Interface homme-machine et langage Java

Henri Garreta, Faculté des Sciences (Luminy)
Cyril Pain-Barre, IUT d'Aix-Marseille (Aix)

http://henri.garreta.perso.luminy.univmed.fr/IHM_Java

Cours 2. L'interface graphique d'une application Java

Héritage

- pas d'héritage multiple
- pas de modification de l'accessibilité à l'occasion de l'héritage

```
public class Point {                          // la super-classe
    int x, y;
    ...
}

public class Pixel extends Point {            // la sous-classe
    Color couleur;
    ...
}
```

- à tout endroit où un `Point` est requis on pourra mettre un `Pixel`
- l'héritage est simple : une classe a *au plus* une super-classe
- chaque classe, sauf Object, a *une* super-classe
- il y a donc un arbre de toutes les classes ; la racine se nomme `Object`
- `Object` introduit les méthodes de `tous les objets` : `toString`, `clone`, `equals`...

Construction et destruction des objets

```
public class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}

public class Pixel extends Point {
    private Color couleur;
    public Pixel(int x, int y, Color c) {
        super(x, y);
        couleur = c;
    }
    ...
}
```

- `super(...)` doit être la *première* instruction d'un *constructeur*
- explicite ou implicite, il y a *toujours* un appel d'un constructeur de la super-classe
- exemple : si `Point` n'a pas de constructeur sans arguments, ceci ne passe pas :

```
...
public Pixel(int x, int y, Color c) {
    couleur = c;
}
...
```


Redéfinition des méthodes

```
public class Point {
    private int x, y;
    ...
    public String toString() {
        return "<" + x + "," + y + ">";
    }
    ...
}

public class Pixel extends Point {
    private Color couleur;
    ...
    public String toString() {
        return super.toString() + "-" + couleur;
    }
    ...
}
```

- emploi :

```
Point p;
...
System.out.println(p.toString());
```

si on a fait, par exemple, `p = new Point(10, 20);`
affichage : `<10,20>`

si on a fait `p = new Pixel(10, 20, Color.RED);`
affichage : `<10,20>-java.awt.Color[r=255,g=0,b=0]`

Méthode abstraite

- Méthode seulement « annoncée » dans une classe,
  elle doit obligatoirement être définie dans une sous-classe

```
abstract public class ObjetGraphique {

   public void seDessiner(arguments) {
        ... quoi mettre ici ? ...

    public abstract void seDessiner(arguments);

    autres membres de la classe ObjetGraphique
}

public class Rectangle extends ObjetGraphique {

    public void seDessiner(les mêmes arguments) {
        code effectif pour dessiner un rectangle
        ...
    }
    ...
    autres membres de la classe Rectangle
}
```


Interface

- classe entièrement faite de méthodes *abstraites* (c.-à-d. une liste de « promesses »)
- exemple : un objet `Repondeur` doit savoir « dire » *Oui* et *Non*

```
public interface Repondeur {
    void direOui();
    void direNon();
}
```

- dès qu'on a une telle interface, on peut programmer avec :

```
void traiterQuestion(Repondeur repondeur, autres arguments) {
    ...
    if ( condition )
        repondeur.direOui();
    else
        repondeur.direNon();
    ...
}
```


Interface

- classe entièrement faite de méthodes *abstraites* (c.-à-d. « promises »)
- exemple : un `Repondeur` doit savoir dire *Oui* et *Non*

```
public interface Repondeur {
    void direOui();
    void direNon();
}
```

- mais comment obtenir effectivement un objet `Repondeur` ?
- 1° définir une classe, appelée `implementation` de l'interface, par les moyens ordinaires...

```
public class RepAnglais implements Repondeur {
    public void direOui() {
        System.out.println("Yes");
    }
    public void direNon() {
        System.out.println("No");
    }
}
```

- ...puis l'instancier

```
Repondeur rep = new RepAnglais();
...
traiterQuestion(rep, autres arguments);
```


Interface

- classe entièrement faite de méthodes *abstraites* (c.-à-d. « promises »)
- exemple : un `Repondeur` doit savoir dire *Oui* et *Non*

```
public interface Repondeur {
    void direOui();
    void direNon();
}
```

- mais comment obtenir effectivement un objet `Repondeur` ?
- 2° définir une classe « dans la foulée » au moment de l'instanciation (classe *anonyme*)

```
Repondeur rep = new Repondeur() {
    public void direOui() {
        System.out.println("Yes");
    }
    public void direNon() {
        System.out.println("No");
    }
};
...

traiterQuestion(rep, autres arguments);
```

Bibliothèque *JFC* (*Java Foundation Classes*)

