51IF1IF1
Introduction à l'informatique et à la programmation

Support du cours-td

2013-2014

Table des matières

Présentation de l'enseignement

Objectifs et organisation

Cet enseignement a plusieurs objectifs : comprendre un certain nombre des concepts généraux des machines et de la programmation, réaliser le codage effectif d'algorithmes simples, compiler et exécuter des programmes dans un environnement de type Unix. Voici son organisation :

Cours le jeudi de 14h30 à 16h30 en amphi 1A

	septembre			octobre			novembre				décembre
MATH+MASS	12	19		03		31		14		28	
INFO+MI+CPEI+LI	*09*		26		17		07		21		05

Cours-TD un créneau de deux heures hebdomadaire

TP un créneau de trois heures hebdomadaire

En complément, un système de soutien est mis en place :

Tutorat le midi du lundi au vendredi au SCRIPT (fin septembre début octobre)

Commission de suivi sur rendez-vous au département sciences exactes

Contrôle de connaissances

L'évaluation se fait au cours de tests (sur papier et/ou machine) et d'examens (sur papier) :

- **Td** résultat des tests effectués en cours-TD
- **Tp** résultat des tests effectués en TP
- **E0** partiel (samedi 26 octobre de 13h à 15h, à confirmer)
- **E1** examen session 1 en décembre
- **E2** examen session 2 en juin

Les notes finales sont calculées selon le principe suivant :

Contrôle continu	**Cc** = (**Td** + **Tp**) / 2
Note d'écrit	**Ne** = max (**E1**, (**E0** + **E1**) / 2)

Note session 1	(3**Ne** + **Cc**) / 4
Note session 2	max (**E2**, (3**E2** + **Cc**) / 4)

La présence à au moins un test en cours-TD et un test en TP est nécessaire au calcul de **Cc**.
Attention : pas de note ⇒ pas de moyenne ⇒ pas de compensation.

Bibliographie

[1] Alfred Aho and Jeffrey Ullman, *Concepts fondamentaux de l'informatique*, Dunod, Paris, 1993, (cote bibliothèque : 004 AHO).

[2] Mary Campione, Kathy Walrath, and Alison Hulm, *The java tutorial (fourth edition)*, Addison-Wesley, 2000, `http://java.sun.com/docs/books/tutorial/`.

[3] Bruce Eckel, *Thinking in java*, `http://www.mindview.net/Books/TIJ/` ou `http://penserenjava.free.fr/`.

[4] Yakov Fain, *Programmation java pour les enfants, les parents et les grands-parents*, 2005, `http://java.developpez.com/livres/javaEnfants/`.

Chapitre 1

Présentation du langage Java

1.1 Qu'est-ce-qu'un programme ? À quoi ça sert ?

Extraits de wikipedia :

> En informatique, un programme est une suite d'opérations pré-déterminées destinées à être exécutées de manière automatique par un appareil informatique en vue d'effectuer des travaux, des calculs arithmétiques ou logiques, ou simuler un déroulement.
>
> Le *code source* [1] est la forme sous laquelle un programme est créé par un programmeur : un ensemble de textes décrivant formellement, étape par étape, les opérations à effectuer ; rédigés conformément aux règles d'un langage de programmation.
>
> [http://fr.wikipedia.org/wiki/Programme_(informatique)]

> La particularité principale de Java est que les logiciels écrits dans ce langage sont très facilement portables sur plusieurs systèmes d'exploitation [...] avec peu ou pas de modifications. [...] L'indépendance vis-à-vis de la plate-forme, signifie que les programmes écrits en Java fonctionnent de manière parfaitement similaire sur différentes architectures matérielles. On peut effectuer le développement sur une architecture donnée et faire tourner l'application sur toutes les autres.
>
> Ce résultat est obtenu par les compilateurs Java qui compilent le code source "à moitié" afin d'obtenir un *bytecode* (plus précisément le bytecode Java, un langage machine spécifique à la plate-forme Java). Le code est ensuite interprété sur une machine virtuelle Java (JVM en anglais), un programme écrit spécifiquement pour la machine cible qui interprète et exécute le bytecode Java. [http://fr.wikipedia.org/wiki/Java_(langage)]

Parler de l'exécution d'un programme Java est donc un abus de langage : c'est la machine virtuelle Java qui s'exécute ; on lui fournit un argument qui désigne le bytecode correspondant au programme.

