Master Sciences du Langages — Université de Paris

Introduction à la programmation orientée objet en Java

Guillaume Wisniewski

2 mars 2020

Introduction à la POO & à Java

Disclaimer
You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at :
https://github.com/fmarotta/kaobook
(You are welcome to contribute!)

Publisher
First printed in May 2019 by

The harmony of the world is made manifest in Form and Number, and the heart and soul and all the poetry of Natural Philosophy are embodied in the concept of mathematical beauty.

– D'Arcy Wentworth Thompson

Ce document reprend les principales notions abordées dans le cours « Introduction à la POO en Java » du cours de L3 LI de l'Université de Paris.

Il s'agit d'un *work in progress* : le contenu comporte très certainement de nombreuses erreurs et imprécisions et sera amené à changer fréquemment au cours du semestre. N'hésitez pas à me faire part de toute remarque, commentaire ou correction.

Version du 2 mars 2020

Table des matières

Table des figures

Liste des tableaux

Programmation impérative en java

1

1.1 Compilation et exécution d'un programme java

Le listing 1 donne un exemple du code source d'un programme java. Ce code source décrit une *classe* qui correspond à une *unité de compilation*, c'est-à-dire à l'ensemble des instructions nécessaires à l'exécution d'un programme. Nous verrons, au chapitre 3, que la notion de classe est ambiguë en java et qu'il existe d'autres type de classes.

```java
public class FirstProgram {

    public static void main(String[] s) {
        System.out.println("Bonjour les amis.");
    }

}
```

Listing 1 – `Hello Word` en java.

Le java est un langage *compilé* : le code source doit être « traduit » en un code objet directement exécutable par un ordinateur. Les premiers compilateurs généraient du code en langage machine (une suite de bits) qui était directement interprété par le processeur de l'ordinateur. Le compilateur java produit du *byte code*, une représentation binaire intermédiaire. Cette représentation doit être exécutée par une *machine virtuelle* qui fait abstraction du système d'exploitation ou de la machine : un même programme java peut être exécuté sur un ordinateur, un téléphone voire une machine à laver.

Lors de la compilation, le compilateur réalise une analyse globale (la totalité du code source est accessible et peut être analysée) ce qui lui permet de détecter plus facilement et plus rapidement d'éventuelles erreurs : contrairement aux langages interprétés (comme le python) dans lesquels les instructions sont exécutées au fur et à mesure que celle-ci sont lues dans le fichier source et, par conséquent, dans lesquels les erreurs de syntaxe ne peuvent être détectées qu'au moment où une ligne est exécutée, un programme java ne pourra être compilé (et donc exécuté) que s'il ne contient aucune erreur de syntaxe. La compilation permet également d'optimiser le code en cherchant à améliorer la vitesse d'exécution de celui-ci ou son occupation mémoire

Pour exécuter le programme du listing 1, il est nécessaire de réaliser deux étapes :

— *compiler* le code source à l'aide de l'instruction :

```
> javac FirstProgram.java
```

La commande `javac` (pour Java Compiler) permet d'appeler le compilateur Java qui va transformer le code source (contenu dans

un simple fichier texte avec l'extension `.java`) en fichier contenant le byte code portant l'extension `.class` qui pourra être exécuté par la machine virtuelle. La compilation échoue si la syntaxe java n'est pas respectée : comme illustré à la figure 1.1 le compilateur renvoie une erreur dès qu'il détecte une erreur de syntaxe.

— appeler la machine virtuelle à l'aide de la commande :

```
> java FirstProgram
```

La commande `java` permet d'appeler la JVM (Java Virtual Machine). Il prend en paramètre le nom d'une classe (contenue dans un fichier `.class`) et exécute celle-ci.

En pratique, les programmes java sont composés de plusieurs dizaines voire plusieurs centaines de classes et compiler manuellement des fichiers comme dans les exemples précédents est impossible : le développement se fait généralement dans un EDI (environement de développement intégré) qui prend en charge la compilation des différents fichiers nécessaires.

```
> javac FirstProgram.java
FirstProgram.java:4: error: ';' expected
        System.out.println("Bonjour les amis.")
                                               ^
1 error
```

Figure 1.1 – Exemple d'erreur de compilation : la ligne 4 du fichier `FirstProgram.java` ne se termine pas par un point-virgule.

La distinction entre les étapes de compilation et d'exécution n'est pas toujours visible : dans des environnements de développement intégré comme `eclipse`, le code java est compilé à la volée (c'est-à-dire au fur et à mesure qu'il est saisi) et il peut être exécuté directement sans avoir besoin d'appeler le compilateur explicitement.

Lors de l'exécution d'un programme java, la machine virtuelle, va chercher le *point d'entrée* du programme et exécuté la première ligne de celui-ci. En java, le point d'entrée est défini par la fonction `main`[1] . Si la machine virtuelle ne parvient pas à identifier le point d'entrée du programme (par exemple, parce que le nom de la fonction n'est pas correctement orthographié ou que la définition de celle-ci ne commence pas par les mots clés `public static void`), elle génère une erreur, comme illustrer à la figure 1.2.

1: Nous verrons à la section 1.4 comment définir une fonction.

```
> java FirstProgram
Erreur : la méthode principale est introuvable dans la classe
FirstProgram, définissez la méthode principale comme suit :
   public static void main(String[] args)
ou une classe d'applications JavaFX doit étendre
   javafx.application.Application
```

Figure 1.2 – Erreur obtenue lors de l'exécution d'une classe ne contenant pas de point d'entrée.

Exception Il y a donc deux types d'erreur en java : les erreurs de compilations, détectées par le compilateur avant l'exécution du programme et les exceptions qui ont lieu *pendant* l'exécution du programme.

La compilation permet de détecter plusieurs types d'erreur : erreur de syntaxe, code non exécutable, variable non utilisée, ... Cependant, certaines erreurs n'apparaissent qu'à l'exécution du programme. C'est par exemple, lors de la divsion par une variable `b`, une erreur doit être détectée lorsque `b` est nulle. Cette erreur ne peut être détectée que lors de l'exécution, lorsque la valeur de `b` est connue.

En java, les erreurs détectées lors de l'exécution donne lieu à des exceptions, comme celle qui est décrite à la figure 1.3. Une exception comporte trois informations :

