Programmation objet : java par la pratique ...

Laure DANTHONY

Avril 2001

Table des matières

1 Introduction

Les deux notions de base en java sont les suivantes :

- la notion de **classe** : ce sont les plus utilisées, le véritable cœur du programme. Chaque classe "dérive" obligatoirement de *sa* classe parente. Elle a accès à ses membres et aux membres de ses sur-classes (classe mère, grand-mère ...). En Java, une classe particulière, la classe *Object* constitue la racine de toutes les classes.
- la notion d'**interface** : elle est moins cruciale. C'est en quelque sorte une "boîte à outils" qui fournit en cas de besoin des services aux objets. Elles permettent l'héritage multiple, c'est à dire qu'une classe pourra grâce à elles "dépendre" de plusieurs interfaces. L'exemple le plus répandu d'interface est celle de *Listener* : elle permet d'associer à une action souris une action programme.

Les classes contiennent :

- des **attributs** : c'est à dire des objets personnels où l'on ne pourra accéder que par leur intermédiaire. Par exemple, une classe peut posséder un entier compteur qui ne peut s'incrémenter que par son intermédiaire.
- un (ou plusieurs) **constructeur(s)** ; par défaut, un contructeur vide, celui de la classe parente, est présent. Un constructeur de C_1 permet par exemple à partir d'une classe C_2 de créer une instance de C_1 propre à la classe C_2.
- des **méthodes**. Elles permettent d'agir sur les classes. Par exemple, une méthode peut permettre d'afficher les valeurs des attributs de la classe courante.

Attributs, constructeurs, méthodes d'une classe sont appelés ses **membres**. Ces membres peuvent avoir différents degrés d'accessibilité (public, privés, ...).
Voici un exemple (qui ne sert à rien) pour situer les différentes caractéristiques d'une classe :

```
public class Mouton{
    /* attributs */
    int matricule; // variable d'instance
    static int nombre; // variable de classe

    Mouton(){ // constructeur
        nombre = nombre +1;
        matricule = nombre;
    }

    public void afficherMatricule(){ // une méthode
        System.out.println("Mouton n°"+matricule);
    }
}
```

2 Un peu de syntaxe

Cette section peut être lue plus rapidement par les personnes familières du langage C.

En java, les types de base sont *short*, *int*, *double*, *long*, *float* *char*, *boolean*, le tableau de type de base est noté *int*[] par exemple. Noter qu'il n'y a pas de type *string* (seulement une classe *String*).

Instructions de base :

- Le test d'égalité $x == y$; l'affectation $x = 2$.
- Le test :

```
if (i==0) {
    i = i+1;
}
else {
    i=i-1;
}
```

  Evidemment le `else` est facultatif.

- Le "tant que" :

```
while (j<10)
{
    j++;
}
```

  Le `j++` est mis pour `j=j+1`.

- La boucle "pour" :

```
for (k=0,k<10,k++)
{
    System.out.println(k+" "); // imprime les chiffres de 0 à 9
                               // avec un espace entre les deux
}
```

  Remarquer le polymorphisme de l'impression ! et la syntaxe du commentaire C qui peut aussi être `/*comment*/`

Autres remarques :

- Dans les arguments d'une méthode `main` (méthode exécutée par un programme), il faut mettre `String[] args`, c'est à dire déclarer un tableau d'arguments. Cela permet par exemple de créer des programmes à arguments variables, et/ou qui utilisent les arguments passés par l'utilisateur sur la ligne de commande.
- La déclaration préalable des variables est *obligatoire*, même les variables de boucle.

3 Au travail !

3.1 Environnement de travail

On utilisera pour ce TD :

- JDK 1.2 de Sun : *http://www.javasoft.com/products/jdk/1.2/index.html*
- Un éditeur de texte ; de préférence *emacs* qui inclut un éditeur java.
- Un shell pour le travail en dehors de son éditeur (pour la compilation).
- Un browser pour consulter la doc API *http://www.java.sun.com/products/jdk/1.2/docs/api/index.html* (présente sur le disque à /java/api/index.html).

3.2 Utiliser l'aide en ligne

La fenêtre est découpée en trois frames :

- en haut à gauche : les "packages".
- en bas à gauche : l'index des classes.
- à droite : la description de la classe sélectionnée.

Le but en apprennant java n'est pas de connaître toutes ces classes par cœur, mais de savoir s'y retrouver et de savoir ce que l'on peut faire ou pas. La majeure partie du temps est passée à consulter la doc pour savoir de quelle classe on a besoin. Moralité : laisser Netscape allumé pendant toute la durée du td.

