# Quelques aspects de la programmation Java

F. Barthélemy

28 février 2006

Ce document contient un certain nombre de programmes java complets illustrants des points vus dans le cours d'introduction à Java. Chacun de ces programmes illustre un point précis. Vous pouvez les compiler et les exécuter (ce que nous vous encourageons à faire). Et essayez de prédire le comportement de ces programmes. Vous pouvez copier ces programmes par copier-coller pour éviter de les retaper.

## 1 Compiler et exécuter un programme : aspects pratiques

Il existe deux sortes de programmes Java : les programmes classiques, qu'on exécute dans une fenêtre MS-DOS ou avec un environnement de programmation et les applets, qu'on exécute en visualisant une page web dans un navigateur.

Les premiers exemples que nous allons vous proposer ici sont des programmes classiques. Pour les compiler, il faut un compilateur. Soit vous en avez un, soit vous vous en procurez un. Nous vous conseillons le JDK de sun que vous pouvez obtenir gratuitement à l'URL `http://java.sun.com/downloads/`.

En ce qui concerne un environnement de programmation, nous vous conseillons BlueJ, que vous pouvez trouver sur `http://www.bluej.org/`. Cet outil est d'une puissance limitée, mais il est simple à utiliser et gratuit. Eclipse est plus puissant, mais plus complexe.

Supposons que vous n'ayez installé que le JDK, le compilateur brut sans environnement. Il faut d'abord taper votre programme. Vous allez pour cela créer un fichier texte qui a l'extension .java. Vous pouvez le créer avec `edit` ou `notepad`. Ce fichier peut comprendre plusieurs classes. Pami les classes, il y en a une qui doit contenir le programme principal, sous la forme d'une méthode de classe déclarée avec l'entête `public static void main(String[] argv)`.

Prenons un premier exemple :

```
class PremierExemple{
    public static void main(String[] argv){
        (new A()).saluer();
    }
}

class A{
    B b = new B();
    void saluer(){
        System.out.println("Bonjour, je suis une instance de A!");
        System.out.println("Je vais appeler une instance de B!");
        b.saluer();
    }
}
```

```
class B{
    void saluer(){
        System.out.println("Bonjour, je suis une instance de B!");
    }
}
```

On compile le fichier `PremierExemple.java` à l'aide de la commande de compilation `javac`.

```
javac PremierExemple.java
```

S'il n'y a pas d'erreur à la compilation, cela crée autant de fichiers que de classes dans le fichier .java, chacun de ces fichiers ayant pour nom le nom de la classe et pour extension .class. Ce sont les fichiers contenant le code compilé des classes. Dans notre exemple, la compilation crée les trois fichiers `PremierExemple.class`, `A.class` et `B.class`.

Pour exécuter le programme, il faut appeler l'interprète Java suivi du nom de la classe contenant la méthode `main`. Il faut bien donner le nom de la classe et non pas le nom du fichier (il ne faut pas préciser .class et encore moins .java).

```
java PremierExemple
```

Ce qui donne le résultat suivant :

```
> java PremierExemple
Bonjour, je suis une instance de A!
Je vais appeler une instance de B!
Bonjour, je suis une instance de B!
```

# 2 Les types référence

## 2.1 Deux variables de type objet

Voici un programme qui illustre le fait que les variables sont des pointeurs (références) vers des objets.

```
class Compteur{
    private int x = 0;
    void incremente(){
        x = x+1;
    }
    int valeur(){
        return x;
    }
}
class Reference1{
     public static void main(String[] argv){
         Compteur cpt = new Compteur();
         Compteur autre = cpt;
         System.out.print("Valeur de cpt: ");
         System.out.print(cpt.valeur());
         System.out.println();
         System.out.print("Valeur de autre: ");
```

```
        System.out.print(cpt.valeur());
        System.out.println();
        autre.incremente();
        System.out.println("Incrementation de autre");
        System.out.print("Valeur de cpt: ");
        System.out.print(cpt.valeur());
        System.out.println();
        System.out.print("Valeur de autre: ");
        System.out.print(cpt.valeur());
        System.out.println();
    }
}
```

Exécution :

```
> java Reference1
Valeur de cpt: 0
Valeur de autre: 0
Incrementation de autre
Valeur de cpt: 1
Valeur de autre: 1
```