1.2 Exemples de programmes Java

```
1 /** affiche Hello World!
   */
3 class Hello {
    public static void main(String[ ] args) {
5     System.out.println("Hello World!");
    } /* fin de la definition de main */
7 } // fin de la definition de Hello
```

Hello.java

1. On pourra parler de code ou de source pour désigner le code source. (NdA)

La compilation d'un code Java s'effectue avec la commande `javac` (*java compiler*) suivie du nom du fichier contenant le source qui porte le suffixe `.java`. Cette commande transforme le code qui est un texte lisible par un être humain en *bytecode* lisible par la machine virtuelle Java. Au passage le compilateur vérifie que la syntaxe du langage est respectée, par exemple qu'une opération effectuée sur une variable est compatible avec le type (déclaré) de celle-ci, par exemple on ne peut pas affecter la valeur `3.5` à un entier. Le compilateur ne connaît pas les valeurs des variables.
Un programme Java s'exécute grâce à la commande `java` suivie du nom de la classe à exécuter (celle qui contient la fonction `main` avec les bonnes spécifications). Durant l'exécution, la machine virtuelle suit dans l'ordre les instructions du `main`.
Par exemple, le programme suivant demande à l'utilisateur de donner un nombre et affiche sa partie entière.

```
import java.util.Scanner;

class PartieEntiere{
    public static void main(String[] args){
        double x;
        Scanner sc = new Scanner(System.in);
        System.out.print("Donnez␣un␣nombre␣quelconque␣:␣");
        if(sc.hasNextDouble()){
            x = sc.nextDouble();
            System.out.println("[" + x + "]␣=␣" + partieEntiere(x));
        }
    }

    /** renvoie la partie entiere d'un reel
     * @param x
     * @return [x]
     */
    static int partieEntiere(double x){
        int e = 0;
        if(x >= 0){
            while(e <= x) e++;
            return e - 1;
        } else {
            while(e > x) e--;
            return e;
        }
    }
}
```

PartieEntiere.java

► **Exercice 1** (Examen 2006/2007, exercice 2, question 2.1) :
Soit le code Java suivant :

```
class Exam2 {
  public static void main(String[] args){
    int v1 = 3, v2 = 6;
    if (v1+2 < v2) System.out.println("ric");
    if (v1+4 < v2) System.out.println("hoc");
    if (v1+4 > v2) System.out.println("het");
  }
}
```

Quel doit être le nom du fichier contenant ce code source, quelle commande permet de le compiler, quel est le nom du fichier *bytecode* obtenu et comment réalise-t-on l'exécution ?

1.3 Erreurs

1.3.1 Erreurs à la compilation

Si le fichier comporte des erreurs de syntaxe (c'est-à-dire s'il ne respecte pas les spécifications de `Java`), une erreur se produit à la compilation et aucun fichier `.class` n'est créé.

Exemples d'erreurs pouvant se produire :
- le `;` manque à la fin d'une instruction,
- un identificateur ne peut être interprété,
- il y a incohérence dans les paires d'accolades.

Attention : une compilation qui se passe bien ne signifie pas que votre programme est correct, ni même qu'il s'exécutera sans erreur.

1.3.2 Erreurs à l'exécution

Des erreurs peuvent se produire à l'exécution.
Par exemple dans le programme ci-dessous, on effectue une division par 0. Le compilateur ne peut pas la détecter car il ne connaît pas la valeur d'une variable mais juste son type.

```
class DivisionParZero{
    public static void main(String[] args){
        int c, a=3, b=0;
        System.out.println("debut");
        c = a/b; // erreur a l'execution
        System.out.println("fin");
    }
}
```