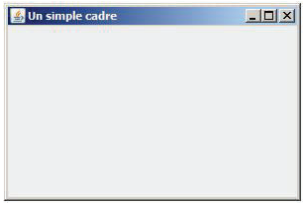
- *AWT* (*Abstract Windowing Toolkit*)
  - première version, rôle important dans le succès de Java
  - composants *lourds* (appariés avec des composants natifs)
- *Swing*
  - composants *légers*, « 100% pur Java »
  - plus nombreux, complexes et indépendants de la plate-forme
- en fait, on emploie :
  - les composants de *Swing*
  - certains éléments importants (*événements*, *gestionnaires de disposition*, etc.) de *AWT*
- attention aux noms des composants
  - *AWT* : `Frame`, `Button`, `Panel`
  - *Swing* : `JFrame`, `JButton`, `JPanel`

La plus petite application avec *IUG*

```
import javax.swing.JFrame;

public class Simple {

    public static void main(String[] args) {
        JFrame cadre = new JFrame("Un simple cadre");
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.setSize(300, 200);
        cadre.setVisible(true);
    }
}
```



- un cadre vide, mais vivant
- `JFrame` : objet nécessaire (et « suffisant ») de toute application
- attention, la case de fermeture ne termine pas l'application

Sous-classer

- créer sa propre classe cadre (sous-classe de `JFrame`)
- i.e. encapsuler la personnalisation du cadre dans le constructeur

```
public class Simple extends JFrame {

    public Simple() {
        super("Un cadre");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 200);
        setVisible(true);
    }

    public static void main(String[] args) {
        JFrame cadre = new Simple();
    }
}
```


Le haut de la hiérarchie

`Object`
- `Component`
  - visible à l'écran (*paint(...)*)
  - source d'événements (*addXXXListener(...)*)
  - `Container`
    - contient d'autres composants (*add(...)*)
    - a un gestionnaire de disposition (*setLayout(...)*)
    - `Window`
      - composant de niveau supérieur (*setVisible(...)*)
      - forment un arbre dynamique (*getOwner(...)*)
      - `Frame`
        - bord, bandeau de titre, barre de menus, etc.
        - souvent unique et permanente
      - `Dialog`
        - multiples et éphémères
        - peuvent être *modaux* ou *non modaux*

Programmation événementielle

Programmation « procédurale »

- le déroulement est contrôlé par une séquence d'instructions écrites
- le programmeur écrit la boucle principale

programme principal
initialisations
répéter
lire une commande
traiter une commande
jusqu'à la commande "finir"

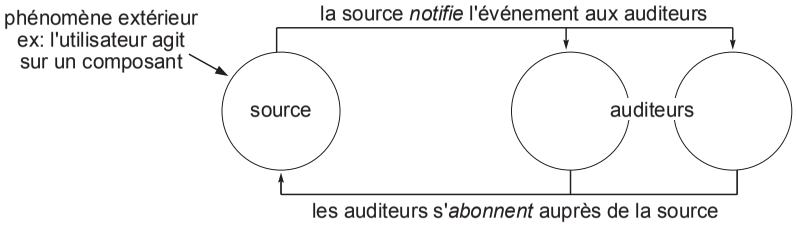
Programmation événementielle

- le déroulement est contrôlé par la survenue d'événements (dont les actions de l'utilisateur)
- pas de boucle principale (elle est enfouie dans la bibliothèque)

fonctions (réactions aux événements)

programme principal
initialisations
(guetter des événements)

Modèle événementiel de Java



- exemple : les événements souris sont notifiés par

```
void mousePressed(MouseEvent e)
void mouseReleased(MouseEvent e)
void mouseClicked(MouseEvent e)
void mouseEntered(MouseEvent e)
void mouseExited(MouseEvent e)
```

ces cinq méthodes constituent l'interface `MouseListener`
tout auditeur d'événements souris doit l'implémenter

- un tel auditeur s'abonne auprès de la source par

source`.addMouseListener(`*auditeur*`)`

Quelques catégories d'événements

- `MouseEvent` : actions *discrètes* sur la souris

`mousePressed`, `mouseReleased`, `mouseClicked`
`mouseEntered`, `mouseExited`

- `MouseMotionEvent` : actions *continues* sur la souris

`mouseMoved`, `mouseDragged`

- `FocusEvent` : gain et perte du clavier

`focusGained`, `focusLost`

- `KeyEvent` : actions sur le clavier

`keyPressed`, `keyReleased` événements de bas niveau
`keyTyped` événement élaboré

- `ActionEvent` : pression d'un bouton, choix dans un menu, etc.

`actionPerformed`

- `WindowEvent` : événements survenant sur une fenêtre

`windowActivated`, `windowIconified`, `windowOpened`
`windowDeactivated`, `windowDeiconified`, `windowClosed`
`windowClosing`

La suite se passe dans Eclipse...

- Exemple : détection des événements souris
  ⇒ voir projet *Exemples cours IHM*
- Exemple : tracé d'une ligne polygonale
  ⇒ voir projet *Exemples cours IHM*
- Exemple : gestionnaires de disposition
  ⇒ voir projet *Exemples cours IHM*