## 2.2 Les paramètres sont des références

Comme les variables, les paramètres de types références sont des pointeurs.

```
class Compteur{
    private int x = 0;
    void incremente(){
        x = x+1;
    }
    int valeur(){
        return x;
    }
}
class Reference2{
    static void modifieSonParametre(Compteur c){
        c.incremente();
    }
    public static void main(String[] argv){
        Compteur cpt = new Compteur();
        System.out.print("Valeur de cpt: ");
        System.out.print(cpt.valeur());
        System.out.println();
        modifieSonParametre(cpt);
        System.out.println("Appel de modifieSonParametre");
        System.out.print("Valeur de cpt: ");
        System.out.print(cpt.valeur());
        System.out.println();
    }
}
```

Exécution :

```
> java Reference2
Valeur de cpt: 0
Appel de modifieSonParametre
Valeur de cpt: 1
```

### 2.3 Les tableaux sont des tableaux de références

```
class Reference3{
    public static void main(String[] argv){
        Compteur[] t = new Compteur[3];
        Compteur cpt = new Compteur();
        for (int i=0; i<3; i++){
            t[i]=cpt;
        }
        for (int i=0; i<3; i++){
            System.out.print("Valeur␣de␣t[" + i + "]␣:␣");
            System.out.println(t[i].valeur());
        }
        for (int i=0; i<3; i++){
            t[i].incremente();
        }
        for (int i=0; i<3; i++){
            System.out.print("Valeur␣de␣t[" + i + "]␣:␣");
            System.out.println(t[i].valeur());
        }
    }
}
```

Exécution :

```
> java Reference3
Valeur de t[0] : 0
Valeur de t[1] : 0
Valeur de t[2] : 0
Valeur de t[0] : 3
Valeur de t[1] : 3
Valeur de t[2] : 3
```

### 2.4 Les tableaux sont des références

```
class Reference4{
    public static void main(String[] argv){
        int[] t1 = new int[3];
        int[] t2;
        for (int i = 0; i<3; i++){
            t1[i]=i+1;
            System.out.println("t1[" +i+ "]=" + t1[i]);
```

```
        }
        System.out.println();
        t2 = t1;
        t2[1] = 17;
        for (int i = 0; i<3; i++){
            System.out.println("t1[" +i+ "]=" + t1[i]);
        }
    }
}
```

Exécution :

```
> java Reference4
t1[0]=1
t1[1]=2
t1[2]=3

t1[0]=1
t1[1]=17
t1[2]=3
```

## 2.5 Références mutuelles

Comme les objets sont des références, on peut parfaitement avoir des structures de données mutuellement récursives : un objet x contient un objet y dans une variable d'instance et y contient x dans une variable d'instance. Il faut prendre quelques précautions pour éviter les boucles dans les traitements.

```
class Personne{
    String nom;
    Personne conjoint;
    Personne(String leNom){
        nom = leNom;
    }
    void seMarier(Personne autre){
        this.conjoint = autre;
        if (autre.conjoint != this){
            autre.conjoint = this;
        }
    }
    void divorcer(){
        if (this.conjoint.conjoint == this){
            this.conjoint.conjoint = null;
        }
        conjoint = null;
    }
    void print(){
        if (conjoint == null){
            System.out.println(this.nom + " célibataire");
        }else{
            System.out.println(this.nom + " époux/épouse " +
                                this.conjoint.nom);
```

```
        }
    }
}
class Reference5{
    public static void main(String[] argv){
        Personne martine = new Personne("Martine");
        Personne jose = new Personne("Jose");
        martine.seMarier(jose);
        martine.print();
        jose.print();
    }
}
```

Exécution :

```
> java Reference5
Martine époux/épouse Jose
Jose époux/épouse Martine
```

