Programmation Par Objets

Java

Environnement et constructions spécifiques

Java

- 1995 (Sun microsystems, repris par Oracle), entre C++ et Smalltalk
- C++
  - syntaxe familière « à la C », typé statiquement, jeu de types primitifs (int, double, …)
    gestion des exceptions
- Smalltalk
  - "tout objet" (ou presque, hors types primitifs)
  - machine virtuelle (JVM = Java Virtual Machine)
  - 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 (côté client), Servlets (côté serveur), code mobile, objets répartis
- Nombreuses bibliothèques de classes (JDK : Java Development Kit) : SD, accès BD, graphique virtuel, réseau, ...
- Free:
  http://www.oracle.com/technetwork/java
  JDK = Java SE (Standard Edition) Development Kit

Applets/applications autonomes

Applet : code destiné à être invoqué dans des documents HTML sous un navigateur intégrant une JVM

```
//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 ») introduit une méthode "`main`"

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

system> javac HelloWorld.java // => HelloWorld.class
system> java HelloWorld // sans ".class"
```

- un fichier source peut contenir plusieurs classes, `javac` générera autant de fichiers `.class` (bytecode) correspondants
- mais 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 de `String`

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

```
public class echo {
   public static void main(String[] args) {
      for(int i=0; i<args.length; i++)
         System.out.print(args[i] + " ");
      System.out.print("\n");
   }
}
system> java echo bonjour le monde
system> bonjour le monde
```


Au passage (depuis Java 5)

`"for each"` et `printf` formaté "à la C"

```
public class echo {
   public static void main(String[] args) {
      for (String chaine : args) // for each
         System.out.printf("%s ", chaine);
      System.out.print("\n");
   }
}
```

Entrées/sorties standards

- **Les e/s (fichiers) sont définies par une hiérarchie de `Streams` (flots) dans le package `java.io`**
- **Les flots chargés des e/s standards sont fournis dans 3 champs `static` de la classe `System` :**

```
public class System {...
   public static PrintStream err; // sortie texte
   public static InputStream in; // entrée binaire
   public static PrintStream out; // sortie texte
...}
=>
System.in // stdin de C façon objet
System.out // stdout de C
System.err // stderr de C
```


Sortie standard

- **`System.out.print` et `System.out.println`**
  - ces méthodes sont *surchargées* pour chaque type primitif (`int`, `double`, ...)
  - pour un objet elles invoquent automatiquement sa méthode `toString()`
- Exemple

```
public class And { …
   public String toString() {
      return
         ''[e1=''
         + String.valueOf(e1)
         + '' e2=''
         + e2 // transfo automatique => String.valueOf(e2)
         + '' s=''
         + s
         + '']'';
}}
```

Depuis Java 5 : sortie formatée

- **`System.out.printf(String format, args)`**
  - « à la C » : formats : `%d, %f, %s` ...
  - pour l'affichage d'un objet utiliser `%s`
    => appel automatique à sa méthode `toString()`

- Exemple

```
And a1 = new And(), a2 = new And();
// affichage apres manipulation de a1 et a2 :
System.out.printf("Etats:\n a1=%s \n a2=%s", a1, a2);
// resultat
Etats:
 a1=[e1=true e2=false s=false]
 a2=[e1=true e2=true s=true]
```


Entrée standard

- **System.in** = entrée binaire (bytes) à la base (`InputStream`)
  depuis Java 5 : facilités pour la "scanner" de manière formatée (à la `''scanf''` de C) grâce à la classe de *wrapping* `Scanner du package java.util` :

```
public class java.util.Scanner {
   public String next() // scanf(%s) de C
   public int nextInt() // scanf(%d) de C
   public double nextDouble() // scanf(%f) de C
   public String nextLine() // gets de C
...}
```

- Exemple

```
import java.util.Scanner;
Scanner scan = new Scanner(System.in);
System.out.printf("entrer un int, une chaine et le reste");
int i = scan.nextInt();
String s = scan.next();
String reste = scan.nextLine();
```

Variables et types

- 2 catégories de variables (exclusives):
  - de types *primitifs* : contiennent des *valeurs*
  - de types d'objets (*classes* et *interfaces*) : contiennent des *références*

- Types primitifs
  - comme en C (C++) + `boolean` et `byte`
  - de taille constante quelque-soit la machine
  - gérés par valeur, ce ne sont pas des objets, mais «enrobables» par des classes *Wrapper* en correspondance
  - `boolean` est un vrai type (ce ne sont pas des entiers)
  - les `char` sont codés sur deux octets Unicode compatible ASCII. Les caractères spéciaux sont les mêmes qu'en C: `\n \t \b` …

Table de référence des 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

Variables de types d'objets

- Contiennent des références d'objets, initialement `null` (« pas d'objet »)
- Création d'objet dynamique par instanciation :

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

- `<Classe>()` est appelé « constructeur »
- Par défaut il 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. table de référence précédente)
    - `null` pour les variables de types d'objets
- Il est possible de le redéfinir, de le surcharger en le paramétrant:

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

- C'est le premier traitement exécuté par l'instance à sa création.

