IFT1015 Introduction à la programmation : Exigences

Avec exercices de programmation

Miklós Csűrös

Département d'informatique et de recherche opérationnelle
Université de Montréal

Automne 2008

Chapitre 1

Notions de base, boucles, et tableaux

1.0 Unix/Linux

Il est absolument nécessaire de se familiariser avec le système d'exploitation Unix (ou Linux). Utilisez la ligne de commande pour gérer vos fichiers et répertoires. Même si vous travaillez sur votre propre ordinateur chez vous, uilisez la ligne de commande : en MacOS X, vous pouvez lancer l'application Terminal ; si vous travaillez avec un PC, installez une version de Linux ou Cygwin.

Il existe une version online du manuel du Séminaire Unix offert par le DIRO : `http://www-etud.iro.umontreal.ca/~semunix/` (v. «Documentation»).

Pour l'édition de fichiers, uilisez un éditeur générique comme Emacs. Avant que vous soyez expert en programmation Java, il est fortement découragé d'utiliser un environnement integré de développement comme Netbeans ou Eclipse.

1.0.1 Exigences

- ⋆ Commandes de base : `cd`, `pwd`, `mkdir`, `rmdir`, `cp`, `rm`, `mv`, `chmod`, `ls`, `man`.
- ⋆ Utilisation d'Emacs : ouvrir un fichier existant, ou créer un nouveau fichier, édition d'un fichier, sauvegarder.

1.1 Exécution de programmes Java

1.1.1 Fichiers de source

Votre code Java doit être placé dans un fichier de classe (c'est un fichier dont l'extension est `.java` et a le même nom que la classe).

Exercice 1.1. *Écrire le code suivant.*

```
public class ProgrammeSimple
{
    public static void main(String[] args)
    {
        System.out.println("C'est mon programme simple.");
    }
}
```

1.1.2 Compilation et exécution

Pour compiler votre code source, utilisez `javac`. Cela produit un fichier avec le même nom que votre fichier de source mais avec l'extension `.cls`.

Le programme `java` exécute le bytecode d'une classe. Plus précisement, la commande `java XXXXX` exécute le code de la méthode `main` dans la classe `XXXXX`, en utilisant la compilation encodée dans le fichier `XXXXX.cls`.

Exercice 1.2. *Compiler et exécuter* `ProgramSimple.java` *de l'exercice 1.1.*

Exercice 1.3. *Écrire, compiler et exécuter un programme qui affiche le début du poême de Verlaine :*

Les sanglots longs
des violons
de l'automne
blessent mon coeur
d'une langueur
monotone.

Exercice 1.4. *Écrire, compiler et exécuter un programme qui calcule le nombre de seconds dans un an de 365 jours.*

1.1.3 Exigences

⋆ Écrire, modifier, compiler et exécuter une classe Java exécutable en utilisant la ligne de commande et un logiciel d'édition générique (comme Emacs).

1.2 Variables et types

1.2.1 Notions de variable et de type

En théorie de langages de programmation, un **type** est défini par l'ensemble de valeurs possibles et les opérations sur celles-ci. Par exemple, le type `byte` en Java est défini par l'ensemble de valeurs entières $\{-128, -127, \ldots, 0, 1, \ldots, 127\}$ avec les opérations qui peuvent y appliquer comme les opérations arithmétiques `+`, `*`, etc.

Une **variable** est l'abstraction d'un emplacement en mémoire. Une variable a un **nom**, une adresse physique en mémoire (qui n'est pas nécessairement visible au programmeur) qui s'apelle la valeur gauche (**l-value**), un type, une valeur (stockée à l'adresse physique associé, en utilisant un bloc de cellules de mémoire selon l'implantation du type) qui s'apelle sa valeur droite (**r-value**), et une **visibilité** qui détermine les règles d'usage de la variable par son nom dans le code.

En Java, le type de toutes les expressions est fixé lors du temps de compilation. Le compilateur `javac` infère ce type en appliquant des règles bien définies, et génère une erreur de compilation pour l'utilisation illégale de types.

1.2.2 Types primitifs

Rangées Java définit les types simples («primitifs» dans la términologie de Java) : `boolean`, et les types numériques `char`, `byte`, `short`, `int`, `long`, `float`, `double`. Les deux derniers représente des nombres flottants ; `char`,...,`long` sont des types entiers. Le type `char` prend des valeurs entre 0 et 65535 ($2^{16} - 1$). Les types entiers `byte`,...,`long` prend des valeurs entre -2^{r-1} et $\left(2^{r-1} - 1\right)$ où $r = 8$ pour `byte`, $r = 16$ pour `short`, $r = 32$ pour `int` et $r = 64$ pour `long`.

Litéraux et constantes Le langage Java offre plusieurs types de constantes comme celles des types `bool` (`true` («vrai») et `false` («faux»)), `char` (`'a'`), entiers (`-99181`), nombres flottants (`1.3988`, `.5e-3`). Pour forcer l'interprétation d'une constante entière comme `long`, il faut ajouter 'L' à la fin : `12L`.

Les bornes des types sont prédéfinis comme les variables constantes `byte Byte.MIN_VALUE`, `byte Byte.MAX_VALUE`, `int Integer.MIN_VALUE`, `int Integer.MAX_VALUE`, etc. Les constantes `double Double.MAX_VALUE` et `double Double.MIN_VALUE` définissent le plus grand et le plus petit nombre positif qui peut être représenté par un type `double` ; les constantes `float Float.MAX_VALUE` et `float Float.MIN_VALUE` ont une utilité analogue.

D'autres variables utiles : `double Double.POSITIVE_INFINITY` et `double Double.NEGATIVE_INFINITY` qui représentent des valeurs infinies, `double Math.PI` qui a la valeur `double` la plus proche à $\pi = 3.14159265358979\cdots$.