DivisionParZero.java

L'exécution s'arrête au moment de l'erreur.
Le message d'erreur est important car il permet de tracer l'erreur depuis son origine. On modifie la classe précédente :

```
class DivisionParZero2{
    static void diviserParZero(){
        int c, a=3, b=0;
        System.out.println("debut");
        c = a/b; // erreur a l'execution
        System.out.println("fin");
    }

    public static void main(String[] args){
        diviserParZero();
    }
}
```

DivisionParZero2.java

Son exécution donne le traçage d'erreur suivant :

```
$ java DivisionParZero2
debut
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivisionParZero2.diviserParZero(DivisionParZero2.java:5)
        at DivisionParZero2.main(DivisionParZero2.java:10)
```

trace de l'exécution de `DivisionParZero2`

► **Exercice 2** :
Écrire ce que serait le message d'erreur à l'exécution de `DivisionParZero`.

Autre exemple : le source ne contient pas de `main` avec les bonnes spécifications : la machine virtuelle ne sait pas ce qu'elle doit faire.

1.4 Structure d'un programme `Java` simple

La structure de base d'un programme est la suivante :

```
class ClasseLaPlusSimple {
    public static void main(String[] args){
        // declarations de variables
        // instructions
    }
}
```

ClasseLaPlusSimple.java

Le nom du fichier doit être le nom de la classe suivi du suffixe `.java`. Si la compilation n'amène pas à une erreur, la compilateur produit un fichier bytecode dont le nom est celui de la classe suivi du suffixe `.class`. Pour éviter d'avoir un `main` trop long et illisible, on fabrique des "boîtes noires" en écrivant des méthodes ou fonctions (la différence sera expliquée au second semestre). Celles-ci peuvent être appelées plusieurs fois et à n'importe quel moment dans le programme.

```
// importations

class ClassePlusEvoluee {
    static void uneFonction(){
        // corps de la fonction
    }

    public static void main(String[] args){
        // declarations de variables
        // instructions (appel a uneFonction, etc.)
    }
}
```

ClassePlusEvoluee.java

1.5 Identificateurs

Pour désigner les éléments d'un programme (classes, variables, objets, fonctions, méthodes, constantes), on leur donne un nom, appelé *identificateur*. Cet identificateur commence par une lettre (le caractère souligné _ est considéré comme une lettre) qui est suivie de lettres ou de chiffres ; il a une longueur quelconque.
Les minuscules et majuscules sont des lettres disctinctes.

Exemple 1. `MaClasse`, `AutreClasse`, `uneMethode`, `uneAutreMethode`, `x`, `X`, `nbCotes`

Il existe des conventions de nommage en `Java`. Ne pas les respecter n'empêche pas la compilation, ni l'exécution, mais rend la relecture et le travail à plusieurs plus difficiles. Parmi ces conventions :
- les noms de classes commencent par des majuscules,
- les noms de méthodes/fonctions et de variables commencent par des minuscules,
- quand on "colle" plusieurs mots, chaque initiale (sauf éventuellement pour le premier mot) est une majuscule : `ClasseQuiRepresenteQuelqueChose`, `methodeQuiCalculeQuelqueChose`.

De même, utiliser un nom pas trop long mais qui permette d'identifier le rôle d'un élément facilite le travail.

1.6 Comment fabrique-t-on un programme ?

On commence par écrire un algorithme en français :
- pas de contrainte de syntaxe,
- niveau d'abstraction adapté au problème et aux connaissances.

Pour des programmes écrits dans cette syntaxe abstraite, on parle de langage algorithmique abstrait.
Par exemple, si l'on souhaite écrire un programme qui demande à l'utilisateur de fournir un entier n et qui ensuite calcule la factorielle de n et affiche le résultat, le programme en langage algorithmique abstrait pourra avoir la forme suivante :

```
Lire un entier et le mettre dans n
Si n>=0 alors
  res <- 1
  Si n>0 alors
    Pour i allant de 1 a n
      res <- res*i
    FinPour
  FinSi
FinSi
Afficher res
```

Le programme correspondant en `Java` s'écrira alors :

```
import java.util.Scanner;

class Factorielle{
    public static void main(String[] args){
        int n,res,i;
        res=0;
        Scanner sc=new Scanner(System.in);
        System.out.print("Donner␣un␣nombre␣entier␣:␣");
        n=sc.nextInt();
        if(n>=0){
            res=1;
            if(n>0){
                for(i=1;i<=n;i++)
                    res=res*i;
            }
        }
        System.out.println("Le␣resultat␣est␣:␣"+res);
    }
}
```

Factorielle.java

► **Exercice 3** (Examen 2004/2005, exercice 2, questions 2.1.1 et 2.1.2) :

Un diviseur strict d'un entier $n > 0$ est un entier strictement positif, différent de n et qui le divise (cela signifie que le reste de la division entière est nul).

