*adr*) et le port de l'écoute (*port*).
  - *DatagramPacket paquet = new DatagramPacket(tampon, tampon.length, adr, port);*
- Enfin on ouvre une *DatagramSocket* en utilisant sa méthode *send()* pour l'envoi du datagramme.
  - *DatagramSocket sock = new DatagramSocket();*
  - *sock.send(paquet);*

Récepteur

- Une zone mémoire tampon pour recevoir les données.
  - *byte[] tampon = new byte[200];*
- Ensuite nous créons un *DatagramSocket* pour écouter sur le port de destination en attente de données.
  - *DatagramSocket sock = new DatagramSocket(port);*
- L'ordre d'attente se fait par la méthode *receive()*
  - *sock.receive(paquet);*
- La méthode se charge de placer les données dans un *DatagramPacket* gérant le tampon.
  - *DatagramPacket paquet = new DatagramPacket(tampon, tampon.length);*

Déroulement de l'exécution



Le client UDP

```
import java.io.*;
import java.net.*;
public class ClientUDP {
  public static void main(String argv[]) throws Exception {
   String Message = "Bonjour tout le monde !";
   int l = Message.length();
    // Résoudre l'adresse du récepteur
   InetAddress serv = InetAddress.getByName("localhost");
    // Coder en octets
   byte buf[] = Message.getBytes();
    // Préparer la datagramme
    DatagramPacket data = new DatagramPacket(buf,l,serv,5000);
    // Ouvrir la socket
   DatagramSocket socket = new DatagramSocket();
    // Envoyer le paquet
   socket.send(data);
  }
}
```


Le Serveur UDP

```
class ServeurUDP {
  public static void main(String argv[]) throws Exception {
    int port = 5000; int taille = 1024; String message;
     byte buffer[] = new byte[taille];
    // Créer la socket pour installer le port d'écoute
     DatagramSocket socket = new DatagramSocket(port);
     System.out.println("Serveur UDP sur le port 5000");
     while(true) {
        // Réception de la datagramme
        DatagramPacket packet = new
   DatagramPacket(buffer,buffer.length);
        socket.receive(packet);
        // Décodage du message
        message=new String(buffer);
        System.out.println(message);
     }
   }
}
```

Programmation événementielle et réseau avec le langage Java

Module I6

IUT d'Aix-en-Provence
Réseaux et Télécommunications
Mars 2011
Ivan Madjarov

Partie 8
Programmation en Java par Socket en mode connecté

Programmation réseau en Java

Introduction

- Java propose un paquetage spécial *java.net*.
- Le paquetage fournit des facilités pour la programmation réseau par des sockets en implémentant les protocoles TCP/IP et UDP.
  - Le protocole UDP permet d'établir une connexion, sur une socket, en mode non connecté
    - Transmission de données en mode datagramme.
  - Le protocole TCP/IP permet d'établir une connexion en mode connecté.
    - Transmission de données en mode connecté

La Socket

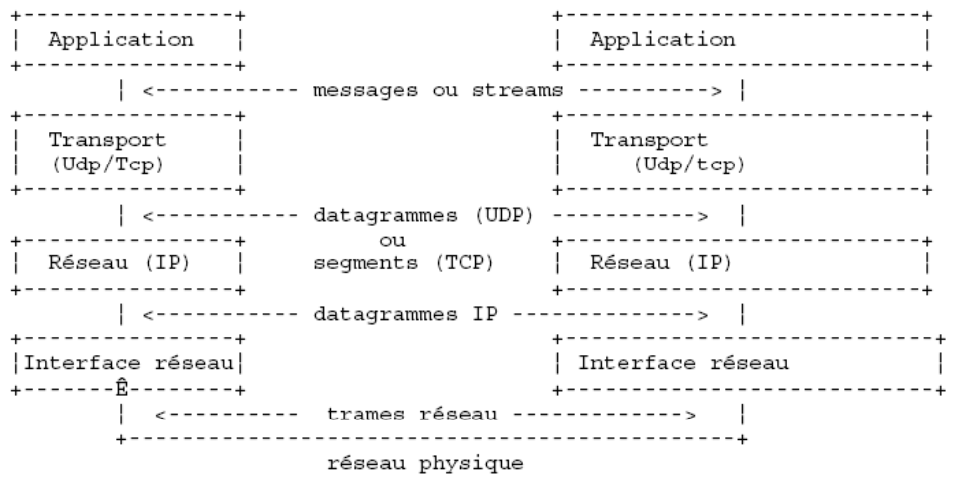
- La Socket (*connecteurs réseau*) représente une interface de programmation pour les communications entre processus.
- Il existe généralement quatre types de sockets :
  - Une *socket datagram* permet une communication bidirectionnelle qui n'est pas séquencée. Un processus utilisant ce type de socket peut donc recevoir les données dans un ordre différent de l'ordre de départ. il s'agit du protocole UDP.
  - Une *socket stream* permet une communication bidirectionnelle, sûre, séquencée et un flux de données sans duplication pouvant entraîner une fragmentation des paquets transmis. Il s'agit du protocole TCP.
  - Une *socket raw* et une *socket sequenced packet*.