Exercice 1.5. *Donner le code pour définir les variables locales a, b, c, d de types* `double`*,* `char`*,* `long` *et* `bool` *et valeurs initiales* $6.0221417 \cdot 10^{23}$*, le signe de tabulation TAB,* 2008 *et faux ; dans cet ordre.*

Exercice 1.6. *Écrire, compiler et exécuter un programme qui calcule le nombre de seconds dans le* 20^{e} *siècle.*

1.2.3 Variables locales et statiques

En Java, les variables sont déclarés par le syntaxe `type nom` qui peut être suivi par la définition de valeur initiale `=valeur`. À l'omission d'une valeur initiale spécifique, les variables sont initialisées par défaut : 0 pour des types entiers, 0.0 pour les nombres flottants et faux pour booléens. Il vaut mieux toujours faire une initialisation explicite pour rendre le code plus facile à comprendre et vérifier.

Si une variable est declarée à l'extérieur de toutes les méthodes dans un fichier de classe avec le mot-clé `static`, alors c'est une variable statique qui est visible partout dans la classe. Si la variable est déclaré dans une séquence d'instructions, alors c'est une variable locale qui est visible dans le même bloc d'instructions après sa déclaration. Si une variable locale a la même nom qu'une variable de classe, alors le nom réfère à la variable locale. Pour accéder à la variable de la classe, il faut utiliser son **nom qualifié**, c'est à dire fournir le contexte dans le style `Classe.variable`.

Exercice 1.7. *Compléter le pièce de code suivant (la ligne* `// CALCULER...`*) :*

```
public class ClasseExample
{
    public static int nombre=9;
    ...
    public static void main(String[] args)
    {
        int nombre = 12;
        ...
        // CALCULER LA SOMME DES DEUX VARIABLES 'nombre'
    }
}
```

1.2.4 Tableaux

Les tableaux sonts des types composés pour représenter des vecteurs de valeurs du même type. On peut avoir un tableau de n'importe quel autre type, donc même d'un autre type tableau. Par exemple `int[]` est le type d'un tableau d'entiers, `double[][]` est un tableau de tableaux de doubles, `String[]` est un tableau de `String`s.

Les tableaux sont des objets en Java, et donc leur valeur initiale est `null`. Pour allouer du mémoire, faire `tableau = new type[taille]`. Pour des tableaux de tableaux, on peut spécifier des tailles en commun pour des représenter des matrices de n'importe quelle dimension :
`int[][][][] x = new int[2][3][4][5];`. On peut également allouer les éléments d'un tel tableau séparamment :

```
int[][][][] x = new int[2][3][][];
for (int i=0; i<2; i++)
    for (int j=0; j<3; j++)
        x[i][j] = new int[i+j+1][4];
```

La valeur droite d'une variable de type tableau est la **référence** à l'objet qui contient les entrées. Le nombre d'éléments dans le tableau est stocké par la variable `length` du tableau. On peut intialiser les éléments d'un tableau en utilisant le syntaxe
`{ entree0, entree1, ..., }`.

Exercice 1.8. *Écrire du code qui convertit le mois défini dans une variable* `int jour` *en une chaîne de caractères* `String nom_du_jour` *correspondante. Pour* $\texttt{jour} = 1$, `nom_du_jour` *doit être* `dimanche`, *pur* $\texttt{jour} = 2$, `nom_du_jour` *doit être* `lundi`, *etc. Utiliser un tableau de* `String` *qui contient les noms des jours.*

1.2.5 Conversion de types

«Conversion» est le terme pour dénoter le changement de type, accompagné par la traduction nécessaire de valeurs. La conversion est automatique quand il s'agit d'un enlargement de type, comme entre les types simples dans la série `byte` → `short` → `int` → `long` → `float` → `double` ou de `char` à `int`. (La conversion n'est pas automatique de `byte` ou de `short` à `char` — la valeur est converti en `int`, et ensuite les règles de conversion de `int` à `char` sont appliquées.) La conversion peut être forcée en utilisant un *cast* avec le syntaxe `(type) expression` qui convertit `expression` d'un type quelconque au type désiré. Dans le cas d'une conversion d'un nombre flottant (`double` ou `float`) vers un nombre entier (`int`, `long`, etc.), Java utilise la troncation (arrondi vers le nombre entier inférieur) pour la projection de valeurs. Notez qu'il n'existe pas de conversion à partir d'un `String` à un type primitf : il faut utiliser les méthodes `.parse...` (voir §1.4.2).

Exercice 1.9. *Donner une expression qui calcule l'arrondi usuel (vers le plus proche nombre entier) d'une variable* `double z`.

Exercice 1.10. *Donner une expression qui calcule l'arrondi au deuxième chiffre après le point décimal pour une variable* `double z`. *Par exemple, si* $z = 78.0567$*, l'expression devrait donner* 78.06.

1.2.6 Exigences

- ⋆ Types primitifs, leurs bornes
- ⋆ Opérations arithmétiques `+`, `-`, `*`, `/` `%` ; incrémentation/decrémentation `++` et `--` ; comparaisons `<`, `>`, `<=`, `>=`, `==`, `!=` ; opérations logiques `&&`, `||`, `!` ; opérateur conditionnel `? :`.
- ⋆ Usage de variables locales : déclaration, initialisation et valeurs par défaut
- ⋆ Visibilité de variables
- ⋆ Usage de tableaux : syntaxe, allocation de mémoire, variable de tableau `length`, valeur initiale, tableaux de deux dimensions ou plus
- ⋆ Conversion automatique entre types simples, conversion forcée (*cast*)

1.3 Instructions de contrôle

Les instructions sont normalement executées l'une après l'autre dans un bloc d'instructions. Le flux de contrôle peut être changé par les constructions `if`–

`then-else` et `switch` (exécution conditionnelle), et les instructions de boucles. Dans un bloc de `switch`, il est typiquement nécessaire de forcer la **complétion précipitée** de l'exécution avec le mot-clé `break` : sinon, toutes les intructions après le `case` approprié sont executées, même à travers d'autres `case`s.