1. Donner une borne supérieure raisonnable du plus grand diviseur strict d'un entier $n > 0$.
2. Écrire en langage algorithmique abstrait un algorithmique qui demande un entier et qui affiche tous ses diviseurs stricts. (Pour se faire, on suppose que l'on dispose de l'opérateur booléen $|$ (divise) tel que, étant donnés deux entiers a et b, $a|b$ est vrai si a divise b et est faux sinon).

► **Exercice 4** (Partiel 2004/2005, exercice 2) :

1. On dispose de trois pièces de monnaie apparemment identiques nommées `a`, `b` et `c` dont on sait qu'au plus une est fausse (elle n'a pas le même poids que les deux autres). On dispose d'une balance permettant de comparer le poids des objets posés sur chacun des deux plateaux. Décrire un algorithme permettant de tester si une des trois pièces est fausse et si oui laquelle (pour alléger l'écriture, on pourra supposer que la valeur d'une variable `x` correspondant à une pièce est égale au poids de cette pièce).
2. Décrire un algorithme pour 4 pièces `a`, `b`, `c` et `d` (au plus une étant fausse) en procédant uniquement par comparaison du poids des pièces (on posera toujours deux pièces sur chaque plateau).

► **Exercice 5** (Partiel 2005/2006, exercice 2, question 2.3) :

On dispose de m pièces de 50 centimes, de n pièces de 20 centimes et de p pièces de 10 centimes (on supposera que p est non nul).
On s'intéresse au paiement de sommes sans rendu de monnaie avec un tel assortiment de pièces.
Exprimer, sous forme algorithmique simple, une stratégie de paiement d'une somme s avec un nombre minimum de pièces pour un assortiment donné de pièces de 50, 20, 10 centimes si le paiement est possible et qui détecte l'impossibilité de réaliser le paiement.

Indication On prendra par ordre décroissant des valeurs des pièces, le maximum de pièces possibles. On admettra que cette stratégie donne une solution si le paiement est possible (cela en raison de l'hypothèse qu'on dispose d'au moins une pièce de 10 centimes).

Chapitre 2

Variables

2.1 Types

Un *type* correspond à la définition de :
– un ensemble de valeurs,
– un ensemble d'opérations applicables aux éléments de cet ensemble de valeurs et la spécification de comment utiliser ces opérations.
En `Java`, on distingue les *types primitifs* (par exemple entiers, réels et booléens) et les *types références* (tableaux, *classes*).
Nous définirons les types au fur et à mesure de leur utilisation.

2.1.1 Des types primitifs

Les tests suivants existent pour tous les types primitifs (il s'agit de comparer des éléments de même type) :

tests de comparaison	
`==`	égalité
`!=`	différence

le type booléen `boolean`

valeurs booléennes
`true`
`false`

opérations sur les booléens	
`&&` ou `&`	et
`||` ou `|`	ou
`!`	non
`^`	ou exclusif

un type entier (il en existe d'autres, voir § 6.1) : `int` (codé sur 4 octets)

opérations sur les entiers	
`-`	moins unaire
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	quotient entier
`%`	reste

tests de comparaison	
> (ou >=)	supérieur (ou égal)
< (ou <=)	inférieur (ou égal)

Exemple 2. `5, -17`

La multiplication, le quotient entier et le reste sont prioritaires sur l'addition et la soustraction. Le moins unaire est l'opération la plus prioritaire. Quand deux opérations de même priorité apparaissent, c'est celle de gauche qui est effectuée en premier. On peut mettre des parenthèses pour forcer une priorité.