Les sockets (*connecteurs réseau*)

- Les données transitent sur l'Internet dans des paquets d'octets de taille limitée qu'on appel *datagrammes.* Ils sont formés d'un entête :
  - adresse,
  - numéro de port de l'expéditeur et du destinataire,…
  - une charge utile composée des données.
- Un transfert unique peut nécessiter plusieurs paquets.
- Les sockets permettent la gestion d'un transfert de données comme un flux ordinaire (les paquets sont triés à l'arrivée).

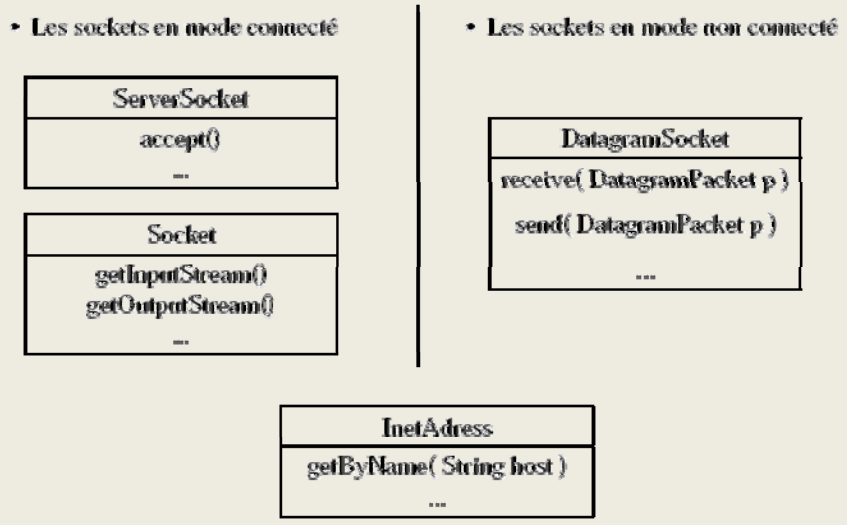
Connexion réseau

- Le schéma montre le parcours de l'information



Mode UDP et TCP

- Les principales classes de ***java.net.*;***



Le mode connecté (TCP)

- Le mode connecté garanti l'arrivée en séquence des données émises.
- Le mode est assuré au niveau de la couche de transport selon le modèle OSI par le protocole TCP (*Transmission Control Protocol*).
- Le paquetage java.net fourni les classes
  - Socket
  - ServerSocket