— le nom de l'exception (ici : `ArithmeticException`);
— un éventuel message d'erreur qui complète la description de l'erreur (il s'agit dans ce cas d'une division par 0);
— une *trace d'exécution* qui permet à la fois d'identifier à quelle ligne l'erreur c'est produite (ici la ligne 5) mais également les appels de fonctions successifs qui expliquent pourquoi la ligne ayant causé l'erreur a été exécutée : ici l'erreur se situe à la ligne 5 de la fonction `divisionParZero` qui a été appelée à la ligne 44 de la fonction `main`.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Prog.divisionParZero(Prog.java:5)
        at Prog.main(Prog.java:44)
```

Figure 1.3 – Exemple d'une exception levée à la suite d'une division par 0.

1.2 Anatomie d'un programme java

Le code du listing 1 permet d'illustrer plusieurs aspects de la syntaxe java : un programme java est composé d'une ou de plusieurs classes, chaque classe étant définie dans un fichier qui lui est propre. Une classe est composée de deux éléments :

— un nom précédé des deux mots clés `public class`;
— d'un bloc de code, identifié par des accolades ({ marque le début du bloc et } sa fin).

Le compilateur java impose que le code source d'une classe soit stocké dans un fichier portant le même nom que la classe : il est nécessaire que le code du listing 1 soit contenu dans un fichier `FirstProgram.java`. Le compilateur détectera une erreur de syntaxe si cela n'est pas le cas.

Une classe est une unité de compilation qui regroupe plusieurs fonctions nécessaires à l'exécution du programme. En attendant d'introduire la syntaxe permettant de définir des fonctions (à la section **??**), l'ensemble du code sera toujours contenu dans la fonction `main` : en java, une classe ne peut pas contenir directement de code, celui-ci doit toujours être contenu dans une fonction qui est elle-même toujours définie à l'intérieur d'une classe (c.-à-d. du bloc de code correspondant à celle-ci).

Par convention les noms de classe doivent toujours être écrits en *Camel Case* : les différents mots composant le nom sont liés sans espace ni ponctuation et en mettant en capitale la première lettre de chaque mot. Il s'agit d'une convention : le compilateur compilera le code d'une classe commençant par une minuscule ou dont les mots sont séparés par des tirets bas *(underscore)*, mais une telle pratique rendra la compréhension du code plus difficile pour les autres programmeurs.

De manière similaire, par convention, les blocs suivent toujours la même mise en forme : l'accolade ouvrante se met à la fin de la ligne où commence le bloc concerné et l'accolade fermante sur une ligne à part, indentée au niveau de l'instruction qui a entraîné l'ouverture du bloc. L'ensemble du code contenu dans le bloc doit être indenté[2] .

Un bloc de code est composée d'une ou de plusieurs instructions séparées par des points-virgules (`;`). Par convention, chaque instruction correspond

Les conventions sont des règles conçues pour faciliter la lecture et la compréhension du code en permettant d'identifier le plus d'éléments possibles « du premier coup d'œil ». Si leur respect n'est pas obligatoire (un programme ne respectant pas les conventions fonctionnera aussi bien qu'un programme les respectant), l'expérience montre que respecter les conventions facilite grandement l'écriture, la maintenance et aident à éviter certaines erreurs. Elles pqermettent également d'identifier les programmeurs expérimentés.

2: L'indentation consiste en l'ajout de tabulations ou d'espaces visant à faciliter l'identification des blocs de code

à une ligne, mais il reste nécessaire de placer un point-virgule à la fin de chaque ligne.

Il est également possible d'inclure des commentaires, soit en faisant précéder une ligne de deux barres obliques *(slash)* `//`, soit en insérant la partie à commenter entre les symboles `/*` (début de commentaire) et `/*` (fin de commentaire). Cette dernière syntaxe permet de réaliser des commentaires sur plusieurs lignes.

1.3 Types et variables

Dans un programme informatique, les variables permettent de nommer des valeurs : lors de l'exécution du programme, la machine virtuelle remplacera chaque apparition du nom d'une variable par la valeur que celle-ci contient. Ainsi lors de l'exécution du programme du listing 2, l'instruction de la ligne 7 va afficher le contenu de la variable `temperature`[3] ; lors de l'exécution de la ligne 10, la machine virtuelle va commencer par remplacer la variable, effectuer la soustraction et afficher le résultat. Au final, la sortie du programme sera :

3: Tout se passe comme si la ligne contenait en fait l'instruction `System.out.println("5778");`

```
Température du soleil (K)
5778
Température du soleil (\degree{}C)
5504.85
```

Convention java pour les variables

Le listing 2 montre les deux principales conventions généralement appliquées pour nommer et manipuler des variables en java : les noms de variables sont systématiquement écrits en *camel case* et commencent par une minuscule ; l'opérateur d'affectation (=), comme tous les opérateurs binaires, est précédé et suivi d'une espace.

```
1  public class SimpleVariable {
2
3      public static void main(String[] args) {
4          // température à la surface du soleil en kelvin
5          int temperature = 5778;
6          System.out.println("Température du soleil (K)");
7          System.out.println(temperature); //
8          System.out.println("Température du soleil (degree C)");
9          // conversion : degree C = K - 273.15
10         System.out.println(temperature - 273.15); //
11     }
12
13 }
```

Listing 2 – Exemple de programme utilisant des variables.

Les variables sont caractérisées par leur type qui définit la nature des valeurs représentées ainsi que les opérations qu'il est possible de réaliser avec celle-ci. Par exemple, en java, une variable de type entier (`int`) représente un nombre entier compris entre -2^{31} et $2^{31} - 1$; il est possible d'additionner deux entiers entre eux à l'aide de l'instruction :

```java
int a = 2;
int b = 4;
int c = a + b;
```

La variable c représentera la valeur 6. Il est également possible de réaliser la division entière entre deux entiers :

```java
System.out.println(a / b);
```

Cette instruction affichera 0 (et non 0, 5 comme on pourrait s'y attendre), l'opérateur de division `/` étant défini comme la division entière lorsqu'il est appliqué à des entiers. De la même manière, il est possible d'« additionner » deux chaines de caractères entre elles ou une chaine de caractères et un entier (dans ces deux cas, l'addition correspond à une concaténation), comme dans l'exemple du listing 3, mais il n'est possible de diviser deux chaines de caractères ou d'additionner un entier et une chaine de caractères : ces deux opérations ne sont pas définies pour les types concernés. Lors de la compilation, les opérations entre types non autorisées sont détectées et entraine une erreur de compilation. Par exemple, la compilation du programme du listing 3 génère l'erreur suivante :

```
> javac StringOperation.java
StringOperation.java:10: error: bad operand types for binary operator '/'
        String d = a / b;
                   ^
  first type:  String
  second type: String
1 error
```

```java
public class StringOperation {

    public static void main(String[] args) {
        String a = "Bonjour";
        String b = "les amis";
        int c = 1;
        System.out.println(a + " " + b);
        System.out.println(a + c);

        // Opérations interdites pour le type String
        String d = a / b;
        System.out.println(1 + c);
    }

}
```

Listing 3 – Programme réalisant une opération non définie pour le type `String`. La compilation de ce programme donne une erreur.

On distingue en java deux sortes de types :

les types primitifs qui correspondent à des types pouvant être manipulés directement par un microprocesseur. Les types primitifs sont stockées « tel que » en mémoire sans aucune information supplémentaire. La table 1.1 donne la liste des principaux types primitifs existant en java.[4]

4: Seuls les types primitifs que l'on emploi couramment sont présentés. Le langage java définit d'autres types offrant un contrôle fin sur l'occupation mémoire et la précision (l'ensemble des valeurs qui peut être représenté). Les caractéristiques et l'utilisation de ces types dépassent le cadre de ce document.

type	valeurs représentées	commentaire
`int`	$[\![-2^{31}, 2^{31}-1]\!]$	nombres entiers
`boolean`	{`true`, `false`}	booléen
`double`	sous-ensemble de $\mathbb{R}$	représentation approchée des nombres réels

Table 1.1 – Liste des principaux types primitifs utilisés en java.

les types complexes qui correspondent à des *objets* pouvant être définis soit dans la bibliothèque standard java, soit directement par le programmeur. Ces types complexes peuvent représenter n'importe quelle entité. L'utilisation des types complexes sera détaillée à la section 1.5.

En java les variables sont typés de manière explicite et statique : il est nécessaire de déclarer, à la création d'une variable, le type de celle-ci et ce type ne pourra plus changer. Ainsi, la définition d'une variable de type primitif aura toujours la forme suivante :

```
int a = 0 ;
```
type — nom de la variable — valeur initiale

Les variables de type complexe devront être définies de la manière suivante :