Exercice 1.11. *Écrire du code qui convertit le mois défini dans une variable* `int jour` *en une chaîne de caractères* `String nom_du_jour` *corréspondante. Pour* `jour` $= 1$*,* `nom_du_jour` *doit être* `dimanche`*, pur* `jour` $= 2$*,* `nom_du_jour` *doit être* `lundi`*, etc. Utilisez la structure* `switch` *dans la solution.*

1.3.1 Boucles

Pour exécuter le même bloc d'instructions autant de fois qu'on veut, il faut l'emballer dans une structure `for`, `while` ou `do-while`. Le syntaxe de la boucle `while` est `while(condition)` où `condition` est une expression boléenne qui est évalué au début de chaque itération. Les instructions de la boucle sont exécutées tandis que la `condition` s'évalue à `true` (0 ou plusieurs fois). La condition de la boucle `do...while(condition)` est évaluée à la fin de chaque itération. Les instructions sont exécutées de nouveau si la condition évalue à `false` (donc au moins 1 itération est faite). L'instruction `for` est composé de trois blocs de contrôle d'itération : initialisation, condition et itération dans le syntaxe
`for(initialisation;condition;iteration) instructions` Le code d'initialisation peut contenir la définition de variables locales visibles dans les instructions d'itération. La condition est évaluée au début de chaque itération. Le code `iteration` est exécuté à la fin de chaque itération. Chacun des trois blocs peut être vide, donc on peut spécifier une boucle sans condition `for(int i=0; ; i++)`, sans itération `for(int i=0; i<10; )` ou même `for(;;)`.

La complétion normale de boucles peut être précipitée par `break` (sort de l'itération) et `continue` (saute à la fin des intructions d'itération et continuation par l'itération suivante si nécessaire).

Exercice 1.12. *Calculer la somme de valeurs dans un tableau* `double[] A`.

Exercice 1.13. *Trouver la valeur minimale dans un tableau* `double[] A`.

Exercice 1.14. *Trouver la deuxième plus grande valeur dans un tableau* `double[] A` *de taille au moins 2. (On peut supposer que toutes les valeurs sont distinctes.)*

Exercice 1.15. *Compter le nombre de valeurs* `true` *dans un tableau booléen.*

Exercice 1.16. *Soit* `int[] A` *un tableau de nombres entiers. Écrire le code pour construire un tableau* `int[] B` *qui contient chaque valeur distincte de* `A` *exactement une fois.* ***Indice :*** *utilisez un tableau auxilaire* `boolean[] deja_vu` *où* `deja_vu[i]` *est vrai si la valeur* `A[i]` *apparaît parmi* `A[0]`, ...`A[i-1]`.

Exercice 1.17. *Étant donné deux matrices* `double[][] A` *et* `double[][] B` *(tableaux de deux dimensions de la même taille), calculer la matrice* `double[][] C` *de la même taille où* $C[i][j] = A[i][j] + B[i][j]$.

Exercice 1.18. *Écrire le code pour calculer* F_n*, le* n*-ème nombre Fibonacci. Par définition* $F_0 = F_1 = 1$ *et* $F_{i+1} = F_i + F_{i-1}$.

Exercice 1.19. *Écrire le code pour calculer le logarithme d'une variable* `int x` *à la base* `int a`*, arrondi au plus proche entier inférieur. N'utiliser que de l'arithmétique sur entiers.* ***Indice*** *: utiliser une boucle dans laquelle* x *est divisé par* a *jusqu'à ce qu'on arrive à 1 (dont le logarithme est 0, bien sûr).*

Exercice 1.20. *Écrire le code pour calculer la racine de* `double x` *avec la méthode de Newton. L'idée est de calculer une série d'appoximations* $y_1, y_2, y_3, \ldots$ *jusqu'à ce qu'on arrive assez proche de* $\sqrt{x}$ *:* $|y_n - \sqrt{x}| < \epsilon$*. Ici,* ϵ *est un petit nombre qui définit la précision souhaitée. La règle est de commencer avec un* $y_0 > 0$ *quelconque (p.e.,* $y_0 = x/2$*) et de calculer* $y_{i+1} = \frac{y_i + x/y_i}{2}$*.* ***Indice*** *: utiliser une boucle* `do-while` *ou* `while` *dans laquelle la variable* `y` *est utilisée pour calculer les* y_i*s ; la condition de finir l'itération est d'avoir* $\frac{|y_n - y_{n-1}|}{y_{n-1}} \leq \epsilon$.

Exercice 1.21. *Implémenter la méthode d'Ératosthène pour trouver des nombres premiers inférieurs à un seuil* N*. On forme une table avec tous les entiers naturels compris entre* 2 *et* N *et on raye les uns après les autres, les entiers qui ne sont pas premiers de la manière suivante : dès que l'on trouve un entier qui n'a pas encore été rayé, il est déclaré premier, et on raye tous les autres multiples de celui-ci. Uilisez un tableau* `boolean[] premier` *de taille* N *dans votre implantation, où chaque entrée contient la valeur* `true` *au début. En débutant avec l'indice* $i = 2$*, vérfiez si* `premier[i]` *est vrai. Si oui, alors affichez* i*, c'est un nombre premier. Après, marquez chaque case* $2i$*,* $3i$*, ... comme faux, et continuez avec le* i *suivant dans la recherche (*`i++`*). Notez que les multiples de* i *sont déjà rayonnés si* `premier[i]` *est faux.*

Afficher chaque nombre premier en une ligne sur `System.out`.

```
2
3
5
```

```
7
11
13
...
```

1.3.2 Exigences

- ⋆ Blocs `if–then–else`, et `switch`. Imbrication de multiples `if`s. Complétion précipitée par `break`.
- ⋆ Boucles `while`, `do-while` et `for`. Complétion précipitée par `break` et `continue`.