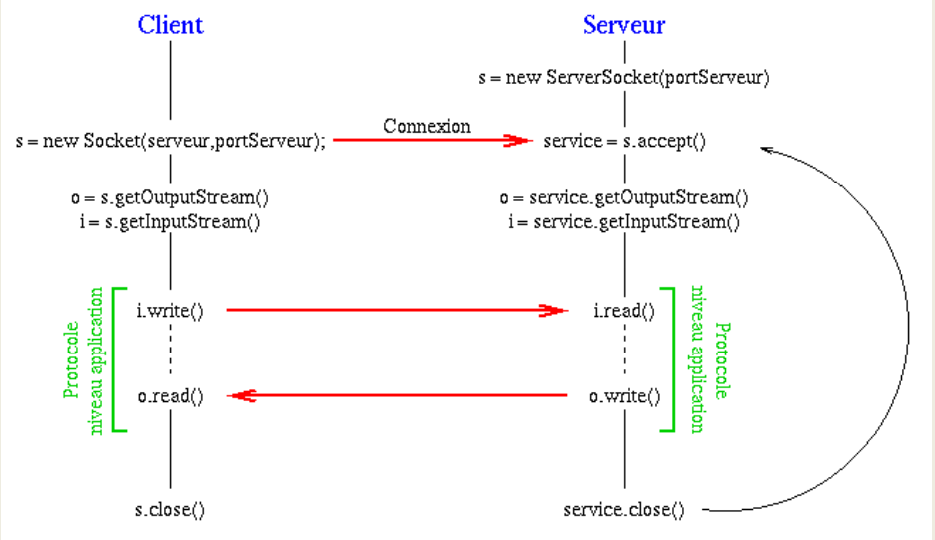
  pour travailler avec le mode connecté.
- Le mode connecté est une interconnexion stable entre un client et un serveur.

Le mode connecté (TCP)

- Les étapes côté serveur :
  1. Instancier la classe ServerSocket et l'instruire à écouter sur un certain port.
  2. Accepter les connexions par la méthode accept() et créer un objet Socket pour référencer la nouvelle connexion.
  3. Passer la nouvelle connexion au programme approprié.
  4. Fermer la connexion par la méthode close().
- Les étapes côté client :
  1. Se connecter au service approprié en instanciant la classe Socket et en lui passant comme paramètres l'adresse du serveur et le port.
  2. Lorsque l'échange est terminé ferme la connexion par la méthode close().

Le mode connecté (TCP)

- Le Client et le Serveur en mode connecté



Les sockets (*connecteurs réseau*)

- Les classes *java.net.Socket* et *java.net.ServerSocket* implémentent les sockets TCP/IP sous Java.
- Les Sockets assurent 7 opérations élémentaires:
  - connexion à une machine distante (Socket),
  - envoi de données (Socket),
  - réception de données (Socket),
  - fermeture d'une connexion (Socket),
  - attachement à un port (ServerSocket),
  - attente de demande de connexion émanant de machines distantes (ServerSocket),
  - acceptation de demandes de connexion sur le port local (ServerSocket).

Les sockets (*connecteurs réseau*)

- Mise en œuvre du modèle Client-Serveur:
  - La Socket est créée sur le client.
  - Le ServerSocket fonctionne sur le serveur en attente de connexion (méthode accept()).
  - La Socket tente de se connecter sur le serveur distant.
  - Connexion établie, le ServerSocket génère une Socket pour communiquer avec le client.
  - La connexion établie, les systèmes Client et Serveur établissent un canal de communication par flux à partir de leurs Sockets pour échanger des données.
  - Après fin des échanges, l'un des interlocuteurs clôt le canal de communication.

Les sockets (*connecteurs réseau*)

- Constructeur :
  - *public Socket(String host, int port) throws UnknownHostException, IOException;*
  - *public ServerSocket(int port) throws IOException;*
- Méthodes :
  - *host (String) :* nom de la machine vers laquelle la socket est créée.
  - *addresse (InetAddress) :* adresse IP de la machine vers laquelle la socket est créée.
  - *port (int) :* numéro du port sur lequel la socket est créée.
  - *localAddr (InetAddress)* : adresse IP locale à la quelle la socket est associée.
  - *localPort (int)* : numéro du port local auquel la socket est associée.
  - *public void close() throws IOException* : ferme la socket.
  - *public inputStream getInputStream() throws IOException* : retourne le flux d'entrée associé à la socket.

Les sockets (*connecteurs réseau*)

- *public outputStream getOutputStream() throws IOException* : retourne le flux de sortie associé à la socket.
- *public void close() throws IOException* : ferme le *ServerSocket*.
- *public Socket accept() throws IOException* : place le *ServerSocket* en attente de requête d'ouverture de *socket*.
  - Cette attente est bloquante. Une *socket* est automatiquement générée et retournée lors de cette requête.