```
File a = new File("mon_fichier.txt") ;
```
type — nom de la variable — appel au constructeur

Dans cet exemple, le type complexe `File`, défini dans la bibliothèque standard java, permet de représenter et de manipuler un fichier ou un répertoire (p. ex. en vérifiant si le fichier existe, en créant des répertoires, ...). La création d'un type complexe se fait au moyen d'un *constructeur* : c'est une fonction particulière dont le nom est absolument identique au nom du type (casse compris) et qui peut prendre en paramètre d'éventuels paramètres nécessaires à l'initialisation de l'objet. Dans l'exemple précédent, les paramètres du constructeur permettent de spécifier le chemin du fichier. L'utilisation des types complexes sera détaillée à la section **??**.

Il existe un type particulier, le type `String` qui permet de représenter des chaines de caractères : bien que ce soit un type complexe, il est possible d'initialiser une chaine de caractères en utilisant la syntaxe des types primitifs :

```
1 String s = "Bonjour les amis";
```

Cette initialisation est, en première approximation, équivalente à :

```
1 String s = new String("Bonjour les amis");
```

1.4 Instruction de contrôle de flots

Lors de l'exécution d'un programme java, les instructions d'un programme java sont exécutées les unes à la suite des autres en commençant par la première ligne de la fonction principale. Il existe deux instructions permettant de contrôler la manière dont les lignes sont exécutées :

— une instruction de *branchement conditionnel* qui permet de n'exécuter une ligne ou un bloc de lignes que si une condition est respectée ;
— une instruction de *répétition* qui permet de répéter une ligne ou un bloc de lignes.

Branchement conditionnel Comme dans la plupart des langages, le branchement conditionnel s'exprime en java à l'aide du mot clé **`if`**. Sa syntaxe générale est :

```
1  if (condition1) {
2    // bloc d'instructions exécutés si la condition1
3    // est vraie
4  } else if (condition2) {
5    // bloc d'instructions exécutés si la condition2
6    // est vraie
7  } else {
8    // bloc d'instructions exécutés si toutes les
9    // conditions précédentes sont fausses
10 }
```

Une condition doit nécessairement être entre parenthèses. La présentation du bloc suit les conventions suivantes : l'accolade ouvrante est sur la ligne contenant la condition (précédé d'un espace) et l'accolade fermante soit seule sur une ligne avec l'indentation au même niveau que le **`if`** soit sur la même ligne que le **`else`** (ou **`else if`** en cas de conditions multiples).

La condition peut être n'importe quelle expression renvoyant un booléen. C'est en général soit une fonction renvoyant un booléen, soit le résultat d'une comparaison : `==` pour le test d'égalité des types primitifs (nous verrons à la section 1.5 comment tester l'égalité de types complexes), `<`, `>`, `<=` ou `>=` pour les tests d'ordre.

Instructions de répétition Il existe deux manières de répéter un bloc. La première consiste à utiliser un compteur et à définir explicitement le nombre de fois où l'on veut répéter le bloc. Par exemple, le code suivant :

```
1 for (int i = 0; i < 10; i += 1) {
2   if (i == 0) {
3     System.out.println((i + 1) + "ère ligne")
4   } else {
5     System.out.println((i + 1) + "ème ligne")
6   }
7 }
```

permet de « compter » les lignes (en distinguant la première). La sortie de ce programme sera :

```
1ère ligne
2ème ligne
3ème ligne
4ème ligne
5ème ligne
6ème ligne
7ème ligne
8ème ligne
9ème ligne
10ème ligne
```

La syntaxe générale de la boucle **`for`** est la suivante :

```
for (initialisation; test; itération) {
  bloc
}
```

Les instructions du bloc seront répétées tant que la condition exprimée dans le test sera vraie ; à chaque itération (répétition), le code contenu dans `itération` sera exécuté pour mettre à jour le compteur. Ce dernier peut-être initialisé à l'aide de l'instruction `initialisation` qui est exécutée avant la première exécution du bloc. La figure 1.4 schématise le fonctionnement de la boucle **`for`**. Ce schéma montre, notamment, que le bloc d'instructions peut ne pas être exécuté si la condition est fausse après l'exécution des instructions `initialisation`.

Figure 1.4

La seconde manière de répéter l'exécution d'une séquence d'instructions consiste à parcourir tous les éléments d'une collection, c'est-à-dire un ensemble d'éléments de même types (la notion de collection sera détaillée à la section 1.6). La syntaxe générale de ce type de boucle est :

```
for (Type el : collection) {
  // bloc d'instructions à répéter
}
```

Ce boucle correspond aux boucles de type *foreach* présente dans de nombreux langage : la variable `el`, de type `Type`, va successivement prendre la valeur des différents éléments de la collection. Par exemple, le code du listing 4 permet de déterminer la somme des entiers contenus sur une ligne, ces entiers étant séparés par une virgule : la boucle va permettre de parcourir les différents éléments (obtenus en appliquant un *split* sur la chaine de caractères initiales), les convertir en entiers et additionner les entiers ainsi obtenus.

```java
public class SumInt {

    public static void main(String[] args) {
        String input = "12,45,3,9";

        int sum = 0;
        for (String el : input.spit(",")) {
            sum += Integer.parseInt(el);
        }
        System.out.println("la somme est " + sum);
    }
}
```

Listing 4 – Programme réalisant la somme des entiers contenus dans une chaine de caractères.

Exercice 1.4.1

Pour illustrer le fonctionnement de ces deux instructions nous nous intéressons au problème suivant : étant donné un nombre de lignes spécifié par une variable `n`, dessiner une pyramide telle celle représentée à la figure 1.5. Cette pyramide est composée de n lignes ; la première comporte 2 symboles, la dernière $2 \cdot n$ symboles ; les lignes sont centrées. Les lignes paires sont composées de `+`, les lignes impaires de $*$.

(correction page 25)

```
**
++++
******
++++++++
**********
++++++++++++
```

Figure 1.5 – Exemple de pyramide à réaliser dans l'exercice 1.4.1

Fonctions Le listing 5 montre comment il est possible de définir et d'appeler une fonction en java. En java, les fonctions sont nécessairement définies à l'intérieur d'une classe (comme la fonction `main` que nous avons utilisé jusqu'à présent). Une fonction est constituée :

— d'un entête définissant le nom de la fonction, son type de retour le nombre et le type de ses paramètres ;

```
public static   int    max   (   int a, int b   )
      ↑          ↑      ↖          ↑
  définition  type de retour  nom  paramètres
```

— d'un corps comportant des instructions arbitraires (définition de variables, boucles, conditions, appel à d'autres fonctions, ...).

Il n'existe techniquement pas de fonction en java, mais uniquement des méthodes statiques[5] comme le suggère l'utilisation du mot clé **`static`**. En pratique, si ce mot-clé n'est pas présent, l'appel à la fonction générera lors de la compilation l'erreur suivante :

5: La notion de méthode statique sera définie plus précisément au chapitre 3.

```
MaxFunction.java:12: error: non-static method max(int,int)
cannot be referenced from a static context
        System.out.println("max(5, 3) = " + max(5, 3));
                                             ^
1 error
```

Une fois définie une fonction peut être appelée, comme à la ligne 12 du listing 5 en précisant son nom et, entre parenthèses la valeur des éventuels paramètres. Si la fonction est définie dans une autre classe que

```java
public class MaxFunction {

    public static int max(int a, int b) {
        if (a <= b) {
            return b;
        } else {
            return a;
        }
    }

    public static void main(String[] s) {
        System.out.println("max(5, 3) = " + max(5, 3)); //
    }

}
```

Listing 5 – Définition et appel d'une fonction en java.

la classe courante, le nom de la fonction devra être précédé du nom de la classe. Par exemple, pour appeler la fonction `min` définie dans la classe `Math`[6] , la syntaxe est :