1.4 Méthodes statiques

1.4.1 Arguments et valeur retournée

Le type de la valeur retournée doit être spécifié pour chaque méthode. Si aucune valeur n'est retournée, alors on utilise le mot-clé `void`. Tous les types (incluant les types primitifs, tableaux, objets, tableaux d'objets, etc.) sont permis pour la valeur retournée. L'exécution de la méthode est completée par le mot-clé `return`. La liste d'arguments peut être vide (aucun argument), ou non : on utilise les arguments d'une méthode quelconque dans le code de la méthode comme des variables locales (donc, pour accéder à une variable statique avec le même nom, il faut utiliser le nom qualifié `Classe.variable` parce que l'argument le cache). Si un tableau ou un objet quelconque est passé comme argument, la méthode reçoit la référence à cet objet, et peut changer son état (p.e., changer les éléments du tableau). Une méthode de la classe (méthode statique) est déclarée avec le mot-clé `static`.

La **signature** d'une méthode est composée de son nom et la liste des types des arguments (qui peut être vide). Donc rien n'empêche la définition de plusieurs méthodes avec le même nom si les arguments ont de types différents. Le type de la valeur retournée peut être différent entre des méthodes avec signatures différentes. Exemple : `Math.max(int,int)` retourne un `int` et `Math.max(double,double)` retourne un `double`.

1.4.2 Contexte de classe

Tout comme avec les variables statiques, les méthodes statiques peuvent être utilisées par leur **nom qualifié** dans le syntaxe `Classe.methode(...)`. La partie avant la période donne le **contexte** de l'appel à la méthode. De cette façon, on peut utiliser les méthodes de plusieurs classes en même temps.

La classe `Math` fournit des méthodes statiques importantes pour faire du calcul numérique avec des fonctions comme e^x, $\ln x$, y^x, $\sin x$, etc.

Les classes `Byte`, `Short`, `Integer`, `Long`, `Double` et `Float` définissent des méthodes de traduction d'une chaîne de caractères vers un type simple.

Méthode	Type de valeur retournée
`Byte.parseByte`	`byte`
`Short.parseShort`	`short`
`Integer.parseInt`	`int`
`Long.parseLong`	`long`
`Float.parseFloat`	`float`
`Double.parseDouble`	`double`

(Toutes les méthodes prennent un `String` comme argument.) Cela permet dans la méthode `main`, par exemple, la communication avec la ligne de commande à l'aide de l'argument `String[]`.

```
public class TroisArguments
{
    public static void main(String[] args)
    {
        int x0    = Integer.parseInt(args[0]);
        double x1 = Double.parseDouble(args[1]);
        // pas de String.parseString() ...
        String x2 = args[2];
    }
}
```

Après compilation, l'exécution est faite p.e. par
`java TroisArguments 10 0.07 "Bonjour tristesse"`

1.4.3 Droits d'accès `public/private`

Si une méthode ou une variable est déclarée `public`, alors elle peut être utilisée dans n'importe quelle autre classe avec le nom qualifié dans le syntaxe `Classe.methode(...)` ou `Classe.variable`. Par contre, si elle est déclarée

`private`, alors elle ne peut être utilisée que dans la même classe. (Si on essaye d'y accéder par son nom qualifié dans une autre classe, le code de cette autre classe ne compile pas.)

Exercice 1.22. *Écrire une méthode statique privée qui calcule la solution de l'équation quadratique $ax + b = c$. Les arguments de la méthode sont a, b, c (comme des* `doubles`*), et la valeur retournée doit être un tableau de* `double` *avec les solutions pour x (tableau de taille 0,1 ou 2 selon le nombre de solutions.)* ***Rappel*** *: la solution est $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2}$.*

Exercice 1.23. *Écrire du code exécutable qui prend deux arguments x et y (nombres réels) à la ligne de commande et affiche la valeur de x^y.*

Exercice 1.24. *Écrire une méthode statique privée* `moyenne` *qui calcule la moyenne des éléments d'un tableau* `double[]`*. Écrire le code de* `main` *qui construit un tableau* `double[]` *contenant les arguments à la ligne de commande, appelle* `moyenne` *et affiche la valeur retournée.*

Exercice 1.25. *Écrire une méthode qui calcule la solution x de $f(x) = e^x + x = y$ où $y \geq 1$ est l'argument de la méthode. La solution est approximée par une procédure itérative utilisant une intervalle $[a, b]$ rétrécissante qui contient la solution x. Initialiser avec $a = 0$ et $b = \ln y$ (on a $a = 0 \leq x < \ln y = b$), et vérifier en chaque itération si à $c = (a + b)/2$ on a $f(c) < y$ ou $f(c) > y$. Si $f(c) < y$, lors remplacer a par c. Si $f(c) > y$, alors remplacer b par c. Si $f(c) = y$, alors retourner c parce que c'est la solution. Terminer la boucle si a et b sont assez proches (disons, $|a - b| < \epsilon = 10^{-7}$), et retourner $(a + b)/2$ comme la solution.*

Exercice 1.26. *Écrire une méthode statique qui calcule la balance d'une carte de crédit après un nombre de mois quand on paie le même montant chaque mois. Les arguments de la méthode sont la balance initiale (*`double balance`*), taux d'intêret mensuel (*`double taux`*), paiement mensuel (*`paiement`*), et le nombre de mois (*`int mois`*). Si la balance au début d'un mois est b, alors la balance à la fin du mois est $b + \max\{b, 0\} \times (1 + t) - p$, avec taux d'intêret t et paiement p.*

Exercice 1.27. *Écrire une méthode statique qui calcule le nombre de mois pour repayer la balance d'une carte de crédit comme en Exercice 1.26. Arguments de la fonction : balance initiale (*`double balance`*), taux d'intêret mensuel (*`double taux`*), paiement mensuel (*`paiement`*) ; la valeur retournée est le nombre de mois pour arriver à une balance non-négative.*

1.4.4 Exigences

- ⋆ Définition d'une méthode statique.
- ⋆ Appel d'une méthode statique : dans la même classe, dans d'autres classes.
- ⋆ Méthodes de traduction de chaînes de caractères en types numériques : `Integer.parseInt`, `Double.parseDouble`.
- ⋆ Méthodes de base de la classe `Math` :`max`, `min`, `sqrt`, `pow`, `log`, `sin`, `cos`, `atan`, `exp`.
- ⋆ Appel d'une méthode en utilisant le contexte de classe (nom qualifié)
- ⋆ Méthodes multiples avec le même nom, signatures
- ⋆ Contrôle d'accès par `private` et `public`

1.5 Solutions aux exercices de chapitre 1

Exercice 1.1 Ouvrir le fichier `ProgrammeSimple.java` avec Emacs (p.e., écrire `emacs ProgrammeSimple.java` à la ligne de commande). Copier le code exactement comme montré ici. Sauvegarder le fichier (tapez Ctrl-X Ctrl-S dans Emacs).