Constructeurs

- à programmer dans la classe et doivent porter son nom
- n'ont pas de résultats: traitement d'initialisation par l'instance

- Exemple

```
class And {...
   And(boolean in1, boolean in2) { // un constructeur
        e1=in1;
        e2=in2;
        run(); // <=> this.run()
   }}
// utilisation
And a = new And(true,false);
```

- En Java il n'y pas de destructeur (contrairement à C++) :
  - automatique par garbage-collector (pas de ''free'')
  - Il existe cependant un protocole de "finalisation" utilisable dans des cas particuliers (libération de ressources systèmes...) : `finalize()`

Construction d'objets composites

- Construction d'objets par composition d'autres objets

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

- Initialisation par défaut de `origin,corner` à `null`, d'où :

```
class Rectangle {...
 Rectangle(double x1, double y1, double x2, double y2) {
       origin = new Point(x1,y1);
       corner = new Point(x2,y2);
 }
 Rectangle(Point p1, Point p2) { // surcharge de constructeurs
       origin=p1;
       corner=p2;
 }
}
```


Tableaux

- En Java les tableaux sont des objets :
  - créés dynamiquement (avec leur `length`) par instanciation :
    `new <type des elements>[<length>]`
  - libérés automatiquement (garbage collector)
  - manipulés par référence : variables tableaux et passage en paramètre
  - compatibles avec la classe `Object` dont les méthodes sont applicables.
- Mais syntaxe spécifique (« à la C », pas de classe Tableau explicite): accès par : `[]`, création avec initialisation par : `{}` , …
- Type des éléments : types primitifs (tableaux homogènes) ou objets (potentiellement hétérogènes).
- 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");
}}}
// façon « for each »
   for (int[] ligne : tabtab) { // ligne : variable tableau
    for (int x : ligne)
       System.out.print(x);
```

Tableaux d'objets

- Exemple

```
Rectangle[] t = new Rectangle[10];
```

- Mais attention...
  - crée l'objet tableau mais pas les objets `Rectangle`
  - tableau `t` = 10 **variables** indicées pour contenir des `Rectangle` quelconques
  - initialisées à `null` comme pour toute variable de type objet
  - les `Rectangle` n'ont aucune raison d'être créés à ce niveau

```
t[0] = new Rectangle(10,20,30,40);
Rectangle r; // créé par ailleurs
t[1] = r;
...
```


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`**
  - opérateur `'+'` de concaténation
  - les méthodes `String.valueOf(…)` de transformation valeurs de type primitif vers `String`
  - `int length()`
  - `int compareTo(String) // strcmp de C`
  - `boolean equals(Object) // <=> compareTo(String) == 0`
  - `char charAt(int) throws StringIndexOutOfBoundsException`
  - `String substring(int,int)`
    `throws StringIndexOutOfBoundsException`
- **`StringBuffer` : chaines modifiables, en contenu et en taille** :
  - `StringBuffer append(String)`
  - `StringBuffer insert(int,String)`
    `throws StringIndexOutOfBoundsException`
  - `void setCharAt(int, char)`
    `throws StringIndexOutOfBoundsException`

`static` = variables et méthodes de classe

- Permet de définir une ressource
  - attachée à une classe
  - en exemplaire unique (comme les classes le sont)
  - commune à toutes ses instances (accessibles directement)
  - et même accessible globalement si la classe l'est et par sa désignation, 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` = attaché à une classe

- Exemple de constante

```
public class Circle {...
   public 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 de l'exterieur
        // 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 de base

- Syntaxe « à la C »
- Commentaires

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

- Identificateurs de classes, variables, méthodes, …

```
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 '_' en tête (réservés aux librairies C sous-jacentes)
    - `ceciEstUnIdentificateur` de variable, méthode, paramètre
    - `NomDeClasse` (majuscule en tête)
    - `CONSTANTE` (tout en majuscule)

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`
  - « `for each` » depuis Java 5 permet d'itérer sur toute séquence « itérable », en particulier les tableaux et les collections.

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
```