```java
System.out.println(Math.min(12.123, 12.456));
```

6: La classe `Math` définie la plupart des fonctions mathématique courante : `log`, `sin`, `sqrt`, ...

Comme schématisé à la figure 1.6, les fonctions permettent de suspendre le flot d'instructions. Lorsqu'une ligne contient un appel à une fonction, le déroulement du programme est interrompu et l'exécution continue au début de la fonction. Lors de l'appel, les valeurs passées à la fonction sont affectées aux paramètres de celle-ci. Ceux-ci peuvent ensuite être considérés comme des variables *locales* qui seront détruites à la fin de la fonction (elles ne sont donc accessibles que dans le corps de la fonction). Les lignes composant la fonction sont ensuite exécutées les unes à la suite des autres jusqu'à une ligne comportant un **`return`**. Lors de l'exécution d'une ligne comportant ce mot-clé, l'exécution reprend au point d'appel : la ligne contenant l'appel est exécutée en faisant comme si la fonction était « remplacée » par la valeur retournée.

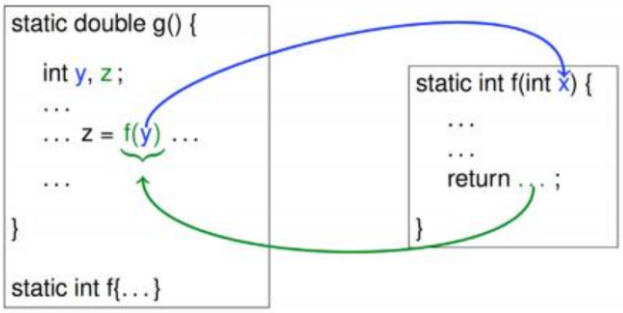

Figure 1.6 – Déroulement du flots d'instructions lors d'un appel à une fonction.

Définir des fonctions présente plusieurs intérêts. C'est un moyen de *factoriser* le code : plutôt que de répéter le même bloc d'instructions à différents endroits, une même séquence d'instructions n'est jamais dupliquée et n'a besoin d'être écrite qu'une seule fois. Ainsi, grâce à la factorisation, il n'y a plus qu'un endroit à modifier pour faire évoluer le code ou corriger une éventuelle erreur, ce qui rend le code plus robuste (le risque d'oublier de modifier certaines instances du bloc du code est limité).

La définition de fonction est également un moyen de *structurer* le code : plutôt que d'avoir une longue suite d'instructions résolvant un problème, la résolution de celui-ci est diviser une succession d'étapes pouvant être clairement identifiées, ce qui améliore la lisibilité du code : il est possible de comprendre les « grandes étapes » de la résolution du problème sans nécessairement savoir comment ces étapes sont réalisées. Un des grands intérêt des fonctions est de réaliser une *abstraction* du code : il est possible de comprendre ce que fait un bloc de code, sans savoir comment il le fait et donc d'utiliser une fonction comme une « boite noire » sans savoir comment celle-ci fonctionne.

Pour se convaincre de l'utilité de cette abstraction, il suffit de chercher à comprendre ce que fait la boucle suivante :

```
  r = n / 2;
  while (abs( r - (n / r) ) > t) {
    r = 0.5 * (r + (n / r));
  }
  System.out.println("r = " + r);
```

et de le comparer à :

```
public static double squareRootApproximation(double n) {
  r = n / 2;
  while (abs( r - (n / r) ) > t) {
    r = 0.5 * (r + (n / r));
  }
  return r;
}
```

Le simple fait d'encapsuler le code dans une fonction et de nommer celle-ci rend l'interprétation du code triviale : il n'y a même pas besoin d'ajouter des commentaires !

De manière plus générale, lorsque l'on écrit `double res = 3 * 3 * 3;` ou même `double res = x * x * x;`, le programme est capable de calculer des cubes, mais notre *langage* n'a pas accès au *concept* de « élever un nombre à la puissance 3 ». La définition d'une fonction permet d'enrichir le langage en lui ajoutant, d'une certaines manières, de nouvelles instructions.

Figure 1.8 – La notion d'abstraction en informatique. source : https://xkcd.com/676/

1.5 Utilisation des objets

En plus des types primitifs, java permet également de manipuler des types complexes. Les variables dont le type est complexe sont généralement appelées des *objets*. Les types complexes ou *classes* sont directement définis par un programmeur pour étendre le langage de base en lui ajoutant de nouvelles fonctionnalités. D'une certaine manière leur rôle est similaire à celui d'une fonction qui permet au programmeur d'enrichir un langage en lui ajoutant de nouvelles instructions. Nous verrons plus en détail au chapitre **??** les motivations de la programmation orientée objet.

Il existe trois types de classes différentes :

— les classes de la bibliothèque standard ;
— les classes fournies dans une bibliothèque tierce, comme par exemple `coreNLP` ∗ qui offre différents fonctions pour faire du TAL en java (analyseur morpho-syntaxique, analyseur syntaxique, reconnaissance d'entités nommées ou de coréférences). Celle dernière doit être installée (en téléchargeant le code correspondant et en indiquant à la machine virtuelle qu'elle doit utiliser celui-ci).
— les classes composant le projet.

Il est illusoire de vouloir maitriser l'ensemble des classes existantes ; mais il est important de savoir comment identifier les classes pertinentes pour résoudre un problème et découvrir les fonctionnalités que celles-ci offrent.

La manipulation d'un type complexe comporte toujours deux étapes :

— la création de l'objet à l'aide du mot clé **`new`** qui permet d'obtenir une *référence* sur l'objet (cf. 1.3) ;
— l'utilisation de celui-ci à l'aide de *méthodes* qui lui sont propres.

Utiliser (à l'aide d'une méthode) un objet sans que celui-ci n'ait été initialisé crée une erreur lors de l'exécution du programme. Contrairement aux erreurs de syntaxes qui sont détectées lors de la compilation, ce type d'erreur ne peut pas toujours être détectée avant que le code ne soit exécuté et résulte en une *exception* de type `NullPointerException`.

L'utilisation d'une méthode se fait au moyen de la syntaxe suivante :

```
1  laReference.laMethode(arguments);
```

Contrairement aux fonctions « habituelles » (cf. **??**), une méthode ne peut être appelée que pour modifier ou questionner un objet. Elle est d'une certaine manière « attachée » à une référence : il est impossible d'« appeler » une méthode sans spécifier la référence à laquelle celle-ci s'applique.

Par exemple, le code du listing 6 utilise les méthodes de la classe `String` suivante :

— `toLowerCase()` : qui permet de créer une nouvelle chaine de caractère correspondant à la version en minuscule de chaine sur laquelle la méthode est appliquée. Appeler cette méthode permet de ne pas tenir compte de la casse lors de la comparaison †
— `split` : qui permet d'identifier les sous-parties de la chaine de caractères (ici, les associations entre un nom et un entier, puis, à l'intérieur de chaque association, entre un nom et un entier) ;
— `equals` : qui permet de tester l'égalité de deux chaines de caractères (cf. XXX).

Une description plus précise de ces méthodes est faite dans la javadoc de la classe `String`.

Pour connaitre les objets existants et les méthodes afférentes, il suffit en général d'entrer le nom de la classe dans un moteur de recherche pour tomber sur la *javadoc* de la classe. Celle-ci offre une documentation standardisée d'une classe. Elle comporte trois parties :

— une description générale des fonctionnalités de la classe ;

∗. https://stanfordnlp.github.io/CoreNLP/
†. On aurait également pu utiliser la méthode `equalsIgnoreCase`.