Les sockets (*connecteurs réseau*)

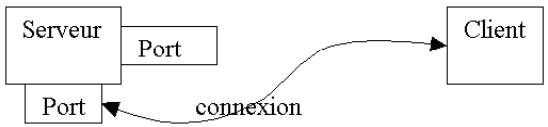
- Réalisation d'un Client en Java :
  - ouverture d'un *socket (new socket(host, port))*,
  - ouverture d'un flot d'entrée et de sortie sur le *socket(in=new BufferReader(...), out=new PrinterWritter(...)* ,
  - lecture et écriture sur les flots en accord avec les protocoles du serveur *in.readLine(...), out.println(...)*,
  - fermeture des flots: *in.close(), out.close()*,
  - fermeture du socket: *socket.close()*

Les sockets (*connecteurs réseau*)

- Réalisation d'un Serveur :
  - ouverture d'un socket serveur,
  - attente de connexion et ouverture d'un socket client,
  - ouverture d'un flot d'entrée et de sortie sur le socket client,
  - lecture et écriture sur les flots en accord avec les protocoles du client (présenté précédemment),
  - fermeture des flots, fermeture du socket serveur et du socket client.

Les sockets (*connecteurs réseau*)

- Une application Client se connecte par une adresse IP sur un numéro de port.
- Le serveur accepte la connexion et fournit une nouvelle socket avec un nouveau port de communication pour continuer à écouter sur la socket originale.



- Du côté du client, si la connexion est acceptée, une socket est créée pour assurer la communication avec le serveur.

Serveur d'heure TCP

```
import java.net.*; import java.io.*; import java.util.*;
class serveurHeure { // Retourne au client la date et l'heure
  public static void main(String args[]) {
    try {
      ServerSocket sks = new ServerSocket(8080);
       System.out.println("Serveur en attente");
      Socket sk = sks.accept();
      System.out.println(sk.toString());
      DataOutputStream out = new
  DataOutputStream(sk.getOutputStream());
      out.writeBytes(new Date()+"\n"); // ecr. date et heure
      sk.close();
      sks.close();
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  } }
```

```
>java serveurHeure
Serveur en attente
Socket[addr=/127.0.0.1,port=49409,localport=8080]
>
```


Le Client TCP

```
// Client cherchant l'heure sur le serveur
import java.io.*;
import java.net.*;
public class ClientTCPHeure {
  public static void main(String args[]) {
     String ligne;
    try {
       Socket sk = new Socket("localhost",8080);
       DataInputStream is = new
                  DataInputStream(sk.getInputStream());
       while ((ligne = is.readLine()) !=  null)
         System.out.println(ligne); // message du serveur
    } catch (Exception e) { }
  }
}
```

```
>java ClientTCPHeure
Wed Nov 25 08:58:32 CET 2009
>
```


Client TCP émet un message

```
import java.net.*;  import java.io.*;
public class ClientTCP {
   Socket sock = null;
   public ClientTCP(String host, int port, String data) {
     try {
       sock = new Socket(host, port);
       PrintStream output = new
                    PrintStream(sock.getOutputStream());
       output.println(data);
       sock.close();
     } catch(IOException e) { }
   }
   public static void main(String[] args) {
       ClientTCP client = new
                    ClientTCP("127.0.0.1",1500,"Bonjour!");
   }
}
```

Serveur TCP traite le message

```
import java.net.*;
import java.io.*;
public class ServeurTCP {
  int port=1500; // Port de connexion
  ServerSocket SkS;
  Socket SkC;
  BufferedReader input;
  String message;
  public ServeurTCP() {
   try {
    SkS = new ServerSocket(port);
    System.out.println("Serveur en attente sur port: " +
                                    SkS.getLocalPort());
   //  Serveur en attente infinie: (1) connexion d'un client,
   (2) réception de messages, (3) fermeture de la connexion
   par le client
```


Les sockets (*connecteurs réseau*)

```
while(true) {
  SkC = SocketServeur.accept();
  System.out.println("Nouvelle connexion acceptée: " +
                                    SkC.getInetAddress());
   input = new BufferedReader(new
                    InputStreamReader(SkC.getInputStream()));
   // Réception et affichage des messages du client
   try {
      while(true) {
          message = input.readLine();
          if (message == null) break;
          System.out.println(message);
      }
   } catch (IOException e) {
          System.err.println(e.getMessage());
}
```