Exercice 1.2 Faire `javac ProgrammeSimple.java` suivi par `java ProgrammeSimple`.

```
% javac ProgrammeSimple
% java ProgrammeSimple
C'est mon programme simple.
```

Exercice 1.3 Écrire le fichier, qui s'appelle, disons, `Verlaine.java` :

```
public class Verlaine
{
    public static void main(String[] args)
    {
        System.out.println("Les sanglots longs");
        System.out.println("des violons");
        ...
        System.out.println("monotone.");
    }
}
```

Altérnativement, on peut utiliser le caractère `\n` pour dénoter la fin-de-ligne en une chaîne :

```
System.out.println("Les sanglots longs\ndes violons\n...");
```

Compiler et exécuter :

```
% javac Verlaine.java
% java Verlaine
Les sanglots longs
des violons
...
monotone.
```

Exercice 1.4

```
public class SecondsAnnee
{
    public static void main(String[] args)
    {
        System.out.println(365*24*60*60);
    }
}
```

Exercice 1.5

```
double a = 6.0221417E23;
char b='\t';
long c=2008L;
bool d=false;
```

Exercice 1.6 Le 20e siècle est la période entre le 1er janvier 1901, et le 21e décembre 2000 (selon le calendrier grégorien). Pendant cette période, il y avait 25 années bissextiles avec 366 jours.

```
public class SecondsSiecle
{
    public static void main(String[] args)
    {
        System.out.println((100L*365L+25L))*24L*60L*60L);
    }
}
```

Notez qu'on doit forcer le calcul avec type `long` parce que le résultat dépasse le maximum de types `int`.

Exercice 1.7

```
public class ClasseExample
{
    public static int nombre=9;
    ...
    public static void main(String[] args)
    {
        int nombre = 12;
        ...
        int resultat = nombre + ClasseExample.nombre;
    }
}
```

Exercice 1.8

```
String[] les_noms_des_jours
    = {"dimanche", "lundi", "mardi",
       "mercredi", "jeudi", "vendredi",
       "samedi"};
int jour;
...

String nom_du_jour = les_noms_des_jours[jour-1];
```

Exercice 1.9 `(int)(z+0.5);`.

Exercice 1.10 On a besoin de calculer

$$\frac{\lfloor 100z + \frac{1}{2} \rfloor}{100},$$

où $\lfloor \cdots \rfloor$ dénote la troncation. Une solution :

```
double z;
...
double resultat = 0.01*((int)(100.0*z+0.5));
```

Exercice 1.11

```
int jour;
...
String nom_du_jour;
switch (jour)
{
    case 1: nom_du_jour = "dimanche"; break
    case 2: nom_du_jour = "lundi"; break;
    ...
}
```

Exercice 1.12

```
double[] A;
...
double somme=0.0;
for (int i=0; i<A.length; i++)
{
    somme += A[i];
}
// maintenant la variable somme contient la somme
// de toutes les valeurs A[i]
// --- meme si A est de longueur 0
```

Exercice 1.13 Une solution qui ne marche que pour des tableaux de taille positive.

```
double[] A;
...
double max = A[0];
for (int i=1; i<A.length; i++)
    if (A[i]>max)
        max = A[i];
```

Une solution qui utilise la valeur par défaut $-\infty$ qui vaut pour tableaux de taille 0.

```
double max=Double.NEGATIVE_INFINITY;
for (int i=0; i<A.length; i++)
    if (A[i]>max)
        max = A[i];
```

Exercice 1.14

```
double[] A;
...
double max = A[0];
double max2=Double.NEGATIVE_INFINITY;
for (int i=1; i<A.length; i++)
    if (A[i]>max)
    {
        max = A[i];
        max2 = max;
    } else if (A[i]>max2)
        max2 = A[i];
// maintenant max2 est la deuxieme plus grande valeur
```

Exercice 1.15

```
boolean[] A;
...
int nb_vrai=0;
for (int i=0; i<A.length; i++)
    if (A[i]) nb_vrai++;
```

Notez qu'il est superflueux d'écrire `A[i]==true` comme condition.

Exercice 1.16

```
int[] A;
...
boolean[] deja_vu = new boolean[A.length];
for (int i=0; i<A.length; i++)
{
    deja_vu[i]=false; // valeur par défaut
    for (int j=0; j<i; j++)
        if (A[j]==A[i])
            deja_vu[i]=true;
}
// compter le nombre d'éléments distincts
int nb_distincts = 0;
for (int i=0; i<A.length; i++)
    if (!deja_vu[i]) nb_distincts++;
int[] B =new int[nb_distincts];
int j=0; // j sera l'indice dans le tableau B
for (int i=0; i<A.length; i++)
{
    // i est l'indice dans le tableau A
    if (!deja_vu[i])
    {
        B[j] = A[i];
        j++; //
    }
}
```

Ou une solution une peu plus efficace.

```
int[] A;
...
boolean[] deja_vu = new boolean[A.length];
int nb_distincts = 0;
for (int i=0; i<A.length; i++)
{
    deja_vu[i]=false; // valeur par défaut
    for (int j=0; j<i; j++)
        if (A[j]==A[i])
        {
            deja_vu[i]=true;
            break;
        }
    if (!deja_vu[i]) nb_distincts++;
}
int[] B =new int[nb_distincts];
for (int i=0, j=0; j<nb_distincts; i++)
    if (!deja_vu[i])
    {
        B[j] = A[i];
        j++;
    }
```