```java
public class SumString {

    public static void main(String[] args) {
        String input = "ab=2,bC=1,Ab=3,cd=2";

        input = input.toLowerCase();

        int total = 0;
        for (String part : input.split(",")) {
            String name = part.split("=")[0];
            if (name.equals("ab")) {
                total += Integer.parseInt(part.split("=")[1]);
            }
        }
        System.out.println("total = " + total);
    }
}
```

Listing 6 – Exemple d'utilisation d'un objet de type `String`. Le programme permet d'identifier tous les entiers associés à la chaine `"ab"` (sans tenir compte de la casse) et de calculer la somme de ceux-ci.

— la liste des méthodes avec une description succincte (en générale une phrase) de celle-ci ;
— une description détaillée des différentes méthodes.

La plupart des EDI permettent également d'accéder directement à la javadoc.

1.6 Structures de données en java

Java pour la manipulation de texte

2

Ce chapitre a pour objectif de donner un aperçu rapide des principales classes de la bibliothèque standard Java permettant de manipuler des données textuels. Nous verrons successivement :

— comment lire des données à partir d'un fichier ;
— les principales structures de données ;
— la manipulation des textes unicode.

La plupart des classes que nous verrons dans ce chapitre ne sont pas directement connues ni du compilateur ni de la machine virtuelle et doivent être *importées* avant de pouvoir être utilisées. Les directives d'importation doivent être situées au début du fichier, avant la déclaration de la classe (le `public class`), comme dans l'exemple suivant :

```
1 import java.util.HashMap;
2
3 public class MaClasse {
4   // ..
5 }
```

2.1 Manipulation de fichier

Deux classes de la bibliothèque standard Java[1] permettent d'accéder et de manipuler un fichier :

— la classe `Path` qui, comme son nom l'indique, représente un *chemin* c'est-à-dire, un objet qui permet de localiser un fichier dans le système de fichiers ;
— la classe `Files` qui contient plusieurs méthodes statiques permettant d'accéder (c.-à-d. de lire) et de manipuler (copier, supprimer, ...) ceux-ci.

1: L'accès aux fichiers à été grandement simplifié à partir de la version 7 de Java. Cette section décrit des classes et des méthodes qui n'existent pas dans les versions de Java antérieures à cette version.

Il faut ajouter à cela la classe `Paths`[2] qui permet, exclusivement, de créer des instances de la seconde à partir de la désignation d'un fichier sous forme d'une chaine de caractères.

2: Attention au `s` !

Lecture La lecture d'un fichier se fait en deux étapes : il faut commencer par créer une instance de `Path` désignant le fichier ; il est ensuite possible, à l'aide des fonctions de la classe `File` d'accéder au contenu de celui-ci. Dans l'exemple du listing 7, la première étape est réalisé à l'aide de la méthode statique `Paths.get` ; la lecture du contenu du fichier se fait à l'aide de la méthode `Files.readAllBytes`. Le contenu du fichier est converti en chaine de caractères en appelant un constructeur de la classe `String`[3] .

3: La différence entre `byte` et `String` sera expliquée à la section **??**

Lors de leur exécution, toutes les méthodes manipulant des fichiers peuvent échouer par exemple lorsqu'elles tentent d'accéder à fichier qui n'existe pas ou pour lequel l'utilisateur exécutant le programme n'a pas les droits. Les erreurs causées par un problème d'accès à un

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class LoadFile {

    public static void main(String[] a) throws IOException {
        String content = new String(
               Files.readAllBytes(Paths.get("LoadFile.java"))
           );

        int nLines = 0;
        for (String line : content.split("\n")) {
            if (!line.isEmpty()) {
                nLines += 1;
            }
        }

        System.out.println("Il y a " + nLines + " lignes non vide");
    }

}
```

Listing 7 – Exemple de lecture du contenu d'un fichier en Java : le programme compte le nombre de lignes non vides contenues dans un fichier .

fichier sont signalées par des exceptions de type `IOException`. Il est obligatoire de signaler au compilateur ces erreurs éventuelles en ajoutant à la signature des fonctions utilisant des méthodes de la classe `File` la directive **`throws`** `IOException`. Cette déclaration s'étend à toutes les méthodes appelant une méthode pouvant lever une telle exception[4] .

4: Ces méthodes sont identifiables par la présence d'une directive **`throws`** dans leur signature et dans leur *javadoc*. Nous reviendrons sur la gestion des exceptions et la directive **`throws`** au chapitre **??**

Le code du listing 7 copie la totalité d'un fichier dans une variable. Il est ensuite possible d'extraire de cette variable les informations dont on a besoin (dans l'exemple, les lignes composant le fichier). Cette approche n'est cependant pas possible lorsque la taille du fichier est trop grande et que son chargement risque de saturer la mémoire. Il est, dans ce cas, préférable de traiter les informations *à la volée*, comme dans le listing 8. Contrairement au listing **??**, le fichier est alors parcouru ligne à ligne et seule une ligne à la fois est stockée en mémoire : la mémoire est libérée dès que la fin de l'itération.

Le listing 8 repose sur l'utilisation de *flux* (représenté par des instances de la classe `Stream`) qui seront présentés au chapitre **??**. La construction, il est vrai particulièrement alambiquée, permet de respecter les contraintes imposées par l'utilisation des flux : en particulier, il est nécessaire de *protéger* à l'aide du mot clé **`try`** le bloc accédant au fichier afin de garantir que celui-ci soit correctement fermé et éviter une « fuite de ressource » *(resource leak)*.

Écriture La classe `Files` fournit un moyen simple d'écrire une chaine de caractères dans un fichier :

```java
Path path = Paths.get("exemple.txt");
String content = "bonjours les amis !\ncomment allez-vous ?";
Files.write(path, content.getBytes("UTF-8"));
```

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class StreamingCountLines {

    public static void main(String[] a) throws IOException {

        int nLines = 0;
        try (Stream<String> lines = Files.lines(Paths.get("exemple.txt"))) {
            for (String line : (Iterable<String>) lines::iterator) {
                if (!line.isEmpty()) {
                    nLines += 1;
                }
            }
        }

        System.out.println("Il y a " + nLines + " lignes");

    }

}
```

Listing 8 – Compte les lignes non vides d'un fichier sans stocker la totalité du fichier en mémoire.

Comme nous le verrons à la section **??**, il est nécessaire de spécifier un *encodage* qui sera toujours (ou presque) l'UTF-8.

2.2 Structures de données

La bibliothèque standard de java contient plusieurs classes pour représenter des *collections* d'objets en Java. Les collections permettent de représenter et de stocker plusieurs éléments de même types : un texte peut, par exemple, être considéré comme une collection de mots, chaque mot étant représenté par une variable de type `String`. Il existe trois principaux types de collections :

— les listes ;
— les ensembles ;
— les dictionnaires.

Ces types de collections se distinguent aussi bien par les opérations qu'ils permettent que par les performances de ces opérations : comme nous le verrons, il est possible, pour les trois types de collections de tester si un élément appartient à la collection ou non mais cette opération sera beaucoup plus efficace pour les ensembles ou les dictionnaires que pour les listes.

En java, toutes les collections sont dynamiques : il est possible d'ajouter et de supprimer des éléments d'une collection sans aucun problème. Elles sont également homogènes : elles ne peuvent contenir que des objets de même type et il n'est pas possible de stocker, par exemple, des chaines de caractères et des entiers dans une même collection.

Les listes

Les listes sont des collections dynamiques et ordonnées d'objets homogènes. La classe `ArrayList` de la bibliothèque standard permet de représenter des listes. Comme toutes les classes représentant des collections, il s'agit d'une *classe paramétrée* : le type des objets qui y seront stockés doit être spécifié au moment de la déclaration d'une instance de cette classe. Pour cela, il faut l'ajouter entre chevron au nom de la classe : par exemple une `ArrayList` contenant des `String` sera de type `ArrayList<String>`; une `ArrayList` contenant des étudiants (représentés par des instances d'une classe `Etudiant`) sera de type `ArrayList<Etudiant>`.

Les collections java ne permettent pas de stocker des éléments de type primitif : le type doit nécessairement être spécifié par un nom de classe. Nous verrons à la section 9 comment définir des collections de **`double`** ou de **`int`**.

Le code suivant permet de créer une `ArrayList` contant des `String` et d'insérer trois éléments dans celle-ci :