Les sockets (*connecteurs réseau*)

```
  // Fermeture de la connexion par le client
  try {
    SkC.close();
    System.out.println("TCP Serveur STOP!");
    break;
  } catch (IOException e) { System.out.println(e); }
}
} catch (IOException e) { System.out.println(e); }
}
   // Application principale avec appel du constructeur
   public static void main(String[] args)   {
       new ServeurTCP();
   }
}
```


Les sockets (*connecteurs réseau*)

- Le côté client:

```
C:\Work\Java>java ClientTCP
Message 'Bonjour!' envoye a 127.0.0.1 sur port 1500

C:\Work\Java>
```

- Le côté serveur:

```
C:\Work\Java>java ServeurTCP
Serveur en attente de clients sur le port: 1500
Nouvelle connexion acceptÛe: /127.0.0.1
Bonjour!
TCP Serveur STOP!

C:\Work\Java>
```

Une requête HTTP 1.0

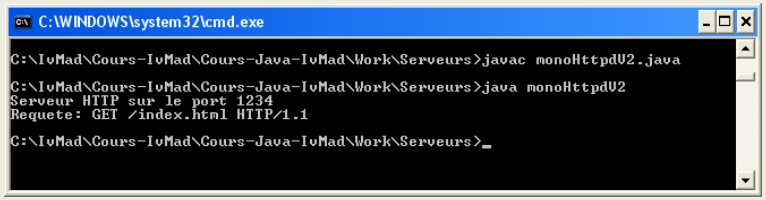
- Une ligne de requête comprend trois éléments devant être séparés par un espace :
  - La méthode (GET, POST, …)
  - L'URL (*Uniform Resource Locator*)
  - La version du protocole utilisé par le client (HTTP/1.0)
- Les champs d'en-tête de la requête : lignes facultatives permettant de donner des informations sur la requête et/ou le client (Navigateur, système d'exploitation, ...).
- Le corps de la requête : lignes optionnelles séparées une ligne vide et permettant un envoi de données par une commande POST lors de l'envoi de données par un formulaire.

Serveur HTTPD

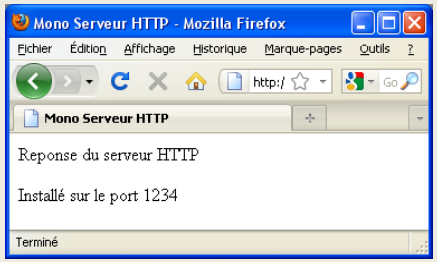
```
import java.net.*;  import java.io.*;  import java.util.*;
public class monoHttpd {                                    // Le serveur traite une seule connexion à la fois
 public static void main(String[] argv) throws IOException {
   ServerSocket ss = new ServerSocket(1234);                // Installation du serveur
   System.out.println("Serveur HTTP sur le port 1234");
   Socket sock = ss.accept();                               // ça boucle en attente de requête
   try {  // Formatage de la sortie
       OutputStream out = sock.getOutputStream();      // Lecture de la requête
       String req = (new BufferedReader(new InputStreamReader(sock.getInputStream()))).readLine();
       System.out.println( "Requête: "+req );             //  Traitement de la requête "GET /index.html HTTP/1.1"
       String str[] = req.split(" ");                      //  Décomposition par " " en éléments de tableau
       if (str[0].equals("GET")) {                         // Vérifier si la requête démarre par "GET"
         req = str[1].substring(1);                        // On enlève '/' devant le nom du fichier
         try {
            FileInputStream fis = new FileInputStream(req);
            byte[] data = new byte[fis.available()];       // available() returns the number of bytes that can be read
            fis.read(data);
            out.write(data);
        } catch ( FileNotFoundException e ) {  new PrintStream(out).println("404 Not Found");   }
       } else {  new PrintStream(out).println("400 Bad Request");  }
       sock.close();
    } catch ( IOException e ) { System.out.println("I/O error " + e.getMessage()); }
 }  }
```


