Master Sciences du Langages — Université de Paris

Introduction à la programmation orientée objet en Java

Guillaume Wisniewski

29 avril 2020

Introduction à la POO & à Java

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 29 avril 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.

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
```

```java
public class FirstProgram {

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

}
```

Listing 1 – `Hello Word` en 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 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 division par une variable `b`, une

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.

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
^^Iat Prog.divisionParZero(Prog.java:5)
^^Iat 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 1.4), 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.

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 permettent également d'identifier les programmeurs expérimentés.

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].

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

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

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 :

```
1  int a = 2;
2  int b = 4;
3  int c = a + b;
```

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

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

```java
public class SimpleVariable {

    public static void main(String[] args) {
        // température à la surface du soleil en kelvin
        int temperature = 5778;
        System.out.println("Température du soleil (K)");
        System.out.println(temperature); //
        System.out.println("Température du soleil (deg. C)");
        // conversion : degree C = K - 273.15
        System.out.println(temperature - 273.15); //
    }

}
```

Listing 2 – Exemple de programme utilisant des variables.

```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.

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 entraîne 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
```

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.

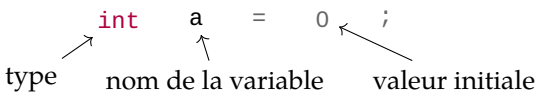
On distingue en java deux sortes de types :

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]

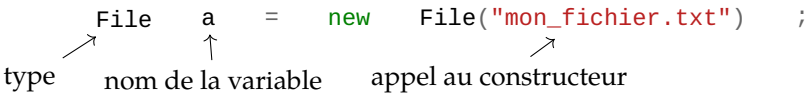
les types complexes qui correspondent à des *objets* pouvant être définis soit dans la bibliothèque standard java, soit directement par le programmeur. Ces types complexes peuvent représenter n'importe quelle entité. L'utilisation des types complexes sera détaillée à la section 1.5.

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.

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 :



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



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'appel au constructeur permet de créer explicitement un nouvel objet en allouant et en initialisant la mémoire nécessaire à celui-ci. Il est également possible d'utiliser un objet créé lors d'un appel à une fonction. Ainsi, si une fonction `getParametersFile()` retournant un objet de type `File` est définie dans le programme, il est possible d'initialiser une référence `paramFile` sans appeler le constructeur :

```
1  File paramFile = getParametersFile();
```

C'est à la fonction `getParametersFile` d'appeler le constructeur et il n'y a pas besoin de créer une instance avant d'appeler la fonction. De manière plus précise, dans l'extrait de programme suivant :

```
1 String maString = "";
2 maString = generateString();
```

deux instances de `String` sont créées : une dans le programme « courant » et une autre dans la fonction `generateString`. Si le programme fonctionne parfaitement, cette « double initialisation » montre que le programmeur ne comprend pas parfaitement comment fonctionne le langage.

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");
```

Les tableaux Le langage java défini un type complexe particulier, les tableaux, qui permet de stocker un nombre donné d'éléments du même type. Les tableaux sont des collections homogènes non dynamiques : ils permettent de stocker une séquence finie d'éléments de même type auquel on peut accéder par leur position ou *indice*. Comme illustré à la figure 1.4, un tableau stocke n éléments, chaque éléments étant associé à un indice compris entre 0 et $n-1$. L'accès à un indice supérieur ou égal à n lèvera une exception `IndexOutOfBoundsException`.

indice :	0	1	2	3
valeur :	`"Oana"`	`"Hanaé"`	`"Ali"`	`"Jiddu"`

Figure 1.4 – Représentation schématique des éléments stockés dans un tableau de `String`

Contrairement aux collections qui seront introduites à la section 2.2, les tableaux ne permettent de stocker qu'un nombre prédéfini d'éléments et il est compliqué d'ajouter ou de supprimer un élément.

En pratique, les tableaux ne sont quasiment plus utilisés, sauf dans certaines fonctions de la bibliothèque standard qui retournent des tableaux. C'est notamment le cas de la méthode `split` particulièrement utile dans les programmes de TAL. Le listing 6 (page 14) donne un exemple utilisant cette méthode et illustre les deux utilisations principales d'un tableau :

- l'accès au i^e élément d'un tableau avec la syntaxe : `tab[i];`
- la possibilité de parcourir toutes les éléments d'un tableau à l'aide d'une boucle *foreach* (cf. §1.4).

C'est pourquoi python, un langage plus récent, ne permet même plus la définition de tableaux mais uniquement de listes. C'est également la raison pour laquelle la présentation des tableaux est extrêmement succincte et ne mentionne pas les différentes manières de créer et d'initialiser un tableau.

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 :

```
if (condition1) {
  // bloc d'instructions exécutés si la condition1
  // est vraie
} else if (condition2) {
  // bloc d'instructions exécutés si la condition2
  // est vraie
} else {
  // bloc d'instructions exécutés si toutes les
  // conditions précédentes sont fausses
}
```

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 :

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

```
  }
}
```

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 1.5 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 1.5

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 2.2). 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 1.6. 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 61)

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

Figure 1.6 – 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

```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.

que 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 1.7, 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.

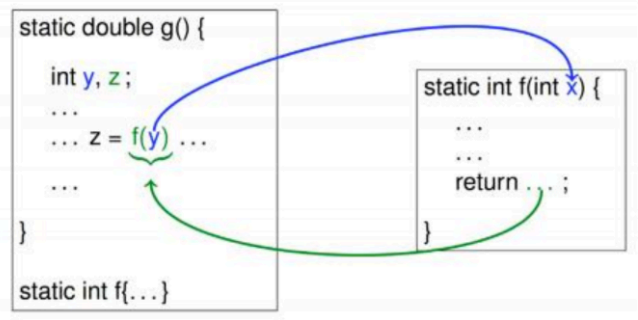


Figure 1.7 – Déroulement du flots d'instructions lors d'un appel à une fonction.

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 :