```
1 ArrayList<String> phrase = new ArrayList<>();
2 phrase.add("Bonjour");
3 phrase.add("les");
4 phrase.add("amis");
```

La principale caractéristique des listes est d'être ordonné : chaque élément inséré dans une liste est associé à un *indice* indiquant l'ordre dans lequel il a été inséré et un élément peut être récupéré directement à partir de son indice. Il est possible de considérer une liste comme l'association entre une série d'indices (des entiers consécutifs et les objets correspondant, comme dans la figure 2.1. Une liste peut contenir plusieurs éléments identiques s'ils ont été insérés à des positions différentes.

Comme souvent en informatique, le premier élément est inséré à la position 0. Les indices d'une liste contenant n éléments seront donc compris entre 0 et $n - 1$.

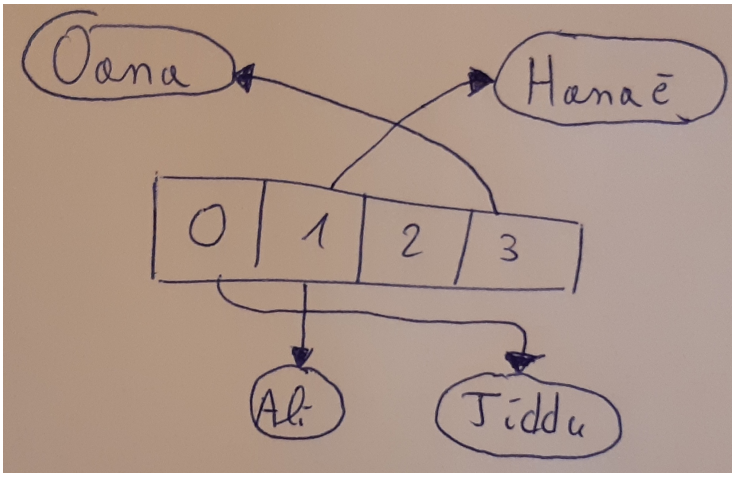

Figure 2.1 – Représentation schématique d'une liste : les 4 éléments sont associés à un indice (compris entre 0 et 3) et il est possible, connaissant un indice d'accéder directement à l'élément correspondant.

L'interface de la classe `ArrayList` fournit de nombreuses méthodes. Les opérations les plus souvent utilisées sont :

— l'ajout d'un élément à la fin de la liste avec la méthode `add`;
— le parcours de tous les éléments par ordre d'insertion à l'aide d'une boucle *foreach* :

```
1 int i = 1;
2 for (String mot : phrase) {
3   System.out.println("le " + i + "ème mot est : "
4         + mot);
5   i += 1;
6 }
```

— l'accès à un élément par son index avec la méthode `get` ;
— le test d'appartenance avec la méthode `contains` qui renvoie **`true`** si un objet est présent dans la collection.

Les deux méthodes d'accès aux éléments d'une `ArrayList` sont « efficaces » : leur durée d'exécution est, en général, indépendante du nombre d'éléments contenus dans la collection. Au contraire, la durée d'exécution de la méthode `contains` est proportionnel au nombre d'éléments stockés dans la collection[5] . En pratique, l'usage de la méthode `contains` est déconseillé.

Le listing 9 donne un exemple d'utilisation des `ArrayList` en montrant comment déterminer les mots communs à deux phrases. Pour cela, le programme commence par construire deux `ArrayList` contenant les mots de la première phrase[6] . Puis il parcourt les mots de la seconde phrase en regardant pour chaque élément si :

— celui-ci est présent dans la première phrase (pour détecter les mots en commun). Comme ceux-ci sont stockés dans une `ArrayList`, ce test peut se faire directement en utilisant les méthodes de la classe.
— celui-ci n'a pas encore été ajouté à l'`ArrayList` contenant le résultat (pour éviter qu'un mot ne soit compter deux fois).

Si ces deux conditions sont vraies, le mot « courant » est ajouté à une deuxième `ArrayList` contenant tous les mots communs. Il suffit, une fois toute la phrase parcourue, de déterminer la taille de cette dernière `ArrayList`.

5: De manière plus précise, la complexité de la méthode `get` est en $\mathcal{O}(1)$, la complexité *amortie* de la méthode `add` en $\mathcal{O}(1)$ et celle de la méthode `contains` en $\mathcal{O}(n)$ où n est le nombre d'éléments stockés dans l'`ArrayList`.

6: La construction d'une `ArrayList` contenant le résultat d'un `split` peut se faire en une seule ligne : `new ArrayList<>(Arrays.asList(s1.split(" ")))`.

```java
import java.io.IOException;
import java.util.ArrayList;

public class CountCommon {

        public static void main(String[] a) throws IOException {

                String sent1 = "le chat et le chien dorment bien .";
                String sent2 = "le chat et la chatte jouent bien .";

                ArrayList<String> words1 = new ArrayList<>();
                for (String word : sent1.split(" ")) {
                        words1.add(word);
                }

                ArrayList<String> common = new ArrayList<>();
                for (String word : sent2.split(" ")) {
                        if (sent1.contains(word) && !common.contains(word)) {
                                common.add(word);
                        }
                }

                System.out.println("Il y a " + common.size() +        \
                                                                " mots communs

        }

}
```

Listing 9 – Exemple d'utilisation des ArrayList : le programme détermine le nombre de mots communs entre deux phrases.

Exercice 2.2.1

En utilisant une `ArrayList` déterminer le nombre de types (mots uniques) apparaissant dans un fichier. On appelle mot une suite de caractères *en minuscule* séparée par des espaces ou des signes de ponctuations (« `chat,` » et « `Chat` » sont donc deux mots indiques).

Correction : page 28.

Cet exercice a uniquement un but pédagogique : l'utilisation d'un ensemble (cf. §**??**) apporte une solution plus jolie et surtout plus efficace.

Exercice 2.2.2

Écrire une fonction qui prend en entrée une chaine de caractères `c` et qui renvoie une `ArrayList` contenant les mots de cette chaine apparaissant exactement une fois dans `c` dans l'ordre dans lequel ils apparaissent dans `c`.

Correction : page 28.

Les ensembles

Les ensembles sont des collections dynamiques non ordonnées d'éléments uniques. Ils permettent de représenter des ensembles au sens mathématique du terme. Contrairement aux listes, l'ordre d'insertion n'est pas conservé (il n'y a donc pas d'indices) et la notion d'indice n'est pas définie[7] .

7: L'opération « accéder au i^e élément » n'a donc pas de sens.

La classe `HashSet` de la bibliothèque standard java offre une implémentation d'un ensemble. Les principales méthodes de cette classe sont :

— `add` qui ajoute un élément à la collection ;
— `contains` qui teste si un élément appartient à l'ensemble ou non.

Le principal intérêt des ensembles est de pouvoir tester de manière très efficace si un élément appartient à la collection ou non : le temps d'exécution de ce test est indépendant du nombre d'éléments stockés dans la collection[8] alors que, comme nous l'avons vu dans la sous-section précédente, cette opération a un temps d'exécution proportionnel au nombre d'éléments stockés pour les listes[9] .

8: De manière plus précise, la complexité de l'opération est en $\mathcal{O}(1)$.

9: De manière plus précise, l'opération a une complexité en $\mathcal{O}(n)$ où n est le nombre d'éléments stockés.

Le listing 10 montre comment le programme du listing 9 peut être amélioré en utilisant des `HashSet` : en plus d'être plus compact (il n'y a pas besoin de tester si le mot est déjà présent dans la collection `common`), le code s'exécutera également nettement plus rapidement pour les « grandes » phrases.

En utilisant les méthodes de la classe `HashSet` et de la classe `Arrays`, il est possible, comme le montre le listing 11 d'apporter une solution encore plus compacte au problème. Cet exemple montre à quel point il est important de toujours vérifier les méthodes fournies par les classes avant de commencer à coder.

Exercice 2.2.3

Écrire une méthode qui teste si une instance de `ArrayList<String>` contient des éléments répétés ou non.