Serveur HTTPD

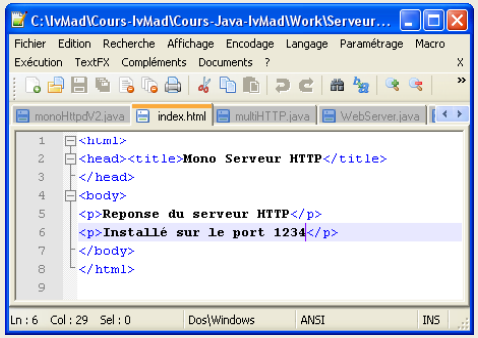
Le côté Serveur



La page 'index.html'



Le fichier 'index.html'



Partie 9
Les Threads dans le langage Java

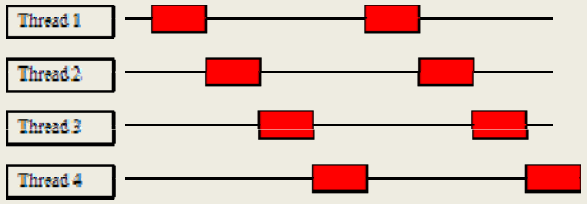
Programmation par Threads en Java

Les Threads

- Un Thread n'est pas un processus.
- Les processus existent dans des espaces virtuels isolés
- Les Threads sont des traitements (file) qui évoluent ensemble au sein d'un même processus.
- Les Threads partagent la même mémoire contrairement aux processus.
- Un Thread est donc une portion de code capable de s'exécuter en parallèle à d'autres traitements.
  - S'adresser de manière personnelle à plusieurs clients simultanément comme les serveurs HTTP ou les Chat.
  - Faire des traitements en tâche de fond, c'est le cas de la coloration syntaxique des éditeurs.

Les Threads

- Les Threads ne s'exécutent pas en même temps mais en temps partagé :



- Le Thread peut avoir 4 états différents
  - Etat nouveau : état initial après l'instanciation
  - Etat exécutable : lancé par la méthode start()
  - Etat en attente : n'exécute aucun traitement (wait(), sleep())
  - Etat mort : Thread qui est sorti de sa méthode run()

Les Threads

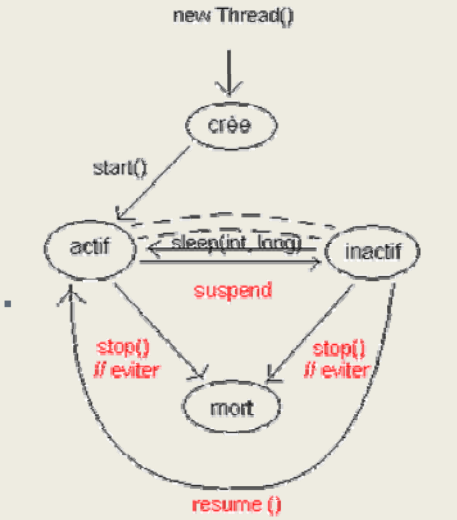


figure 8 : fonctionnement des threads. Les commandes les commandes en rouge sont depreciée car dangereuses donc à eviter.

Les Threads

- En général une application de type serveur se charge d'écouter sur un port les demandes de connexion adressées par les clients.
- A chaque nouvelle connexion il crée un nouveau processus (*ou processus léger*) qui se charge de cette connexion.
  - la méthode *accept()* renvoi une instance de *Socket* avec un port (virtuel) différent du socket serveur.
- Le programme principal retourne à l'écoute de nouvelles connexions sur le port "public".
  - Port 80 pour un serveur Web
  - Port 21 pour un serveur FTP

Les Threads

- Dans le cas où le serveur doit supporter plusieurs clients, il faut créer un *Thread* par client connecté:

```
TantQue (vrai) {
- accepter une connexion
  avec un nouveau client
- créer un Thread pour
  communiquer avec ce client
}
```


Les Threads

- Création et démarrage d'un thread :
- En Java, il existe deux techniques pour créer un thread.
  - Soit le Thread est dérivé de la classe java.lang.Thread
  - Soit l'interface java.lang.Runnable est implémentée.
- Les différences sont subtiles,
- Selon la technique choisie, il y a plus ou moins de facilité pour partager (ou non) des données entre différents threads.
- Le "bon" choix dépend du contexte d'application