Il faut aussi savoir qu'il y a possibilité de construire automatiquement la documentation de ses propres classes, en mettant des commentaires aux bons endroits du code.

3.3 Ma première classe

▶ Recopiez ces quelques lignes dans un éditeur. Nommer le fichier HelloWorld.java (Avec la casse, c'est important !).

```
public class HelloWorld {
    public static void main (String[] argv){
        System.out.println("Hello World \n");
    }
}
```

▶ La compilation s'effectue dans un terminal avec `javac HelloWorld.java`. En sortie, cela donne un fichier `HelloWorld.class` qui s'exécute par `java HelloWorld`. Admirez votre première classe !

3.4 Ma première applet!

Une applet est un programme java un peu spécial car il s'exécute dans une page web. Merci de ne pas trop en mettre dans vos pages car elles ralentissent notablement leur exécution!

▶ Recopiez ces quelques lignes dans un éditeur. Nommez le fichier `HelloApp.java`:

```
import java.applet.*
import java.awt.*

public class HelloApp extends Applet{
    public void paint(Graphics g){
        g.drawString("Hello world !'',50,25);
    }
}
```

▶ De même, recopiez cette page html dans un fichier `applet.html`:

```
<HTML>
<HEAD>
<TITLE>Applet Hello World</TITLE>
</HEAD>

<BODY>
    <APPLET CODE ="HelloApp.class" WIDTH=200 HEIGHT=50>
    </APPLET>
</BODY>
</HTML>
```

▶ Compilez la classe *HelloApp* (`javac HelloApp.java`). Ouvrez dans votre navigateur favori la page `helloapp.html`. Observez le résultat!

Quelques remarques:

- `extends` est le mot-clef pour dire "dérive de". La classe *HelloApp* est une sous-classe de la classe *Applet*.
- une classe qui dérive de la classe applet n'a pas besoin de méthode `main`.
- le mot-clef `import` permet d'introduire le chemin des classes à charger qui sont nécessaires.

4 Quelques programmes plus élaborés

Il faut remarquer que l'on a accès à un champ (ou à une méthode) d'une classe *déjà créé et instanciée* par l'opérateur ".". Par exemple, pour `toto` instance de la classe *Toto* qui contient un champ `int cpt` et une méthode `int print()`, on a accès à `cpt` par `toto.cpt` et à la méthode `print()` par `toto.print()`.

4.1 Résolution d'une équation du second degré

Le but de l'exercice est la résolution d'une équation du second degré dont les coefficients sont fixés à l'intérieur de la classe *EquationTest* dont le code est le suivant :

```
/* Programme de résolution d'une équation du second degré */
/* Utilisation d'un objet de type Equation */
/* On veut résoudre 2x^2+5x+9=0 */

public class EquationTest {
    public void main (String[] args){
        Equation uneEquation; // Déclaration de l'objet uneEquation.
        uneEquation = newEquation(); // création de l'objet uneEquation.

        // instanciation des attributs de l'objet :
        uneEquation.coeffX2=2;
        uneEquation.coeffX1=5;
        uneEquation.coeffX0=9;
        uneEquation.résolution();

        //impression du résultat :
        System.out.println("Racine 1 "+uneEquation.racine1);
        System.out.println("Racine 2 "+uneEquation.racine2);
    }
}
```

Il faut donc créer une classe *Equation* qui doit avoir comme champs `coeffX2`, `coeffX1`, `coeffX0`, `racine1`, `racine2`.

▶ A vous de jouer ! Pour la compilation, faire attention. La classe *Equation* doit être compilée avant la classe *EquationTest*, et seule la classe *EquationTest* doit-être exécutée .

4.2 Encore un peu de maths !

L'objet de cette section est de créer une classe *Complexe* qui contient les outils nécessaires au calcul de la norme, à la multiplication par un scalaire, et au produit vectoriel entre deux vecteurs.

▶ Complétez donc le fichier suivant :

```
public class Complexe{
    double re ;
    ... ;

    /* Les constructeurs */
    Complexe(){ // constructeur vide
        re = 0.0;
        ...
    }
```

```
    Complexe(double x, double y){ // constructeur normal
        re = ... ;
        ... ;
    }

    Complexe(double x){ // constructeur pour complexe sans partie imaginaire
        ... ;
    }

    /* Les méthodes */
    public Complexe plus(Complexe z){ // addition du complexe courant
                                      // et de celui donné en paramètre
        ...;
    }

    public void plusModif(Complexe z){ // le résultat est cette fois mis
                                       // dans l'instance courante.
        ...;
    }

    public double norme(){ // norme du complexe courant.
        ...;
    }

    public void affiche(){ // imprimer im et re sur la sortie standard.
        ...;
    }

}
```

▶ Créez aussi une classe de tests (sur le modèle de *EquationTest*) afin de vérifier que vos méthodes fonctionnent .