## 3 Conversion de type explicite

La conversion explicite de type (cast) permet de changer la façon de voir un objet, sans rien changer à l'objet proprement dit. Les règles sont les suivantes : on peut toujours faire une conversion d'une sous-classe vers une super-classe. On peut toujours compiler une conversion d'une super-classe vers une sous-classe, mais il peut y avoir une erreur à l'exécution.

```
class Animal{
    String type = "Animal";
    void print(){
        System.out.println("Je suis un animal");
    }
}
class Mammifere extends Animal{
    String type = "Mammifere";
    void print(){
        System.out.println("Je suis un mammifere");
    }
}
class Chien extends Mammifere{
    String type = "Chien";
    void print(){
        System.out.println("Je suis un chien");
    }
}
class Rat extends Mammifere{
    String type = "Rat";
    void print(){
        System.out.println("Je suis un rat");
    }
}
class Cast1{
```

```
    public static void main(String[] argv){
        Mammifere m1 = new Mammifere();
        Mammifere m2 = new Rat();
        // Rat r1 = new Mammifere();  incorrect
        // Rat r2 = m2;               erreur compilation
        Rat r3 = (Rat) m2;
        Chien c = (Chien) m2;     // erreur d'execution
    }
}
```

Exécution :

```
> java Cast1
Exception in thread "main" java.lang.ClassCastException: Rat
        at Cast1.main(Cast1.java:32)
```

## 4 Surcharge

Voici un exemple de surcharge de méthodes : dans cet exemple, il y a plusieurs méthodes `m` dans la même classe.

```
class Surcharge{
    void m(){
        System.out.println("Je suis la meilleure.");
    }
    int m(int x){
        System.out.println("Je suis la seule vraie methode m de Surcharge.");
        return 2*x;
    }
    boolean m(String s){
        System.out.println("Je veux un parametre de type String.");
        return s.equals("Hello");
    }
    void m(int x, int y){
        System.out.println("Je suis une methode m parmi d'autres.");
    }
    public static void main(String[] argv){
        Surcharge obj = new Surcharge();
        obj.m();
        obj.m(1);
        obj.m("ToTo");
        obj.m(1,5);
    }
}
```

Exécution :

```
> java Surcharge
Je suis la meilleure.
Je suis la seule vraie methode m de Surcharge.
Je veux un parametre de type String.
```

```
Je suis une methode m parmi d'autres.
```

## 5 Redéfinition de méthode

On peut redéfinir une méthode dans une sous-classe. La nouvelle définition de la méthode annule et remplace la précédente définition.

```
class Animal{
    String type = "Animal";
    void print(){
        System.out.println("Je suis un animal");
    }
}
class Mammifere extends Animal{
    String type = "Mammifere";
    void print(){
        System.out.println("Je suis un mammifere");
    }
}
class Chien extends Mammifere{
    String type = "Chien";
    void print(){
        System.out.println("Je suis un chien");
    }
}
class Rat extends Mammifere{
    String type = "Rat";
    void print(){
        System.out.println("Je suis un rat");
    }
}
class Redefinition1{
    public static void main(String[] argv){
        Mammifere m1 = new Mammifere();
        Mammifere m2 = new Rat();
        m1.print();
        m2.print();
    }
}
```

Exécution :

```
> java Redefinition1
Je suis un mammifere
Je suis un rat
```

On peut redéfinir une variable dans une sous-classe, mais cette nouvelle définition ne remplace pas la définition précédente.

```
class Redefinition2{
    public static void main(String[] argv){
```

```
        Mammifere m1 = new Mammifere();
        Mammifere m2 = new Rat();
        System.out.println(m1.type);
        System.out.println(m2.type);
    }
}
```

La conséquence de cette différence entre variables et méthodes est particulièrement visible lorsqu'on fait une conversion de type.

```
class Redefinition3{
    public static void main(String[] argv){
        Rat r = new Rat();
        Mammifere m = r;
        if (r == m){
            System.out.println("r et m sont deux noms pour le même objet");
        }
        System.out.println(m.type);
        System.out.println(r.type);
        m.print();
        r.print();
    }
}
```

Proposition : ne jamais redéfinir une variable dans une sous-classe.