Exercice 1.17

```
double[][] A;
double[][] B;
...
int nb_rangees = A.length;
int nb_colonnes = A[0].length;
double[][] C = new double[nb_rangees][nb_colonnes];
for (int i=0; i<nb_rangees; i++)
    for (int j=0; j<nb_colonnes; j++)
        C[i][j] = A[i][j]+B[i][j];
```

Exercice 1.18 Solution avec un tableau.

```
int n;
...
int[] F = new int[n+1]; // on a besoin de F[0]...F[n]
F[0] =1;
if (n>0) F[1]=1;
for (int i=2; i<=n; i++)
    F[i] = F[i-2]+F[i-1];
// maintenant F[n] est le n-eme nombre Fibonacci.
```

Solution sans tableau.

```
int n;
...
double
int F_1 = 0; // pour stocker F[i-2]
int F = 0; // pour stocker F[i-1]
for (int i=2; i<=n; i++)
{
    int F_nouveau = F+F_1;
    F_1 = F;
    F = F_nouveau;
}
// maintenant F est le n-eme nombre Fibonacci.
```

Exercice 1.19

```
int x;
int a;
...
int logx = 0;
while (x>1)
{
    x /= a;
    logx++;
}
```

Exercice 1.20

```
double x;
...
double epsilon = 1e-7; // précision souhaitée
double y = x/2.0; // c'est y_0; on suppose x>0
double diff;
do
{
    double y_nouveau = 0.5*(y + x/y);
    diff = (y_nouveau - y) / y;
    y = y_nouveau;
} while (diff > epsilon || -diff > epsilon);
// maintenant y est notre approximation de la racine de x
```

Exercice 1.21

```
int N;
...
boolean[] premier = new boolean[N];
for (int i=2;i<N; i++) premier[i]=true;
for (int i=2; i<N; i++)
    if (premier[i])
    {
        System.out.println(i);
        for (int j=2*i; j<N; j+=i)
            premier[j]=false;
    }
```

Exercice 1.23

```
public static main(String[] args)
{
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    double z = Math.pow(x,y);
    System.out.println(z);
}
```

Exercice 1.24

```
private static double moyenne(double[] x)
{
    double m = 0.0;
    for (int i=0; i<x.length; i++)
        m += x[i];
    return m/x.length;
}
public static void main(String[] args)
{
    double[] A = new double[args.length];
    for (int i=0; i<args.length; i++)
        A[i] = Double.parseDouble(args[i]);
    System.out.println(moyenne(A));
}
```

(Notez l'affichage quand on exécute le code sans arguments : `NaN`. C'est «Not a Number», qui est en fait la constante `Double.NaN`, et est obtenu comme le résultat de division par 0, logarithme d'un nombre non-positif, etc. Sans arguments, `x.length` est 0.)

Exercice 1.22

```
private static double[] solutionQ(double a, double b, double c)
{
    double v = b*b-4.0*a*c;
    double[] x  = null;
    if (v<0.0)
        x = new double[0]; // pas de solution
    else if (v==0.0)
    {
        x = new double[1];
    x[0] = -b/(2.0*a);
} else
{
        x = new double[2];
        double w = Math.sqrt(v);
        x[0] = (-b + w) / (2.0*a);
        x[1] = (-b - w) / (2.0*a);
    }
    return x;
}
```

Exercice 1.25

```
private static double EPSILON = 1e-7;
public static double solutionF(double y)
{
    double a = 0.0;
    double b = Math.log(y);
    while (b-a>EPSILON)
    {
        double c = (a+b)/2;
        double f = Math.exp(c)+c;
        if (f<y) a = c;
        if (f>y) b = c;
        if (f==y) return c;
    }
    return (a+b)/2.0;
}
```

Exercice 1.26

```
public static double balanceFutur
    (double balance, double taux, double paiement, int mois)
{
    for (int i=0; i<mois; i++)
    {
        double interet = Math.min(balance,0.0)*(1.0+taux);
        balance += interet - paiement;
    }
    return balance;
}
```

Exercice 1.27

```
public static int balanceMois
    (double balance, double taux, double paiement)
{
    int mois = 0;
    for (; balance>=0.0; mois++)
    {
        double interet = Math.min(balance,0.0)*(1.0+taux);
        if (interet >= paiement) // évite la boucle infinie
            return Integer.MAX_VALUE;
        balance += interet - paiement;
    }
    return mois;
}
```

(Il n'est pas nécessaire de performer le test `interet>=paiement` dans chaque itération : une telle modification n'est pas montrée ici.)

Chapitre 2

Classes et objets

Dans le paradigme de programmation orientée objet, on groupe les informations et tâches dans des objets. En Java, chaque objet appartient à une classe. Les classes définissent les *méthodes* par lesquelles on peut accéder à l'information associée avec des objets, ou demander un objet à performer une tâche quelconque. Les classes pour les tableaux (qui sont les objets) sont définies automatiquement, mais pour d'autres objets, il faut écrire le code explicitement.

2.1 Objets

En Java, chaque objet appartient à une classe spécifique. La création d'un nouvel objet d'une classe quelconque est **l'instanciation** de la classe. Les objets sont manipulés par référence : la valeur d'une variable qui est de type d'une classe est un emplacement en mémoire où l'objet de cette classe est stocké physiquement. Il y a deux possibilités pour utiliser une variable ou une méthode de l'objet :

- dans une méthode du même objet, constructeur de la même classe, ou l'initialisation d'une autre variable d'objet, on peut juste utiliser le nom (si aucune variable locale ne le cache pas)
- n'importe où, à l'aide du nom qualifié en fournissant un contexte par une référence : `reference.variable` ou `reference.methode(...)`.

Typiquement, la référence dans le contexte est une variable du type approprié, mais on peut également utiliser n'importe quelle expression s'évaluant au bon type (il faut mettre l'expression en parenthèses avant la période '`.`') : `(new String("voici")).length()` donne 5, ou `"voici".substring(0,3).substring(1,2)` donne `"oi"`.

