PROGRAMMATION PAR OBJETS

Java
Environnement et constructions spécifiques

Walter Rudametkin
Maître de Conférences
Bureau F011
Walter.Rudametkin@polytech-lille.fr

Java (Sun 1995)

- Entre Smalltalk et C++
- C++
  - Syntaxe familière à la C/C++
  - fortement typé
  - gestion des exceptions
- Smalltalk
  - "tout objet"
  - machine virtuelle
  - gestion automatique de la mémoire: garbage collector (pas de pointeurs explicites)
- Portable
  - machine virtuelle (bytecode)
  - standards (arithmétique IEEE 754, Caractères 16 bits Unicode)
  - même au niveau graphique (java2D, java.awt et javax.swing)
- Intègre le réseau
  - Applets (clients WEB) / Standalone applications (interprète java)
  - Code mobile (internet), chargement dynamique de code
- Nombreuses bibliothèques de classes (JDK : Java Development Kit)
  - java.util : SD (listes, piles, itérateurs...) ...
  - java.sql : accès aux BD (jdbc)
  - java.awt, javax.swing : graphique et interface
  - java.rmi : réseaux et objets distribués
- Free:
  http://java.sun.com = http://www.oracle.com/technetwork/java
  - JDK = Java SE (Standard Edition) Development Kit (V5 ou 6)

Applets

Programme Java, non autonome, destiné à être invoqué dans des documents HTML:

- sous un navigateur intégrant un interprète Java (JVM) (sous la forme d'un plugin de navigateur)
- ou un visualisateur d'applets (outil appletviewer du JDK)

```
//fichier Salut.java a compiler:
//javac Salut.java => Salut.class
import java.applet.*;
import java.awt.*;
public class Salut extends Applet {
  public void paint(Graphics g) {
        g.drawString("Salut!",20,20);
  }
}
<HTML>
<!-- fichier salut.html sur la meme machine -->
<APPLET CODE="Salut.class" WIDTH=200 HEIGHT=50> </APPLET>
</HTML>
```


Applications autonomes

- Une classe, dite principale et `public`, introduit une méthode "`main`" particulière qui détermine une application exécutable par la commande java

```
// fichier HelloWorld.java
public class HelloWorld {
  public static void main(String[] args) {
      System.out.println("Hello World!");
  }
}

bash>javac HelloWorld.java
bash>java HelloWorld
```

- Fichiers source
  - Un fichier peut contenir plusieurs classes, javac générera autant de .class
  - Mais un fichier ne peut contenir qu'une classe public et doit porter son nom
  - Règle : un fichier par classe (compilable séparément).

Ligne de commande

Un seul paramètre : tableau d'objets `String`

- n'incluant pas le nom du programme,
- sa taille (équivalent de `argc`) peut être obtenue comme pour tout tableau par son champ `length`

```
public class Echo {
  public static void main(String[] argv) {
  for (int i=0; i<argv.length; i++)
    System.out.print(argv[i]+" ");
    System.out.println();
  }
}
bash>java Echo bonjour le monde
bash>bonjour le monde
```


Applications autonomes

Exemple depuis java 5.0

```
public class Echo {
  public static void main(String[] argv) {
    //sequence de valeurs : « for each »
    for (String chaine : argv){
      // sortie formattée a la C :
      System.out.printf("%s ", chaine);
    }
    System.out.print("\n");
  }
}
```


Entrées/sorties standards

- **Les e/s (fichiers, "standards") sont définies par une hiérarchie de Streams (flots) dans le package java.io**

```
InputStream //lecture d'octets dont System.in
OutputStream
 FilterOutputStream
      PrintStream //sortie standard System.out
```

- **Les flots chargés des e/s «standards » sont fournis dans 3 « variables statiques » de la classe System:**

```
public class System {...
// variables de classe
 public static PrintStream err;
 public static InputStream in;
 public static PrintStream out;
...}
```


Sortie standard

- **System.out.print(..) et println(...)**
  - Ces méthodes sont *surchargées* pour chaque type de base (char, int, double, boolean...)
  - pour un objet, invoque sa méthode `toString()` qui doit fournir une `String` de représentation textuelle de l'objet
  - `toString()` est fournie par défaut dans `Object`. Il suffit de la redéfinir.
  - Exemple

```
class And {...
 public String toString() {
  return (
       "e1= "
    + String.valueOf(e1) // transformation explicite
    + " e2= "
    + e2 // transformation automatique
    + " s= "
    + s;
    );
 }
...
}
```

Java 5.0 : sortie standard formattée

- **System.out.printf(String format, Object… args)**
  - printf « à la C »
  - format : %d, %f, %s, ...
  - Remarque :
    dans le cas d 'objet utiliser %s => appel automatique à `toString()`
  - Exemple :

```
And a1 = new And(), a2 = new And();
// utilisation de a1 et a2
// affichage :
System.out.printf("%d ands a1: %s a2: %s\n", 2, a1, a2);
```


