Java

Ce document est aussi disponible en version électronique à partir de la page :
`http://www.site.uottawa.ca/~anquetil/CSI-4516/csi-4516.html`

Introduction

Ce cours est une toute petite introduction à Java. Il suppose que vous connaissez au moins le langage C et peut-être C++. Il suppose aussi que vous êtes familiers avec les notions du modèle à objets.

Pour plus de détails, voir les différentes documentation proposées à la fin de ce cours.

Compilation, exécution, un exemple

Lisez la page web du département pour savoir comment exécuter un et compiler un programme Java sur les machines Unix.

Sur les machines Solaris du département, vous pouvez utiliser le compilateur javac. Pour exécuter vos fichiers compilés, utilisez java. Des informations pour utiliser Java sur les machines SunOS sont disponibles sur le serveur.

La méthode SunOS fonctionne aussi sur les machines Solaris. En cas de doutes utilisez là. Vous pouvez aussi utiliser le Makefile à la fin de ce cours. Il utilise la "méthode SunOS", donc il devrait fonctionner pour toutes les machines unix du département.

Comme premier exemple, vous pouvez copier le fichier source à la fin de ce cours dans votre répertoire. Il doit s'appeler "helloWorld.java". Pour le compiler vous pouvez faire "make" et pour l'exécuter "java helloWorld".

Bases: instructions, types, fonctions

Java ressemble beaucoup à C ou C++, si vous connaissez l'un des deux langages, vous ne devriez pas avoir de difficulté à apprendre les bases.

Les types

Les types sont les mêmes qu'en C (char, int, float, etc.) plus quelques autres :

boolean : pour les deux valeurs true et false.

byte : pour un octet (one byte). Un char occupe deux octets (comme un entier).

Les types composés

les tableaux : une déclaration typique est : `int[] leTableau=new int[10];`

En fait, un tableau est une instance de la classe "array". Cette classe a par exemple un attribut length (appelée par "leTableau.length") qui donne la taille du tableau.

String : pour les chaînes de caractères.

Premières différences avec C

- Il n'y a pas équivalence entre les entiers et les booléens (0 n'est pas false et 1 n'est pas true). Par exemple, `if (x = 4)` ... est une faute, et non pas toujours vrai comme en C.
- Il n'y a pas de pointeurs explicites (parce que tout est un pointeur en fait).
- Il n'y a pas de struct et de union. Ils sont remplacés par les classes.

Instructions

Elles sont en générale identiques à des instructions C. L'affectation, le "if-then-else", les boucles, le "switch" fonctionnent de la même faon.

Les fonctions sont définies comme en C :

```
type-retour nom-fonction(parametres)
  { instructions }
```

Différences importantes avec C :

- Il n'y a pas de préprocessing ("#define", "#include", ...)
- Les entrées/sorties se font au travers de "System.in" et "System.out"

  Exemple d'écriture :

  ```
  System.out.print("Il est ");
  System.out.print(5);
  System.out.println(" heures");
  ```

  Exemple de lecture :

  ```
  char c;
  c = (char)System.in.read();
  ```

  Voir aussi le programme "helloWorld.java".

Les classes

Une classe simple est définie comme ceci (les parties en MAJUSCULES sont à remplir) :

```
class NOM {
  ATTRIBUTS

  METHODES
}
```

- Une déclaration d'attribut se fait comme une déclaration de variable en C.
- Une déclaration de méthode se fait comme une déclaration de fonction en C.

Par exemple :

```
class Mammifere {
  int age;
  String nom;

  int nbDents() {
        return( (age < 18) ?age *2 : 34 );
  }

  void display() {
        System.out.println("Mon nom est : " + nom);
        System.out.println("et j'ai : " + nbDents() + " dents\n");
  }
}
```

Dans cet example, la classe s'appelle "Mammifere", elle a deux attributs "age" et "nom" et deux méthodes "nbDents" et "display".

L'héritage se fait en donnant le mot clé "extends" et le nom de la super-classe :

```
class Vache extends Mammifere {
  public Vache( String n, int a) {
  super(n,a);
  }

  public void display() {
        System.out.println("Mheuuu, je suis une vache");
        super.display();
  }
}
```

Dans cet exemple, La nouvelle classe “Vache” est une sous-classe de “Mammifere”, elle hérite les deux attributs et les deux méthodes définies dans “Mammifere”.

Les classes ont aussi des méthodes spéciales appelées *constructeurs* (comme en C++) qui sont exécutées à la création de chaque objet (par new, voir plus loin). Un constructeur a le même nom que sa classe et pas de type de retour. On ne peut pas l’appeler explicitement (ce n’est pas une méthode “normale”) :

```
class Mammifere {
  int age;
  String nom;

  public Mammifere( String n, int a) {
        nom = n;
        age = a;
  }
...
}
```

Les constructeurs sont habituellement utilisés pour initialiser les objets (ici, en donnant une valeur aux deux attributs). Ils peuvent prendre des paramètres (comme dans l’exemple), ou bien demander des informations à l’usager (à l’écran), ou encore lire les informations dans un fichier, ou n’importe quelle autre méthode.