## 6 This et super

Le mot-clé `super` sert à appeler une ancienne définition masquée pour une redéfinition de méthode.

```
class Animal{
    String type = "Animal";
    void print(){
        System.out.println("Je suis un animal");
    }
}
class Mammifere extends Animal{
    String type = "Mammifere";
    void print(){
        super.print();
        System.out.println("Je suis aussi un mammifere");
    }
}
class Chien extends Mammifere{
    String type = "Chien";
    void print(){
        super.print();
        System.out.println("Je suis aussi un chien");
    }
}
class Super1{
```

```
    public static void main(String[] argv){
        Mammifere m = new Mammifere();
        m.print();
    }
}
```

```
class Animal{
    String type = "Animal";
    void print(){
        System.out.println("Je␣suis␣un␣animal");
    }
}
class Mammifere extends Animal{
    String type = "Mammifere";
    void print(){
        System.out.println("Je␣suis␣aussi␣un␣mammifere");
    }
    void teste(){
        System.out.println(super.type);
        System.out.println(this.type);
        System.out.println(type);
        this.print();
        print();
        super.print();
    }
}
class Chien extends Mammifere{
    String type = "Chien";
    void print(){
        System.out.println("Je␣suis␣aussi␣un␣chien");
    }
    void teste(){
        System.out.println(this.type);
        System.out.println(type);
        System.out.println(super.type);
        this.print();
        print();
        super.print();
    }
}
class Super2{
    public static void main(String[] argv){
        Mammifere m = new Mammifere();
        m.teste();
        System.out.println();
        m = new Rat();
        m.teste();
    }
}
```

# 7 Constructeurs

Le premier exemple montre une classe ayant plusieurs constructeurs (surcharge des constructeurs).

```
class A{
    int x;
    boolean y;
    A(){
        x=10;
        y=true;
    }
    A(int i){
        x=i;
        y=true;
    }
    A(boolean b){
        x=10;
        y=b;
    }
    A(int i, boolean b){
        x=i;
        y=b;
    }
    void print(){
        System.out.print("Valeur de x: ");
        System.out.println(x);
        System.out.print("Valeur de y: ");
        System.out.println(y);
        System.out.println("———————————————————");
    }
}
class Constructeur1{
     public static void main(String[] argv){
         (new A()).print();
         (new A(7)).print();
         (new A(false)).print();
         (new A(12,false)).print();
         (new A(17*5,(5<7)&('a'<'b'))).print();
     }
}
```

Exécution :

```
> java Constructeur1
Valeur de x: 10
Valeur de y: true
---------------------
Valeur de x: 7
Valeur de y: true
---------------------
Valeur de x: 10
```

```
Valeur de y: false
---------------------
Valeur de x: 12
Valeur de y: false
---------------------
Valeur de x: 85
Valeur de y: true
---------------------
```

Le second exemple montre qu'un constructeur peut appeler un autre constructeur de la même classe en utilisant la construction `this` (dans un sens absolument différent du `this` utilisé dans des méthodes pour désigner un objet).

```
class AA{
    int x;
    boolean y;
    AA(int i, boolean b){
        x=i;
        y=b;
    }
    AA(){
        this(10,true);
    }
    AA(int i){
        this(i,true);
    }
    AA(boolean b){
        this(10,b);
    }
    void print(){
        System.out.print("Valeur de x: ");
        System.out.println(x);
        System.out.print("Valeur de y: ");
        System.out.println(y);
        System.out.println("———————————————————");
    }
}
class Constructeur2{
     public static void main(String[] argv){
         (new AA()).print();
         (new AA(7)).print();
         (new AA(false)).print();
         (new AA(12,false)).print();
     }
}
```

Le troisième programme présente une erreur classique concernant les constructeurs. Ce programme ne compile pas. Voyez quelle erreur il provoque.

```
// une erreur classique: ce programme ne compile pas. Pourquoi?
class A{
    private int x;
```

```
    A(int i){
        x=i;
    }
}
class B extends A{
    int y;
}
```

Le dernier exemple d'utilisation de constructeurs est un exemple de l'appel à un constructeur de la superclasse avec le mot-clé `super`.

```
// l'erreur est réparée
class A{
    private int x;
    A(int i){
        x=i;
    }
    void print(){
        System.out.print("Valeur de x : ");
        System.out.println(x);
    }
}
class B extends A{
    int y;
    B(int a, int b){
        super(a); //pourquoi est-on obligé d'utiliser "super"?
        y=b;
    }
}
class Constructeur4{
    public static void main(String[] argv){
        A x = new A(2);
        A y = new B(4,5);
        y.print();
    }
}
```