Entrée standard (depuis Java 5.0)

- **System.in**
  flot de bytes à "scanner" en l'enrobant ("wrapper") dans la classe `Scanner`

```
public class java.util.Scanner {
 public String next()
 public int nextInt()
 public double nextDouble()
 public String nextLine()
...}
```

- Exemple :

```
import java.util.*;
Scanner in = new Scanner(System.in);
System.out.printf("entrer 1 int, 1 double, une chaine,
                   et le reste : \n");
int i = in.nextInt();
double d = in.nextDouble();
String s = in.next();
String reste = in.nextLine();
System.out.printf("i=%d\n d=%f\n s=%s\n reste=%s\n",i,d,s,reste);
```


`static` : variables et méthodes de classe

- Déclarées au niveau de la classe par le « modifier » `static`
- Permet de définir une ressource
  - attachée à la classe
  - en exemplaire unique
  - commune à toutes ses instances (accessibles directement)
  - et même "globale" puisque la classe l'est!
    Accessible en désignant la classe, comme c'est le cas de :
    `System.out, System.in, ...`
- La déclaration **`final`** la rend en plus non modifiable et permet donc de déclarer des constantes

`static` : variables et méthodes de classe

```
class Circle {
    static final double PI = 3.14159265;
    // variables d'instance
     double rayon;
    // methodes d'instance
    double circonference() {
       return 2*PI*rayon; // ou 2*Circle.PI*rayon
       // final => constante => calculee statiquement
    }
}
```

- Méthodes de classes: exemples du langage (package java.lang)

```
public class System
  public static void exit(int status)
  public static Properties getProperties()

public class Math {
  public static double min(double a, double b)
  public static double sin(double a)
```

Syntaxe et éléments de base

- Commentaires

```
/* ceci est
   un commentaire sur plusieurs lignes*/
// ceci est un commentaire ligne
```

- Identificateurs
  - variables, classes, méthodes, packages
  - syntaxe

```
identificateur = initiale suivant*
initiale = “a“|...|“z“|“A“...“Z“|“$“|“_“
suivant = initiale |“0“...“9“|unicode > 00C0
```

  - pas de limitation de longueur, tout caractère significatif (minuscules et majuscules)
  - conventions :
    - ne pas utiliser $ et _ (librairies C)
    - `ceciEstUnIdentificateur`
    - `NomDeClasse`
    - `CONSTANTE`

Mots réservés

```
abstract boolean break byte byvalue case cast
catch char class const continue default do
double else extends false final finally float
for future generic goto if implements import
inner instanceof int interface long native new
null operator outer package private protected
public return short static super switch
synchronized this thread throw throws
transient true try void volatile while
```


Variables

- Déclaration

```
<modifier> type variable_declarator <“,“ variable_declarator> “;“
variable_declarator = identifier <“[“ “]“> [“=“ initializer]
modifier = “abstract“ | “public“ | “private“ | “protected“ |
  “static“ | “final“ ...
type = type_primitif | classe | interface
```

- tableaux: `type identifier[]` ou `type[] identifier`
- `modifier` : concerne la programmation des classes
- 2 catégories de variables :
  - de type d’objets (classes et interfaces) : contiennent des références
  - de type primitif : contiennent des valeurs

Types primitifs

- C (C++) + `boolean` et `byte`
- de taille constante quelque-soit la machine
- gérés par valeur, ce ne sont pas des objets, mais «enrobables» par les Wrapper classes
- les `boolean` ne sont pas des entiers
- les `char` sont codés sur deux octets Unicode compatible ASCII. les caractères spéciaux sont notés (comme en C) `\n \t \b \r \f \\ \’ \“`

Types primitifs

Type	*Valeurs*	*Init*	*Taille*	*Wrapper class*
boolean	true false	false	1 bit	Boolean
char	Unicode	'\u0000'	16 bits	Character
byte	entier signé	0	8 bits	Byte
short	entier signé	0	16 bits	Short
int	entier signé	0	32 bits	Integer
long	entier signé	0L|0l	64 bits	Long
float	IEEE 754 (0.5E-3)	0.0F|0.0f	32 bits	Float
double	IEEE 754	0.0D|0.0d	64 bits	Double

Wrapper Classes

- « Pont » entre valeurs primitives et objets
- Permet de considérer une valeur de type primitif comme un objet quand cela est requis (cf. collections d'objets)

```
Integer nObject = new Integer(2); // wrapping
int n = nObject.intValue();// unwrapping
```

- Offre des utilitaires (`static`) comme le parsing `String ->` valeur inverse de `String.valueOf(…)` (cf. entrée standard avant 5.0, paramètres du `main`, saisies de champs texte dans les interfaces)

```
public class Plus {
    public static void main(String[] args) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        System.out.printf("x+y=%.2f\n", x+y);
}}
```