On crée une nouvelle instance par l’opérateur “new” suivi du nom de la classe :

```
Mammifere unMammifere;

unMammifere = new Mammifere("Leo_le_lion", 22);
```

Si on passe des paramètres à la création de l’objet, il faut qu’ils correspondent aux paramètres d’un constructeur déclaré dans la classe.

Programmes

Un programme est simplement un ensemble de classes et leurs méthodes. Un programme doit absolument contenir une classe avec une méthode “main”. L’exécution du programme commence par cette méthode “main”. Il arrive souvent que la méthode “main” n’appartienne vraiment à aucune classe du programme, on crée alors une classe “bidon” qui aura pour seul usage de contenir la méthode “main”.

Cette méthode est souvent déclarée comme suit (voire plus loin pour une explication des mot “marqueurs” qui précèdent la définition de la méthode : `static public void main()`

Normalement, on déclare une classe par fichier, et le fichier à le même nom que la classe.

Notions plus avancées

Les notions qui suivent sont indispensables pour commencer à programmer.

This et Super

Chaque objet à instance au deux variables prédéfinies : "this" et "super".

La variable "this" est la même qu'en C++, elle contient l'objet lui-même. C'est un moyen de passer l'objet lui-même en paramêtre à un autre objet.

La variable "super" est aussi fondamentale, elle n'existe pas en C++. Cette variable permet d'accéder à une implémentation d'une méthode d'un niveau supérieur dans la hiérarchie d'héritage.

Par exemple, dans cette déclaration :

```
class Vache extends Mammifere {
  public void display() {
        System.out.println("Mheuuu, je suis une vache");
        super.display();
  }
}
```

La méthode "display()" de la classe Vache affiche un message puis appelle la même méthode au niveau supérieur dans la hiérarchie d'héritage (ici Mammifere). On a ici un des cas acceptable de redéfinition d'une méthode dans une sous-classe (v. l'extension dans le cours *modelisation2*).

Static, Public, Protected, Final, Abstract

Vous pouvez utiliser les "marqueurs" suivants pour les attributs ou méthodes :

Public : Pour une méthode ou un attribut qui est dans l'interface de la classe (visible de l'extérieur).

Protected Pour une méthode ou un attribut qui est en dehors de l'interface de la classe (non visible de l'extérieur); utilisé pour les méthodes ou attributs "utilitaires".

Abstract : Pour une méthode qui est abstraite. La classe qui définit cette méthode doit aussi être déclarée abstraite.

Static : Pour une méthode ou un attribut qui appartiennent à la classe (par opposition à une méthode ou un attribut des instances de la classe). Une méthode ou un attribut "static" est à l'intérieur de la classe elle-même, et n'ont pas à l'intérieur de chaque instance. Cela revient à considérer la classe comme un objet normal.

On déclare habituellement la méthode "main" (v. section précédente) comme "static", ceci évite de devoir créer une instance pour appeler la méthode, il suffit d'appeler la méthode pour la classe qui la possède.

Et pour les classes vous pouvez utiliser :

Abstract : Pour définir une classe abstraite. La classe ne peut pas avoir d'instances.

Final : Pour une classe, dont on ne veux pas qu'elle ait des sous-classes.

Il y a d'autres marqueurs, mais il ne sont pas aussi utiles et ils risquent de vous apporter de la confusion. N'en tenez pas compte.

ATTENTION malheureusement les marqueurs ne peuvent pas être mis dans n'importe quel ordre. si vous avez une erreur de compilation, vérifier cela.

"Importations"

Si vous voulez utiliser une des classes du système ou d'une librairie, vous devez importer le "package" qui contient cette classe.

Par exemple, pour utiliser les méthodes d'entrée/sortie, il faut importer le "package" d'entrée/sortie : `import java.io.*` Dans cet exemple, l'étoile indique qu'on veux importer toutes les classes du package "java.io". Si vous savez exactement quelle classe vous voulez importer, vous pouvez indiquer son nom à la place de l'étoile. Cela sauvera un peu de travail au compilateur qui n'aura pas besoin de compiler inutilement toutes les classes du package, amis seulement celle dont vous avez besoin.

Threads, Exceptions

Si vous débutez, oubliez les "threads", ils ne vous seront d'aucune utilité.

Il faudrait aussi ne pas tenir compte des exception, malheureusement ce n'est pas possible car certaines méthodes les nécessitent absolument (lecture à l'écran ou dans un fichier).

Vous n'avez pas besoin de savoir ce que sont les "exceptions" pour utiliser java. Retenez simplement qu'une exception est une erreur d'exécution. Certaines méthodes que vous devrez utiliser (lecture au clavier) peuvent provoquer ces erreurs ("throws ¡exception¿"). Quand vous utilisez une de ces méthodes, il faut dire à java quoi faire si cette erreur apparaît. Le plus simple est alors de dire que si une méthode que vous utilisez provoque une erreur, votre propre méthode va provoquer la même erreur.

Par exemple dans notre programme "helloWorld", notre méthode "main" utilise une méthode du système ("DataInputStream.readline()") qui peut provoquer une erreur de type Entrée/Sortie (IOException).