# 8 Classe abstraite

```
abstract class Forme{
    abstract void translation(int dx, int dy);
    abstract String versChaine();
}
class Point extends Forme{
    int x;
    int y;
    Point(int a, int b){
        x=a;
        y=b;
    }
```

```
    void translation(int dx, int dy){
        x=x+dx;
        y=y+dy;
    }
    String versChaine(){
        return "("+x+","+y+")";
    }
}
class Carre extends Forme{
    Point hautgauche;
    Point basdroite;
    Carre(Point p1, Point p2){
        hautgauche = p1;
        basdroite = p2;
    }
    void translation(int dx, int dy){
        hautgauche.translation(dx,dy);
        basdroite.translation(dx,dy);
    }
    String versChaine(){
        return "Carre␣" + hautgauche.versChaine() + "␣" +
            basdroite.versChaine();
    }
}
class Cercle extends Forme{
    Point centre;
    int rayon;
    Cercle(Point p, int r){
        centre=p;
        rayon=r;
    }
    void translation(int dx, int dy){
        centre.translation(dx,dy);
    }
    String versChaine(){
        return "Cercle␣centre:" + centre.versChaine() + "␣rayon:" + rayon;
    }
}
class ClasseAbstraite{
    public static void main(String[] argv){
        Forme[] tab = new Forme[3];
        tab[0] = new Point(10,10);
        tab[1] = new Carre(new Point(8,9),new Point(15,16));
        tab[2] = new Cercle(new Point(2,3),7);
        for (int i=0;i<3;i++)
            tab[i].translation(20,20);
        for (int i=0;i<3;i++){
            System.out.println("forme␣" + i + ":␣" +tab[i].versChaine());
            System.out.println("————————————————————————");
        }
    }
```

```
}
```

Exécution :

```
> java ClasseAbstraite
forme 0: (30,30)
--------------------------
forme 1: Carre (28,29) (35,36)
--------------------------
forme 2: Cercle centre:(22,23) rayon:7
--------------------------
```

# 9 Interface

```
interface Comparable{
    boolean plusGrand(Object o);
    boolean memeTaille(Object o);
}
class Entier implements Comparable{
    int valeur;
    Entier(int x){
        valeur = x;
    }
    public boolean plusGrand(Object o){
        return valeur > ((Entier) o).valeur;
    }
    public boolean memeTaille(Object o){
        return valeur == ((Entier) o).valeur;
    }
}
class Rectangle implements Comparable{
    int largeur, hauteur;
    Rectangle(int x, int y){
        largeur = x;
        hauteur = y;
    }
    public boolean plusGrand(Object o){
        Rectangle r = (Rectangle) o;
        return largeur * hauteur > r.largeur * r.hauteur;
    }
    public boolean memeTaille(Object o){
        Rectangle r = (Rectangle) o;
        return largeur * hauteur == r.largeur * r.hauteur;
    }
}
class Interface1{
    public static void main(String[] argv){
        Entier e1 = new Entier(12);
        Entier e2 = new Entier(8);
        System.out.println(e1.plusGrand(e2));
```

```
        Rectangle r = new Rectangle(10,6);
        System.out.println(r.memeTaille(r));
        Comparable c;
        c = e1;
        c = r;
    }
}
```

Exécution :

```
> java Interface1
true
true
```

On peut utiliser le nouveau type pour faire de la programmation abstraite. Par exemple, une méthode peut vérifier si un tableau d'éléments comparables est trié.

```
class Interface2{
    static boolean estTrie(Comparable[] ef){
        for (int i = 0; i< ef.length -1; i++){
            if (ef[i].plusGrand(ef[i+1])){
                return false;
            }
        }
        return true;
    }
    public static void main(String[] argv){
        Entier e1 = new Entier(2);
        Entier e2 = new Entier(8);
        Entier e3 = new Entier(13);
        Rectangle r = new Rectangle(10,6);
        Comparable[] t1 = {e1, e2, e3};
        Comparable[] t2 = {e1, e3, e2};
        Comparable[] t3 = {e1, r, e2};
        System.out.println(estTrie(t1));
        System.out.println(estTrie(t2));
        System.out.println(estTrie(t3));
    }
}
```

Exécution :

```
> java Interface2
true
false
Exception in thread "main" java.lang.ClassCastException: Rectangle
        at Entier.plusGrand(Interface1.java:11)
        at Interface2.estTrie(Interface2.java:4)
        at Interface2.main(Interface2.java:20)
```