Initialisation d'objets : constructeur

```
new <Classe>()
```

- `<Classe>()` est appelé « constructeur par défaut » et initialise les variables d'instances de l'objet:
  - aux valeurs déclarées, si elles existent
  - par défaut sinon (et en standard) :
    - valeur d'init pour les types primitifs (cf. tableau des types)
    - `null` pour les types d'objets
- Il est possible de le redéfinir, de le surcharger en le paramétrant:

```
new <Classe>([<parametres>])
```

- C'est le premier traitement exécuté par l'instance

Constructeurs

- Un constructeur permet de réaliser tout traitement à l'initialisation

```
class And {
 …
 And(boolean in1, boolean in2) {
      e1=in1;
      e2=in2;
      run(); // this.run();
 }
}
```

- Pas de destructeur (contrairement à C++) :
  - automatique par garbage-collector (objets non-référencés).
  - Il existe cependant un protocole de "finalisation" utilisable dans des cas particuliers (libération de ressources systèmes...) : méthode `finalize()`.

Construction d'objets composites

- Construction d'objets par composition d'autres objets

```
class Rectangle {
 Point origin, corner;
 …
}
```

- Initialisation par défaut des objets : null, d'où :

```
class Rectangle {
 …
 Rectangle(Point p1, Point p2) {
       origin=p1;
       corner=p2;
  }
 Rectangle(double x1, double y1, double x2, double y2) {
       origin = new Point(x1,y1);
       corner = new Point(x2,y2);
  }
}
```


Tableaux

- Les tableaux sont des objets :
  - créés dynamiquement (avec leur length) par instanciation :
    `new <type des elements>[<length>]`
  - libérés automatiquement (gc=garbage collector)
  - manipulés par référence : variables tableaux et passage en paramètre
  - compatibles avec le type Object dont les méthodes sont applicables.
- Mais syntaxe spécifique (à la C):
  création avec initialisation par : {} , accès par : [], …
- Type des éléments : types primitifs (homogènes) ou classes (tableaux polymorphes).
- Tableaux multidimensionnels = vrais tableaux de tableaux

Tableaux

- Exemples

```
int[] t1= new int[10];

// declaration avec initialisation:
int[] t2= {1,2,3,4,5};

// affectation de variables tableaux
t1=t2; // t1 et t2 sont des variables

int[][] matrice = new int[50][100];

int[][] matrice = new int[][100]; //impossible!
```


Tableaux

```
public class test {
  static void uns(int tab[]) { // passage en parametre
    for (int i=0;i<tab.length;i++) tab[i]=1;
  }
  public static void main(String args[]) {
    int tabtab[][]=new int[3][]; //tableau de 3 tableaux d'int
    tabtab[0]=new int[10]; // de tailles quelconques...
    tabtab[1]=new int[20];
    tabtab[2]=new int[30];
    uns(tabtab[0]); uns(tabtab[1]); uns(tabtab[2]);
    for (int i=0;i<tabtab.length;i++) {
      for (int j=0;j<tabtab[i].length;j++){
        System.out.print(tabtab[i][j]);
      }
      System.out.print("\n");
}}}

//for en 5.0
  for (int[] ligne : tabtab) { // ligne : variable tableau
    for (int x : ligne)
      System.out.print(x);
```

Chaînes de caractères

- **Ce sont des objets à part entière**
  - instances de la classe String
  - mais admettent une forme littérale :
    `String s = "deux\nlignes";`
- **Deux classes principales**
  - **String =** objets chaînes de taille constante
  - **StringBuffer** = objets chaînes de taille variable

Chaînes de caractères

- **String** : quelques opérations
  - `opérateur + (String)`
  - `les méthodes valueOf(...)`
  - `int length()`
  - `int compareTo(String) (équivalent de strcmp)`
  - `boolean equals(Object)`
  - `char charAt(int) throws StringIndexOutOfBoundsException`
  - `String substring(int,int)`
    `throws StringIndexOutOfBoundsException`
- **StringBuffer : chaine modifiables, en contenu et en taille** :
  - `StringBuffer append(String)`
  - `StringBuffer insert(int,String)`
    `throws StringIndexOutOfBoundsException`
  - `void setCharAt(int, char)`
    `throws StringIndexOutOfBoundsException`

Expressions et structures de contrôle

- Pour l'essentiel, très semblables à C (C++).
- L'appel de fonction est remplacé par l'envoi de message:
  - c'est une instruction si la méthode est de type void
  - une expression sinon.
- **Opérateurs**
  - en moins: `*, &, ->, sizeof` (inutiles)
  - en plus: `instanceof` et `+` de concaténation de chaînes
  - les opérateurs logiques procèdent sur le type `boolean`
  - mêmes règles de priorité et d'associativité
- **Structures de contrôle**
  `if/else, while, do/while, switch, for, break`
  - les prédicats sont de type `boolean`
  - `for (int i=0;i<n;i++) // indice local a la boucle`
  - depuis Java 5.0 le « `for each` »permet d'itérer sur toute séquence de valeurs « itérable », en particulier tableaux et collections .