4.3 Une remarque importante pour la suite

L'**encapsulation** est une notion fondamentale de la programmation objet. Dans la mesure du possible, il faut accéder aux attributs d'une classe par des méthodes et non pas par l'opérateur ".". Dans les classes déjà définies par Java, seules les méthodes sont accessibles. Pour accéder aux données, on a des méthodes du type `getItem`, `setColor`, ...

5 Encore plus fort !

Dans cette section nous allons voir les (immenses) possiblités graphiques de Java en créant un petit logiciel de dessin. Il faut distinguer deux phases :

- La phase graphique où les différentes zones graphiques sont allouées.
- La phase dynamique où l'on lie des opérations souris à des actions de dessin ou de sortie de l'éditeur.

5.1 L'application et les événements

Dans un premier temps, nous allons contruire une sorte d'éditeur de texte assez succint, dont les caractéristiques seront les suivantes :

- une barre de menu contiendra un menu contenant lui même les "item" suivants : Nouveau, Ouvrir, Sauver, Quitter. On pourra aussi quitter en cliquant sur la croix en haut à droite de l'application.
- L'éditeur contiendra une zone de texte. C'est le texte de cette zone que l'on pourra modifier.

Nous allons contruire une boîte (extension de la classe *JFrame*) qui contiendra le texte (classe *TextArea*).

▶ Complétez dans un premier temps le schéma suivant afin de préparer la barre de menu :

```
/* déclaration des importations de classe */
import java.awt.*;
import java.awt.event.*;
import java.io.*;


/* la classe principale */

public class DrawEditor extends Frame{
    /*Les attributs*/
    MenuBar mb;
    Menu menu ;
    MenuItem n,s,q,o;
    TextArea ta;


    /*Le constructeur*/
    public DrawEditor(String titre){

        super(titre);

        MenuBar mb = ... ;
        Menu menu = ... ;
        MenuItem n = ... ;
        MenuItem s = ... ;
        MenuItem q = ... ;
        MenuItem o = ... ;

        ... ; // Ajout à la barre d'outils
        setMenuBar(mb);

        ta = ... ;
        ta.setBackground(Color.gray);
```

```
    }

    // Le programme principal //

    public static void main(String args[])
    {
        DrawEditor f = new DrawEditor("Mon éditeur");
        f.setBackground(Color.lightGray);
        f.pack();
        f.setVisible(true);
    }
```

A chaque événement de clic sur un item, il s'agit maintenant d'associer une action. Pour cela, il faut associer un écouteur d'action (extension de la classe *ActionListener*) à chaque item, qui lancera une action (par exemple `quit()`) lorsque l'on cliquera dessus. Voici par exemple comment on associe l'action `quit` à l'item `quitter` :

```
q.addActionListener(new ActionQuit(this)); // ajout de l'écouteur
...

public void quit(){System.exit(0);} // ce qui sera appelé par
                                    // l'écouteur.


...
    class ActionQuit implements ActionListener // implémentation de l'écouteur.
    {
        DrawEditor te;
        ActionQuit(DrawEditor t)
        {
            te=t;
        }

        public void actionPerformed(ActionEvent e)
        {
            te.quit();
        }
    }
```

Quelques précisions : *ActionListener* est une interface. Il s'agit de l'"implémenter". La redéfinition de la méthode `actionPerformed` est donc indispensable. C'est la méthode qui sera lancée dès que l'écouteur d'action sera activé.

▶ Maintenant, à vous de jouer !

- Implémentez l'écouteur `ActionNew` et son action `newfile` (`new` est un mot clef !). On pourra regarder la méthode `setText` de la classe *TextArea*).
- Implémentez l'écouteur `ActionSave` et son action `save`. On pourra regarder les classes *FileDialog* et *FileWriter*.
- Implémentez l'écouteur `ActionOpen` et son action `open`. On utilisera *FileDialog* et *FileReader*, et on écrira un à un les caractères du fichier.

5.2 La gestion des erreurs

Lorsque l'on essaie de compiler, un message d'erreur nous indique que l'on a oublié de gérer les erreurs. Il s'agit par exemple de l'erreur `FileNotFoundException` qui est lancée lorsque l'on essaie d'ouvrir un fichier qui n'existe pas. Nous allons donc créer une classe *PbFichierErreur* qui rattrapera toutes les exceptions levées par des problèmes de fichier. Son constructeur prendra en argument une chaîne qui sera la chaîne à imprimer pour que l'utilisateur soit mis au courant de l'erreur. Plus tard, on pourra par exemple créer une boîte de dialogue qui imprime "joliment" ce message. Pour l'instant, on l'imprime simplement sur la sortie standard.