Les Threads

- Dériver de la classe *java.lang.Thread*.
  - Il faut créer un objet de type *Thread*.
  - Il faut une méthode public *run()* sans paramètres ni retour, mais qui détient le code à exécuter par le *Thread*.
  - Il faut lancer le *Thread* et l'activer au niveau du système par la méthode *start()*, codée au niveau de la classe *Thread*.
  - A titre indicatif, il y a une méthode *sleep()* de la classe Thread qui est utilisée pour laisser le temps de visualiser les résultats.
- Avec cette technique, Il y a autant d'objets que de threads personnalisés.
- Voyons le fonctionnement d'un serveur multithread :

Les Threads

- L'instance de la classe *ServerSocket* permet au serveur de:
  - S'enregistrer auprès du système d'exploitation,
  - Écouter les demandes de connexion sur un port spécifique.
- Le serveur est un processus léger lancé par la méthode *start()*.
- Dans la méthode *run()* (appelée par *start()*), il y a une boucle infinie en attente de nouvelles connexions (méthode *accept()*).
- La nouvelle instance de Socket retournée est passée à une instance de la classe Traitement qui se charge alors de cette nouvelle connexion.
- Le serveur retourne alors à l'état d'attente.

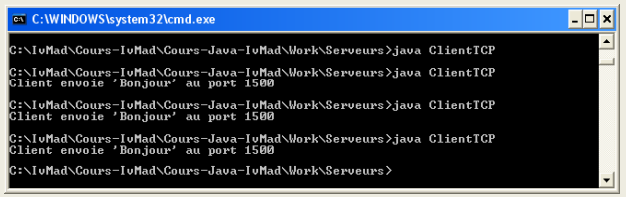
Serveur TCP Multithread

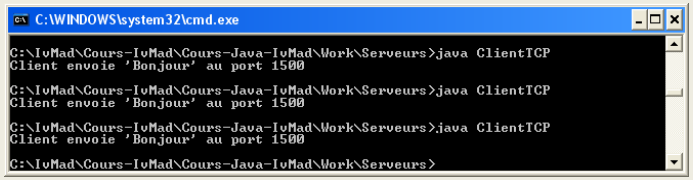
```
import java.net.*;  import java.io.*;
class ServeurConnecte extends Thread {
  ServerSocket reception;
  static final int port = 1500;
 public ServeurConnecte() {
    try {
      reception = new ServerSocket(port);
    }  catch(IOException e)  {
      System.exit(1);  }
    this.start();        // Lance la méthode 'run'
 }
 public void run()  {
    Socket sock;
    Traitement t;
    try {
       while(true)  {
         sock = reception.accept();
         t = new Traitement(sock);
    }
} catch(IOException e) { }
 }  }
```

```
class Traitement extends Thread {
   Socket sock;
   BufferedReader entree;
   public Traitement(Socket socket) {
      sock = socket;
      try {
        entree = new BufferedReader(new
                 InputStreamReader(sock.getInputStream()));
    } catch(IOException e) { }
    this.start();   // Lance la méthode 'run' du Thread
 }
 public void run()  {
    String text;
    try {
       text=entree.readLine();
 System.out.println("Le message du client: "+text+" sur le
port virtuel: "+sock.getPort()+" du port serveur:
"+sock.getLocalPort());
       sock.close();
   }  catch(IOException e) { }
 } }
public class TestServeurConnecte  {
   public static void main(String[] args)  {
      new ServeurConnecte();
   }  }
```


Serveur TCP Multithread

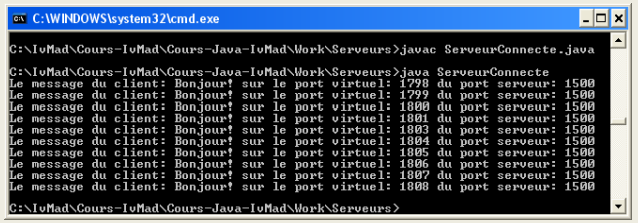
Connexions du client 1





Connexions du client 2

Etat du serveur



Serveur TCP Multithread