► **Exercice 6** :

Donner la valeur des trois expressions suivantes : `3+5/3` `4*1/4` `2/3*3-2`

► **Exercice 7** (Partiel 2005/2006, exercice 2, questions 2.1 et 2.2) :

On reprend la situation de l'exercice 5 : m pièces de 50, n pièces de 20 et $p \neq 0$ pièces de 10 centimes.

1. Exprimer, sous forme d'une expression arithmétique `Java`, la somme maximale que l'on peut espérer payer avec cet assortiement de pièces.
2. Ne disposant pas de pièces de 1, 2 et 5 centimes, un certain nombre de sommes ne sont pas payables a priori (celles non multiples de 10), en plus de celles supérieures à la somme maximale précédente. Exprimer au moyen d'une expression logique, combinant ces conditions, une condition *nécessaire* pour qu'une somme donnée soit *susceptible* d'être payée.

un type réel (il en existe un autre, voir § 6.1) : `double` (codé sur 8 octets)

opérations sur les réels	
`-`	moins unaire
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division

tests de comparaison	
`>` (ou `>=`)	supérieur (ou égal)
`<` (ou `<=`)	inférieur (ou égal)

Exemple 3. `3.57`, `-.002`, `12.6e-3`

La multiplication et la division sont prioritaires sur l'addition et la soustraction. Quand deux opérations de même priorité apparaissent, c'est celle de gauche qui est effectuée en premier.

Ordre des types primitifs

Il existe des compatibilités entre les types primitifs, en particulier certaines conversions implicites sont possibles. Ces compatibilités sont définies comme suit :

`int` $\rightarrow$ `double`

Ainsi, une valeur de type `int` peut être vue comme une valeur de type `double`, mais pas le contraire. Quand on effectue une opération entre deux valeurs qui n'ont pas le même type (mais sont compatibles), le résultat obtenu est du type le plus grand (relativement à l'ordre ci-dessus).

Le type booléen n'est compatible avec aucun autre type.

2.1.2 Les types références

Il s'agit de types contenant souvent plusieurs éléments. Ils apparaissent sous forme de tableaux (plusieurs éléments de même type, voir chapitre 5) ou de classes (plusieurs éléments de types a priori différents). Les classes peuvent être définies dans des paquetages extérieurs ou par le programmeur lui-même.

le type `String` (*chaîne de caractères*) Il sert à coder une suite finie de caractères [1]. Les constantes littérales de ce type sont écrites entre guillemets.

Exemple 4. `"Ceci␣est␣une␣chaine␣de␣caracteres."`
`"Ceci␣est␣egalement\nune␣chaine␣de␣caracteres."`

opération sur les `String`	
`+`	concaténation

► **Exercice 8** (Partiel 2012/2013, exercice 4) :

Préciser quelle erreur empêche la compilation de la classe ci-dessous. Expliquer ce que produirait son exécution après mise en commentaire de la ligne erronée, sauvegarde et compilation.

```
class Soixante17{
    public static void main(String[] args){
        int sept=7, dix=10, soixante=60;
        System.out.println("␣soixante-dix-sept␣␣+␣␣soixante-sept␣");
        System.out.println("␣soixante-dix-sept␣"+  soixante-sept  );
        System.out.println(  soixante-dix-sept  +  soixante-sept  );
        System.out.println(  soixante-dix-sept  +"␣soixante-sept␣");
    }
}
```

2.2 Variables et affectations

Une *variable* correspond à un emplacement en mémoire pouvant contenir une valeur d'un type donné. C'est le type de la variable qui permet l'interprétation correcte du contenu correspondant en mémoire.
Une variable est désignée par un identificateur valide (voir § 1.5). On peut lui affecter une *valeur* du type correspondant ou d'un type compatible.

2.2.1 Déclaration de variables

La première opération à effectuer avec une variable, avant toute utilisation de celle-ci, est sa déclaration : elle permet de lui attribuer un nom et un type.
Une instruction de déclaration prend la forme suivante :
`type nomDeLaVariable;`

Exemple 5.
```
int monEntier;
double x, y;
String str;
```

Variables de types primitifs Leur déclaration alloue la place en mémoire pour y stocker une valeur du type correspondant.