Le mot-clé **`this`** est une référence à l'objet exécutant le code. Entre autres, il permet la désambiguïsation de noms de variables (quand une variable locale porte le même nom qu'une variable d'objet). Il est également possible d'utiliser la référence `this` comme valeur retournée, argument d'une méthode, ou dans une expression quelconque mais elle ne peut pas être modifiée.

La référence à un objet non-existant est fait par le mot-clé `null` ; ce dernier est la valeur initiale d'un objet par défaut.

2.1.1 Variables de l'objet

Un objet est associé avec certains attributs, définissant son état à des points différents de l'exécution d'un programme. Pour stocker de l'information associée avec un objet, on utilise des **variables d'objet**. Les variables d'objet sont déclarées comme les variables de classe, mais sans le mot-clé `static`.

```
public class Banane
{
    // variable de la classe qui vaut pour toutes les instances
    private static String couleur = "Jaune";

    // variable de l'objet ayant de valeurs différentes entre objets
    private int taille;
}
```

On peut accéder à une variable de l'objet en utilisant son nom qualifié, en construisant un contexte, fourni par la référence à l'objet. Dans une méthode de l'objet ou dans son constructeur, on peut omettre le contexte et utiliser seulement le nom de la variable.

```
public class Banane
{
    private int taille;

    public Banane(int T)
    {
        taille = T; // contexte est clair
    }
    public int montreTaille()
    {
        return taille; // contexte est claire
    }
    public boolean memeTaille(Banane autreBanane)
    {
     return taille == autreBanane.taille; // contexte pour le 2e
    }
    public void mordre(int taille)
    {
        this.taille -= taille;
        // contexte "this" nécessaire pour distinguer entre
        // la variable de l'objet et l'argument de la méthode
    }
}
```

2.1.2 Méthodes

Les méthodes d'un objet sont définies comme les méthodes de la classe, mais en omettant le mot-clé `static`. La signature d'une méthode ne considère pas si la méthode est statique ou non, donc on ne peut pas avoir deux méthodes avec le même nom et types d'arguments même si l'une est statique et l'autre ne l'est pas.

2.1.3 Constructeurs

Pour instancier une classe, il faut appeler son constructeur en utilisant le syntaxe `new Classe(...)`. Une classe peut avoir plusieurs constructeurs qui ont de types d'arguments différents. Si on ne définit pas un constructeur dans le code pour la classe, Java fournit automatiquement un constructeur par défaut, sans arguments. Si la classe définit au moins un constructeur explicitement, la classe n'a pas de constructeur par défaut. Notez qu'on a le droit de redéfinir le constructeur sans argument.

Il est possible d'appeler un constructeur à partir d'un autre constructeur en se servant dy syntaxe `this(...)`. Dans ce cas-là, l'appel doit être la première

instruction. Notez que `this()` est possible seulement dans la situation de passer l'exécution d'un constructeur à un autre (p.e., on ne peut pas utiliser `this()` dans une méthode régulière.)

```
public class Banane
{
    public Banane(int T)
    {
        taille = T; // contexte est clair
    }
    public Banane()
    {
     this(2); // taille standarde d'une banane
    }
}
```

Exercice 2.28. *Compléter les méthodes d'accès aux variables de l'objet dans le fragment de code ci-dessous.*

```
public class Rectangle
{
    private int largeur;
    private int hauteur;
    public void soitLargeur(int largeur)
    {
        // À COMPLÉTER: affectation de largeur
    }
    public int montreLargeur()
    {
        // À COMPLÉTER: retourne la largeur
    }
    public void setHauteur(int hauteur)
    {
        // À COMPLÉTER: affectation de hauteur
    }
    public int getHateur()
    {
        // À COMPLÉTER: retourne la hauteur
    }
```

Exercice 2.29. *Écrire une classe pour représenter des fonctions quadratiques* $f(x) = ax^2+bx+c$. *Les trois attributs d'un objet dans cette classe sont* a, b, c. *Voici la signature du constructeur :* `public Quadratique(double a, double b, double c)`.

La classe doit définir la méthode dynamique `public void add(Quadratique q)` *qui fait l'addition de deux fonctions (donc change les attributs de* `this`*) : évidemment,*

$$(a_1x^2 + b_1x + c_1) + (a_2x^2 + b_2x + c) = (a_1 + a_2)x^2 + (b_1 + b_2)x + (c_1 + c_2).$$

Écrire la méthode `public double[] zeros()` *qui calcule la solution à* $ax^2 + bx + c = 0$ *et retourne le tableau des* x*s pour lesquels l'équation est vraie. S'il n'ya pas de solution, alors il faut retourner un tableau de taille 0. (Attention :* a *peut être 0, mais on peut supposer que si* $a = 0$*, alors* $b \neq 0$*).*

Exercice 2.30. *Écrire une classe pour représenter des objets célestes dans le système solaire. En un moment donné, un objet céleste est caractérisé par sa position, vitesse et accélération. Pour simplifier le problème, on travaille en deux dimensions. Donc les attributs d'un objet sont positions X et Y, vitesse X et Y, et accélération X et Y (tous de type* `double`*). Le constructeur est appelé avec la position et vitesse initiales. Écrire les méthodes d'accès aux variables de l'objet comme* `public double getPositionX()`*, etc. Écrire une méthode* `public void delta(double t)` *qui recalcule la positon et la vitesse de l'objet après un temps* t *:*

$$x(t) = x + tv_x \quad y(t) = y + tv_y \quad v_x(t) = v_x + ta_x \quad v_y(t) = v_y + ta_y.$$

Chaque fois que la position change, il faut recacluler l'accéleration (utiliser une méthode privée). Selon la loi de gravitation, l'accélération totale de l'objet vers le Soleil est calculée comme

$$a = \frac{1.3275 \cdot 10^{11}}{x^2 + y^2}.$$

(unité pour x, y *est* km*, pour la vitesse est* $\mathrm{km/s}$ *et pour l'accélération est* $\mathrm{km/s^2}$*).*

Pour calculer les coordonnées X et Y de l'accélération, faire

$$a_x = -\frac{x}{\sqrt{x^2 + y^2}}a \qquad a_y = -\frac{y}{\sqrt{x^2 + y^2}}a.$$

(Maintenant, il est possible de simuler le mouvement d'un objet autour du Soleil : utiliser une boucle infinie `while(true)` pour appeler `delta` et afficher (x, y) après chaque appel. Pour la Terre, p.e., une position initiale est $(x, y) = (147000000, 0)$ et $(v_x, v_y) = (0, 30.3)$. L'argument t donne la résolution pour votre simulation en seconds : essayez $t = 86400$ et d'autres valeurs plus petites.)