▶ C'est reparti : créer la classe *PbFichierErreur* étendant la classe *Exception* qui contient un constructeur vide et un constructeur contenant une chaîne `message`.

Pour récupérer les erreurs, il faut encapsuler les sources d'erreur par un `try`. La récupération se fait à l'aide d'un `catch` qui lui même doit effectuer `throw PbFichierErreur`. Ne pas oublier de déclarer pour la méthode en question qu'elle lance cette erreur, par exemple :
`public void open() throwsPbFichierErreur{ ...}`.

▶ Le faire !

5.3 Un peu de dessin !

Nous allons maintenant réaliser une classe *ZoneDessin* qui permettra de crééer un dessin à main levée. Elle étendra la classe *Canvas*, devra contenir les écouteurs *BoutonAppuye* (extension de *MouseAdapter*) et *BoutonMouvant* (extension de *MouseMotionAdapter*), qui s'occuperont respectivement de prendre les coordonnées courantes de la souris et de tracer une ligne rejoignant cette position à la position précédente (voir la méthode `newline` de la classe *Graphics*).

▶ Il s'agit donc de remplir les manques de la classe suivante :

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

public class ZoneDessin extends Canvas {
    ... ;

    public ZoneDessin(){
        super ();
        setSize (150,400);
        ... // ajout des écouteurs
    }

    class BoutonAppuye extends MouseAdapter
    {
        public void mousePressed(MouseEvent e){
        ... // récupération des coordonnées
        }
    }
```

```
    class BoutonMouvant extends MouseMotionAdapter{
        ... ;

        public void mouseDragged(MouseEvent e){
            ...

            Graphics g = getGraphics();

            ... ;
        }
    }
}
```

5.4 Recollons les morceaux

De nombreux outils en Java permettent de disposer des zones allouées à différentes tâches de façon automatique (c'est-à-dire sans que l'on ait à préciser leur position au pixel près). L'objet de cette section est de disposer une zone de dessin à côté d'une zone de texte, à l'intérieur de la classe *DrawEditor*. Pour celà, nous allons utiliser un *Layout*, c'est-à-dire un outil de disposition d'objets de type *Component*. Par chance, les classes *TextArea* et *Zone Dessin* sont toutes les deux des sous-(sous-)classes de *Component*! De nombreux *Layout* existent, ici on utilise un *GridBagLayout*:

▶ Ajoutez les lignes de codes suivantes à votre *DrawEditor*:

```
    ZoneDessin zd;

    /*Le conteneur*/
    GridBagLayout gbl;
    GridBagConstraints cons;

    void addComposantGridBag(GridBagLayout gb,
                             GridBagConstraints cons,
                             Component composant,
                             int gridx,
                             int gridy)
    {
        cons.gridx=gridx;
        cons.gridy=gridy;
        cons.gridwidth=1;
        cons.gridheight=1;
        gb.setConstraints(composant,cons);
        add(composant);
    }
...
        zd = new ZoneDessin();
        gbl = new GridBagLayout();
        setLayout(gbl);
        cons = new GridBagConstraints ();
```

```
addComposantGridBag (gbl, cons, ta, 0, 0);
addComposantGridBag (gbl, cons, zd, 1, 0);
```

▶ Regardez l'aide de *GridBag* pour voir à quoi correspondent les arguments du constructeur. Regardez aussi d'autres gestionnaires de mise en page (par exemple *BorderLayout*. Essayez !

5.5 Perspectives

Cette application est extensible à l'infini. On peut penser à ajouter des boutons (voir *Button*, c'est très facile). Evidemment, on doit leur associer des actions (par exemple tracer une ligne) à l'aide d'écouteurs. On peut aussi mettre un écouteur pour pouvoir fermer la fenêtre en cliquant sur la croix en haut à droite.

▶ Faites appel à votre imagination et à la doc !

6 Autres possibilités de java

Le langage java offre de nombreuses autres possibilités, dont :

- la gestion de processus légers (threads). Par exemple, avec ceux-ci, on peut faire promener un cercle dans une page web.
- la gestion de protocoles de communications. On peut ainsi facilement (!) créer des applications de type client-serveur, sans trop savoir quelles sont les caractéristiques de TCP-IP par exemple.

▶ A vous de jouer !