Nous devons alors indiquer que notre propre méthode peut aussi provoquer cette erreur:

```
void main()
  throws java.io.IOException {
  ...}
```

Le problème des exceptions, c'est que si une de vos méthode peut lancer une exception (i.e. provoquer une erreur), toutes les méthodes qui vont l'utiliser vont devoir "déclarer" cette exception aussi, et toutes les méthodes qui utilisent celles-ci, etc.

La déclaration "throws ¡exception¿" peut alors avoir tendance à se répandre à toutes les méthodes si votre programme n'est pas très bien conçu.

Interface

Java n'autorise pas l'héritage multiple, une classe ne peut hériter que d'une seule autre classe à la fois. Mais il y a un mécanisme (les "interfaces") qui permet d'avoir certaines facilités de l'héritage multiple.

En Java vous pouvez définir une sorte de type abstrait qu'on appelle une *interface*. Une interface se définie comme une classe un peu spéciale qui n'a que des méthodes. Mais ces méthode n'ont pas de corps, seulement une "signature".

```
interface NOM {
  METHODES
}
```

Il est possible de déclarer des variables dont le type est une interface, mais on ne peut pas créer d'instances d'une interface. À la place, on peut dire qu'une classe "implémente" cette interface. Les instances de la classe sont ensuite considérées comme étant du type de leur classe mais aussi de l'interface, on peut mettre ces instances dans une variable du type de la classe ou du type de l'interface.

Par exemple, on peut définir l'interface suivante :

```
interface AnimalCompagnie {
  String nom();
```

Qui indique qu'un animal de compagnie a un nom, et une méthode pour accéder ce nom.

Maintenant, on peut définir une sous-classe des animaux qui est aussi un animal de compagnie :

```
class Chien extends Mammifere implements AnimalCompagnie {
  String leNom;

  public void display() {
    System.out.println("OuafOuaf, je suis un chien");
    super.display();
  }

  public String nom() { return(leNom); }
}
```

Et finalement, on peut maintenant définir une variable de type "AnimalCompagnie" et lui affecter un "Chien".

Les interfaces, ne permettent pas de faire de la réutilisation de code comme le ferait l'héritage multiple. Chaque classe qui "implémente" une interface, doit définir les méthodes de cette interface.

Par contre, on peut d'efinir des méthodes qui prendront en paramêtre une interface.

Documentation (livre et Web)

- Un bon livre pour débuter : "On To Java", Patrick Henry Winston, sundar Narasimhan, Addison-Wesley.
- Et un autre, encore pour débuter : "Java sourcebook", Ed Anuff, Wiley Computer Publishing.
- Un tutoriel (http://www.javasoft.com:80/nav/read/Tutorial/index.html).
- Java au département (http://www.csi.uottawa.ca/ kwhite/java.html).
- Documentation sur Java (file:/usr/local/java/docs/api/API_users_guide.html).

Fichiers sources

Voici les fichiers sources auxquels nous avons fait références dans le cours.

Makefile

```
## General Makefile to compile java applications
## Just put all your .java files in a directory and call 'make'
## or to compile a given file do 'make the_file.class'

## Compiler and "runner"
JAVA   = netscape -java -classpath .:/usr/local/java/lib/classes.zip
JAVAC  = $(JAVA) sun.tools.javac.Main

## All files
SRC:sh = ls *.java
OBJ    = $(SRC:%.java=%.class)

## General rule for compiling a java file
%.class: %.java
    @echo javac $<
    @$(JAVAC) $<

## By default, compile everything we found
all: $(OBJ)
```

helloWorld.java

```
import java.io.*;    // Needed for using DataInputStream.readLine()
```

```
// Simple hello World Program for Java

class helloWorld {
  static public void main(String argv[])
       throws java.io.IOException {
    String nom;

    System.out.print("Bonjours, quel est ton nom ? ");
    System.out.flush();
    nom = new DataInputStream( System.in).readLine();
    System.out.println("Veux-tu jouer avec moi, " + nom + " ?");
  }
}
```

Animaux.java

```
import java.io.*;

class Mammifere {
  protected int age;
  protected String nom;

  public Mammifere() {
    System.out.println("Creation d'un nouveau mammifere");
  }
  public Mammifere( String n, int a) {
    this();
    nom = n;
    age = a;
  }

  public int nbDents() {
    return( (age < 18) ?age *2 : 34 );
  }

  public void display() {
    System.out.println("Mon nom est : " + nom);
    System.out.println("et j'ai : " + nbDents() + " dents\n");
  }
}


class Vache extends Mammifere {
```

```
  public Vache( String n, int a) {
  super(n,a);
  }

  public void display() {
    System.out.println("Mheuuu, je suis une vache");
    super.display();
  }
}


class Animaux {
  static public void main(String argv[]) throws java.io.IOException {
    Mammifere m;
    Vache     v;
    int a;

    System.out.print("age? ");
    a = new StreamTokenizer(System.in).nextToken();
    System.out.println("age read: "+a);
    m = new Mammifere("Hector", 32);
    m.display();

    v = new Vache("Marcel", a);
    v.display();
  }
}
```