2.1.4 Exigences

- `private` et `public` pour méthodes, variables et constructeurs
- Différences entre variables statiques et non-statiques (variables d'objet) ; méthodes d'accès et d'affectation pour manipuler des variables d'objet privées
- Différences entre méthodes statiques et non-statiques (méthodes de l'objet)
- Utilisation d'une variable d'objet ou une méthode d'objet : dans la même classe, dans d'autres classes
- Instanciation d'une classe
- Nom qualifié, manipulation de références, référence `null`
- `this` comme référence à l'objet actif
- `this` comme appel à un autre constructeur dans la même classe

2.2 Solutions aux exercices de chapitre 2

Exercice 2.28

```
public class Rectangle
{
    private int largeur;
    private int hauteur;
    public void soitLargeur(int largeur)
    {
        this.largeur = largeur;
    }
    public int montreLargeur()
    {
        return largeur;
    }
    public void setHauteur(int hauteur)
    {
        this.hauteur = hauteur;
    }
    public int getHateur()
    {
        return hauteur;
    }
}
```

Exercice 2.29 En utilisant la solution de l'Exercice 1.22 avec la méthode `solutionQ` :

```
public class Quadratique
{
    public Quadratique(double a, double b, double c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    private double a;
    private double b;
    private double c;

    public void add(Quadratique q)
    {
        a += q.a;
        b += q.b;
        c += q.c;
    }

    public double[] zeros()
    {
        if (a==0)
        {
            double[] solution = new double[1];
            solution[0] = -c/b;
            return solution;
        } else
            return solutionQ(a,b,c);
    }
```

Exercice 2.30

```
public class ObjetCeleste
{
    private static double CONSTANTE_GRAVITATIONNELLE = 1.3275e11;
    public ObjetCeleste(double x, double y, double vx, double vy)
    {
        this.x  = x;
        this.y  = y;
        this.vx = vx;
        this.vy = vy;
        calculeAcceleration();
    }

    private double x;
    private double y;
    private double vx;
    private double vy;
    private double ax;
    private double ay;

    public double getX(){return x;}
    public double getY(){return y;}

    private void calculeAcceleration()
    {
        double r = Math.sqrt(x*x+y*y);
        double a = CONSTANTE_GRAVITATIONNELLE/(r*r);
        ax = -x*a/r;
        ay = -y*a/r;
    }

    public void delta(double t)
    {
        x += vx*t;
        y += vy*t;
        vx += ax*t;
        vy += ay*t;
        calculeAcceleration();
    }
```

Pour voir ce que cela donne, p.e. :

```
public class ObjetCeleste
{
...
  public static void main(String[] args)
  {
      double x = Double.parseDouble(args[0]);
      double y = Double.parseDouble(args[1]);
      double vx = Double.parseDouble(args[2]);
      double vy = Double.parseDouble(args[3]);

      ObjetCeleste planete = new ObjetCeleste(x,y,vx,vy);
      for (int i=0; i<400; i++)  // 400 jours
      {
          System.out.println(planete.getX()+"\t"+planete.getY());
          planete.delta(86400.0); // 86400s = un jour
      }
  }
}
```

Maintenant, `java ObjetCeleste 147000000 0 0 30.3` devrait donner la position de la Terre chaque jour — avec des erreur numériques (v. Fig. 2.1).

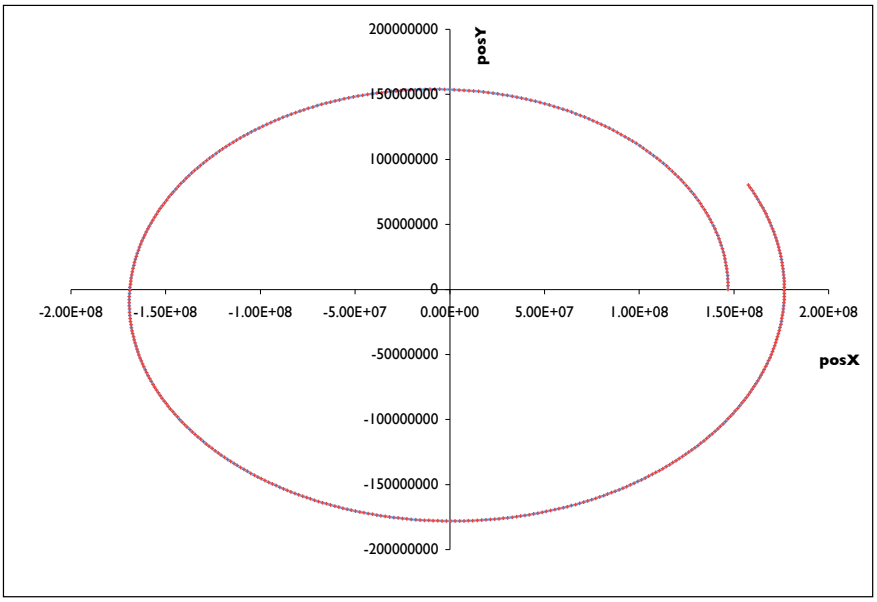

FIG. 2.1 – Résultat de `java ObjetCeleste 147000000 0 0 30.3`. Pour produire cette image : sauvegarder le résultat par rédirection `java ...> resultat.txt` à la ligne de commande, et ouvrir `resultat.txt` en Microsoft Excel.