```java
import java.io.IOException;
import java.util.HashSet;

public class CountCommonHashSet {

  public static void main(String[] a) throws IOException {

    String sent1 = "le chat et le chien dorment bien .";
    String sent2 = "le chat et la chatte jouent bien .";

    HashSet<String> words1 = new HashSet<>();
    for (String word : sentence1.split(" ")) {
      words1.add(word);
    }

    HashSet<String> common = new HashSet<>();
    for (String word : sentence2.split(" ")) {
      if (words1.contains(word)) {
        common.add(word);
      }
    }

    System.out.println("Il y a " + common.size() +
                       " mots communs");
  }

}
```

Listing 10 – Programme déterminant le nombre de types communs entre deux phrases à l'aide d'un `HashSet`.

Exercice 2.2.4

Étant donné une `ArrayList lst` contenant des chaines de caractères, comptez le nombre de paires `(lst[i], lst[j])` avec $i < j$ distinctes. Aisni, si la liste est `[a, a, b]`, il y a deux pairs distinctes respectant la condition : `(a, a)` et `(a, b)`.

Conseil : la classe `Pair` du package `javafx.util` permet de représenter une paire.

Les dictionnaires

Un dictionnaire représente une collection d'associations entre une paire d'objets : une clé et la valeur qui lui est associée. Un des principaux intérêts des dictionnaires est qu'il est possible, connaissant une clé, de retrouver la valeur qui lui est associée. Un dictionnaire peut être vu comme un annuaire téléphonique : celui-ci permet de stocker des associations entre des noms de personnes et des numéros de téléphone et il est possible, connaissant un nom, de retrouver le numéro de téléphone qui lui est associé, le contraire (retrouver un nom connaissant un numéro de téléphone étant beaucoup plus compliqué). Mais, contrairement à un annuaire, les clés doivent cependant être uniques et celles-ci ne sont pas ordonnées : il n'est possible d'associer qu'une seule valeur à une clé donnée ; mais plusieurs clés peuvent être associées à des valeurs identiques. En pratique, il est possible de considérer les `HashMap` comme

Selon les langages, les dictionnaires peuvent également être appelés « tableaux associatifs », « table de hachage » ou « hash map ».

```java
public static void main(String[] a) throws IOException {

  String sent1 = "le chat et le chien dorment bien .";
  String sent2 = "le chat et la chattent jouent bien .";

  HashSet<String> words1 =
         new HashSet<>(Arrays.asList(sent1.split(" ")));
  HashSet<String> words2 =
         new HashSet<>(Arrays.asList(sent2.split(" ")));

  words1.retainAll(words2);

  System.out.println("il y a " + words1.size() +
                     " mots en commun");

}
```

Listing 11 – Refactoring du code du listing 10 utilisant les méthodes de la classe `HashSet`.

des généralisations de listes dans lesquels les indices peuvent être des objets arbitraires.

La classe `HashMap` permet de représenter des dictionnaires en java. Comme toutes les collections, il faut spécifier lors de la déclaration le type des éléments qui y seront stockés. Il y a, pour les dictionnaires, deux types à spécifier : le type des clés et le type des valeurs qui devront être spécifiés, séparé par une virgule, dans cet ordre. Par exemple, la déclaration d'un dictionnaire associant une chaine de caractère à une autre sera :

```java
HashMap<String, String> count = new HashMap<>();
```

un tel dictionnaire permet, par exemple, de représenter un lexique associant à un mot anglais sa traduction en français. Cette représentation impose toutefois une contrainte forte : un mot anglais ne peut avoir qu'une seule traduction en français. Pour associer un mot anglais à une liste de traductions possibles en français, il faut déclarer une variable de type `HashMap<String, ArrayList<String>>`.

L'interface de la classe `HashMap` définie quatre méthodes principales :

- la méthode `put(key, value)` qui créer une association entre une clé et une valeur;
- la méthode `get(key)` qui retourne la valeur associée à une clé. Cette méthode renvoie `null` lorsque la clé n'est pas présente pas dans le dictionnaire. Il est donc nécessaire de vérifier explicitement si la clé est contenue dans le dictionnaire avant d'utiliser la valeur qui lui est associée.
- la méthode `getOrDefault(key, defaultValue)` qui renvoie la valeur associée à la clé `key` si celle-ci est présente dans le dictionnaire et `defaultValue` dans le cas contraire.
- la méthode `containsKey(key)` qui teste si une clé est présente dans le dictionnaire ou non.

Le code du listing 12 met en œuvre ces méthodes. Il permet de « traduire » mot-à-mot une phrase française en anglais : la phrase anglaise est générée en considérant successivement les mots de la phrase française et en insérant soit la traduction de celui-ci si celle-ci est connue soit le mot

directement entre astérisques. Le code tire avantage de la méthode `getOrDefault` qui délègue le test à l'implémentation de la classe `HashMap`, plutôt que de réaliser celui-ci explicitement, par exemple, de la manière suivante ;

```
  if (dico.containsKey(word)) {
    translatedSentence += " " + dico.get(word);
  } else {
    translatedSentence += " *" + word + "*";
  }
```

```
import java.util.HashMap;

public class WordTranslator {

    public static void main(String[] s) {
        String sentence = "the cat sleeps on the mat";

        HashMap<String, String> dict = new HashMap<>();
        String content = "the:le;cat:chat;sleeps:dort";

        for (String pair : content.split(";")) {
            dict.put(pair.split(":")[0], pair.split(":")[1]);
        }

        String translatedSent = "";
        for (String word : sentence.split(" ")) {
            translatedSent += " " + \
                dict.getOrDefault(word, "*" + word + "*");
        }
        translatedSent = translatedSent.substring(1);

        System.out.println(translatedSent);
    }

}
```

Listing 12 – Exemple d'utilisation d'un dictionnaire : traduction d'une phrase mot-à-mot avec gestion des mots inconnus.

Exercice 2.2.5

Écrire une fonction qui prend en paramètre un nom de fichiers et renvoie un dictionnaire associant chaque type contenu dans le fichier au nombre d'occurrence de celui-ci : si le contenu du fichier est `"a b a\na b"`, le dictionnaire renvoyé sera : `{'a': 3, 'b': 1'}`

Opérations sur les collections

2.3 Java & Unicode

Programmation orientée objet | 3

Appendix

Correction des exercices A

A.1 Correction de l'exercice 1.4.1

Comme tous les problèmes d'informatique, il est plus simple de commencer par résoudre une version simplifiée du problème et de modifier ensuite le code de proche en proche pour traiter des cas de plus en plus complexes.

Nous allons commencer par écrire un programme générant la demie pyramide de la figure A.1 (contrairement à l'objectif « final » celle-ci ne comporte qu'un type de symboles). Il suffit, dans ce cas, d'utiliser une boucle permettant de générer successivement les n lignes composant la pyramide et lors de la génération de i-ème ligne, d'afficher $i+1$ symboles. Cette affichage nécessite d'utiliser une deuxième boucles allant de 0 à i (inclus). Ce principe est mis en œuvre dans le listing 13.

Figure A.1 – Demi-pyramide à réaliser dans la première étape de l'exercice 1.4.1.