Variables de types références Leur déclaration alloue la place pour stocker l'adresse en mémoire d'un objet de type correspondant. L'espace nécessaire à stocker l'objet pourra être alloué avec l'opérateur `new` (voir chapitre 5).

2.2.2 Affectation

`nomDeLaVariable = valeurAffectee;`
C'est l'opération qui consiste à attribuer à une variable d'un certain type une valeur compatible avec ce type. **L'affectation d'une valeur à une variable remplace la valeur préalablement associée à cette variable.** Cet opérateur est noté = en `Java`. À la gauche du signe = on trouve toujours le nom d'une variable. À la droite du signe = on trouve une expression (voir chapitre 6 ; en particulier une valeur est une expression) dont le type est compatible avec celui de la variable. La variable prend alors comme nouvelle valeur l'évaluation de cette expression.
On peut déclarer et affecter une variable "en même temps" :

1. Il existe un type primitif caractère `char`, voir § 6.1.

```
int n = 3;
double x = 4.5, y, z = 3;
String s = "chaine";
```

► **Exercice 9** (Partiel 2010/2011, exercice 1) :

Expliquer ce que produit l'exécution du morceau de code Java suivant :

```
int x, y = 7, z = 8;
double s, t;
x = y + 2; System.out.println(x);
y = z - 3; System.out.println(x);
y = y % z; System.out.println(y);
x = x + 4; System.out.println(x);
x = z / y; System.out.println(x);
s = z / y; System.out.println(s);
s = y; t = z / s; System.out.println(t);
```

2.2.3 Constantes

Une *constante* est une variable portant le qualificatif `final`. Elle ne peut être affectée qu'une fois au cours du programme.

```
class DefinitionConstante {
    public static void main(String[] args){
        final double PI = 3.14;
        System.out.println("definition de PI : "+PI);
    }
}
```

DefinitionConstante.java

2.2.4 Portée d'une variable

À partir du moment où une variable existe, elle est visible sur une partie du programme, il est très important de comprendre sur laquelle.
Un *bloc* est un ensemble d'instructions délimité par des accolades ({ }). Un bloc peut en contenir un autre.
Une variable n'est pas visible avant sa déclaration, ni hors du bloc où elle est déclarée.

```
    static void porteeDesVariables1(){
        int i;
        // instructions avec i autorisees
        // instructions avec j, k et l non autorisees
        int j;
        // instructions avec i et j autorisees
        // instructions avec k et l non autorisees
        {
            // instructions avec i et j autorisees
            // instructions avec k et l non autorisees
```

```
            int k;
12          // instructions avec i, j et k autorisees
            // instructions avec l non autorisees
14      }
        // instructions avec i et j autorisees
16      // instructions avec k et l non autorisees
        int l;
18      // instructions avec i, j et l autorisees
        // instructions avec k non autorisees
20  }
```

PorteeDesVariables1.java

Le corps d'une fonction (voir § 3.1) est un bloc, les noms de variables sont donc locaux à une fonction.

```
1   class PorteeDesVariables2 {
        static void deux(){
3           int i=2; // variable i visible uniquement dans deux
        }
5       static void affichage(){
            System.out.println(i); // erreur: variable non declaree
7       }
    }
```

PorteeDesVariables2.java

Un nom de variable ne peut pas être réutilisé dans un sous-bloc.

► **Exercice 10** (Partiel 2012/2013, exercice 1, programme 1) :

Expliquer ce que produit l'exécution du programme suivant :

```
    class Omfang{
2       public static void main(String[] args){
            int a = 3, b = 6+a;
4           System.out.println(a+"␣"+b);
            if(a/b<b/a){
6               int c;
                c = b+a; b = b+c; a = b-c;
8               System.out.println(a+"␣"+b+"␣"+c);
            }else{
10              int c;
                c = b-a; b = b-c; a = b+c;
12              System.out.println(a+"␣"+b+"␣"+c);
            }
14          int c;
            c = b+a; b = b+c; a = b-c;
16          System.out.println(a+"␣"+b+"␣"+c);
        }
18  }
```