```
*
**
***
****
*****
******
```

```java
public class FirstHalf {

    public static void main(String[] args) {
        int n = 6;

        for (int i = 0; i < n; i++) {
            String line = "";

            for (int j = 0; j < i + 1; j++) {
                line += symb;
            }

            System.out.println(line);
    }

}
```

Listing 13 – Code permettant de générer la demie pyramide de la figure A.1.

L'étape suivante consiste à générer la moitié gauche de la pyramide (représentée à la figure A.2). Cette ligne est composée de $n-i-1$ espaces et de $i+1$ symboles et peut être construite à l'aide des deux boucles `for` suivantes :

Figure A.2 – Demi-pyramide à réaliser dans la seconde étape de l'exercice 1.4.1.

```
     *
    **
   ***
  ****
 *****
******
```

```java
  for (int i = 0; i < n; i++) {

    String line = "";

    for (int j = 0; j < n - i - 1; j++) {
      line += " ";
    }

    for (int j = 0; j < i + 1; j++) {
      line += "*";
```

```
    }
  }
```

En fusionnant les deux codes, il est alors possible de générer la pyramide complète. Il ne reste plus qu'à ajouter une condition sur la parité du numéro de ligne pour obtenir la pyramide souhaitée. Le code du listing 14 répond parfaitement à la question. Il reste toutefois à voir s'il peut être rendu plus lisible et simplifié, notamment pour enlever toutes traces des étapes intermédiaires. Cette étape consistant à « améliorer » le code sans ajouter de nouvelles fonctionnalités est appelé *refactoring*.

Listing 14 – Code java pour l'exercice 1.4.1 avant refactoring.

```java
public class FirstHalf {

    public static void main(String[] args) {
        int n = 6;

        for (int i = 0; i < n; i++) {
            String line = "";

            for (int j = 0; j < n - i - 1; j++) {
                line += " ";
            }

            for (int j = 0; j < i + 1; j++) {
                if (i % 2 == 0) {
                    line += "*";
                } else {
                    line += "+";
                }
            }

            for (int j = 0; j < i + 1; j++) {
                if (i % 2 == 0) {
                    line += "*";
                } else {
                    line += "+";
                }
            }

            System.out.println(line);
    }

}
```

Le code final (listing 15) comporte deux modifications principales :

— le nom des variables a été modifiées pour être plus informatif ;
— les tests sur la parité de la ligne et le choix du symbole à afficher ont été *factorisés*. Le principale intérêt de cette factorisation est, en plus de réduire la taille du programme, que le choix du symbole à afficher est fait à un endroit unique ; changer le symbole à afficher ou la condition permettant de le choisir (p. ex. en affichant des + que toutes les trois lignes) est beaucoup moins risqué (on ne risque plus de ne modifier qu'une des deux demies pyramides).

```java
public class Pyramide {

    public static void main(String[] args) {

        int n = 7;

        for (int lineNumber = 0;
             lineNumber < n;
             lineNumber++) {

            String symb = "*";
            if (lineNumber % 2 == 0) {
                symb = "+";
            }

            String line = "";

            for (int rowNumber = 0;
                 rowNumber < n - lineNumber - 1;
                 rowNumber++) {
                line += " ";
            }

            for (int rowNumber = 0;
                 rowNumber < lineNumber + 1;
                 rowNumber++) {
                line += symb;
            }

            for (int rowNumber = 0;
                 rowNumber < lineNumber + 1;
                 rowNumber++) {
                line += symb;
            }
            System.out.println(line);
        }

    }

}
```

Listing 15 – Programme de génération de pyramide après *refactoring*.

A.2 Correction de l'exercice 2.2.1

```
1  public static int countTypes(String content) {
2
3    ArrayList<String> types = new ArrayList<>();
4    for (String word : content.split(" ")) {
5      if (!types.contains(word)) {
6        types.add(word);
7      }
8    }
9
10   return types.size();
11 }
```

A.3 Correction de l'exercice 2.2.2

```
1  public static ArrayList<String> findUnique(String content) {
2    ArrayList<String> types = new ArrayList<>();
3    ArrayList<String> reapeatedWords = new ArrayList<>();
4
5    for (String word : content.split(" ")) {
6      if (!types.contains(word)) { //
7        types.add(word);
8      } else {
9        reapeatedWords.add(word);
10     }
11   }
12
13   types.removeAll(reapeatedWords);
14
15   ArrrayList<String> res = new ArrayList<String>();
16   for (String word : content.split(" ") {
17     if (types.contains(word)) {
18       res.add(word);
19     }
20   }
21
22   return res;
23 }
```

La solution proposée repose sur deux étapes :

— la première consiste à construire la liste de tous les mots qui n'apparaissent qu'une seule fois. Pour cela, nous construisons deux listes : la premières contient tous les types apparaissant dans le paramètre `content` (la condition à la ligne 6 permet d'assurer qu'un mot ne sera ajouter qu'une seule fois à la liste `types`) ; la seconde ne contient que les mots apparaissant au moins deux fois (puisqu'un mot n'est ajouté à `repeatedWords` que s'il a déjà été ajouté à `types` et donc déjà été vu). Il suffit alors de retirer de la liste `types` les éléments apparaissant dans `repeatedWords` à l'aide de la méthode `removeAll`.

— la seconde étape consiste à construire la liste contenant la réponse (mots apparaissant une seule fois par ordre d'apparition). Comme la bibliothèque standard n'impose aucune contrainte sur le fonctionnement de la méthode `removeAll`, il est nécessaire de parcourir explicitement les mots dans l'ordre d'apparition pour garantir que cet ordre est conservé. Il s'agit là d'une règle fondamentale en informatique : *explicit is better than impilcit* !

A.4 Correction de l'exercice 2.2.3

```java
import java.util.ArrayList;
import java.util.HashSet;

public class RepeatedArrayList {

    public static boolean allUnique(ArrayList<String> lst) {
        HashSet<String> set = new HashSet<String>(lst);
        return lst.size() == set.size();
    }

    public static void main(String[] s) {

        ArrayList<String> ex1 = new ArrayList<>();
        ex1.add("a");
        ex1.add("b");
        ex1.add("a");

        ArrayList<String> ex2 = new ArrayList<>();
        ex2.add("a");
        ex2.add("b");
        ex2.add("c");

        System.out.println("ex1 : " + allUnique(ex1));
        System.out.println("ex2 : " + allUnique(ex2));
    }
}
```

L'idée principale de cette solution est de « convertir » l'`ArrayList` en `HashSet` et de comparer la taille des collections : si l'`ArrayList` contient des éléments répétés, ceux-ci ne seront copiés qu'une fois dans le `HashSet` et celui contiendra donc moins d'éléments. Une simple comparaison de la taille des deux collections permet alors de répondre à la question.

L'implémentation proposée repose sur la possibilité d'initialiser un `HashSet` à partir d'une autre collection en utilisant le constructeur de la classe qui prend celle-ci en paramètre.

A.5 Correction de l'exercice 2.2.4

```java
import java.util.ArrayList;
import java.util.HashSet;
```

```java
import javafx.util.Pair;

public class CountPairs {

    public static void main(String[] s) {

        ArrayList<String> lst = new ArrayList<>();
        for (String word : "a a b".split(" ")) {
            lst.add(word);
        }

        HashSet<Pair<String, String>> set = new HashSet<>();
        for (int i = 0; i < lst.size(); i++) {
            for (int j = 0; j < i; j++) {
                Pair<String, String> p = new Pair<>(lst.get(i), lst.get(j));
                set.add(p);
            }
        }

        System.out.println("il y a " + set.size() + " paires distinctes");
        System.out.println(set);
    }

}
```