```
1  r = n / 2;
2  while (abs( r - (n / r) ) > t) {
3    r = 0.5 * (r + (n / r));
4  }
5  System.out.println("r = " + r);
```

et de le comparer à :

```
1  public static double squareRootApproximation(double n) {
2    r = n / 2;
3    while (abs( r - (n / r) ) > t) {
4      r = 0.5 * (r + (n / r));
5    }
6    return r;
7  }
```

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 – Factorisation du code. Source : https://xkcd.com/247/

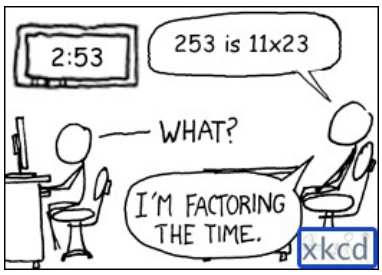


Figure 1.9 – La notion d'abstraction en informatique. source : https://xkcd.com/676/

1.5 Utilisation des objets

En plus des types primitifs, java permet également de manipuler des types complexes. Les variables de type complexe sont généralement appelées des *objets* ou des *instances de classe*. Les types complexes ou *classes* sont 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 : les types complexes enrichissent un langage en définissant de

nouvelles représentations des données. Nous verrons plus en détails au chapitre 3 les motivations de la programmation orientée objet.

Il existe trois types de classes :

- les classes de la bibliothèque standard qui sont diffusées avec toutes les machines virtuelles ;
- les classes fournies dans une bibliothèque tierce, comme par exemple `coreNLP`[7], une bibliothèque de TAL pour 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) avant que les classes et les méthodes qu'elles définit ne puissent être utilisés.
- les classes composant le projet : chaque programmeur peut définir les classes dont il a besoin pour *modéliser* son problème.

7 : `https://stanfordnlp.github.io/CoreNLP/`

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

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

- la création de l'objet soit à l'aide du mot clé `new` qui permet d'obtenir une *référence* sur l'objet soit en appelant une fonction retournant une référence (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. §1.4), une méthode ne peut être appelée que pour modifier ou questionner un objet. Elle est d'une certaine manière « attachée » à une référence et il est impossible d'« appeler » une méthode sans spécifier la référence à laquelle celle-ci s'applique : ainsi dans le programme suivant :

```
1  String firstName = "Meera";
2  String lastName  "Nanda";
3
4  System.out.println(lastName.toUpper());
```

la méthode `toUpper` « s'applique » bien au nom de famille (la variable `lastName`) et à aucune autre chaine de caractères.

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

```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 = " + tot);
  }
}
```

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.

- `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.

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 ;
- 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.

Égalité d'objets Il existe, en java, deux manières de tester l'égalité de deux objets `o1` et `o2` :

- `o1 == o2` va tester l'*égalité des références* et permet de savoir si les deux références représentent bien le même emplacement dans la mémoire de l'ordinateur ;
- `o1.equals(o2)` permet de savoir si les deux objets ont des *contenus* considérés comme identiques, dans un sens qui est défini par le concepteur de chaque classe.

*. On aurait également pu utiliser la méthode `equalsIgnoreCase`.

Dans la quasi totalité des cas, c'est le deuxième type de test qui est utile : de manière contre-intuitive, le programme suivant affichera uniquement `contenus identiques` :

```
1  String s1 = new String("Oluwakemi");
2  String s2 = new String("Oluwakemi");
3
4  if (s1 == s2) {
5    System.out.println("références identiques");
6  }
7
8  if (s1.equals(s2))
9    System.out.println("contenus identiques");
10 }
```

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.

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.

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.

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 2.3

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

```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 .

n'a pas les droits. Les erreurs causées par un problème d'accès à un 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 4

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 7, 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 4. 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 ?";
```

```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.

```
  Files.write(path, content.getBytes("UTF-8"));
```

Comme nous le verrons à la section 2.3, il est nécessaire de spécifier un *encodage* qui sera toujours (ou presque) l'UTF-8.

Accès aux fichiers depuis eclipse Lorsque vous exécutez votre code dans eclipse, le programme principal est exécuté depuis la racine du projet (le premier nom qui apparait dans l'explorateur de package et qui correspond au nom du projet). Il y a donc deux moyens d'accéder à un fichier :

— soit en spécifiant son chemin absolu comme argument de la fonction `Paths.get`;
— soit en spécifiant son chemin relatif par rapport au répertoire `$WORKSPACE/nom_projet` où `$WORKSPACE` est le nom de votre espace de travail choisi lors de la première utilisation d'eclipse.

En pratique, il est préférable pour les « petits » projets de stocker les documents dans le répertoire du projet afin de faciliter l'écriture des chemins d'accès. Pour cela, il faut déplacer le fichier que l'on souhaite utiliser à la racine du projet (le répertoire indiqué par 1 dans la figure 2.1). Le fichier apparait alors dans la structure du projet (2e flèche de la figure 2.1) et il est accessible depuis le programme en indiquant simplement son nom (que la machine virtuelle ira chercher dans le répertoire courant correspondant au répertoire du projet).

Typiquement ceux que vous développerez pendant ce cours. Pour les projets plus gros, les noms de fichiers sont généralement passé en argument du programme.

Figure 2.1 – Ajout d'un fichier dans un projet eclipse : il faut glisser-déplacer le fichier à la racine du projet (1) ; il apparait alors à la suite des fichiers déjà présent (2).

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. 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 des objets de même type, comme dans la figure 2.2. 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 $n-1$.

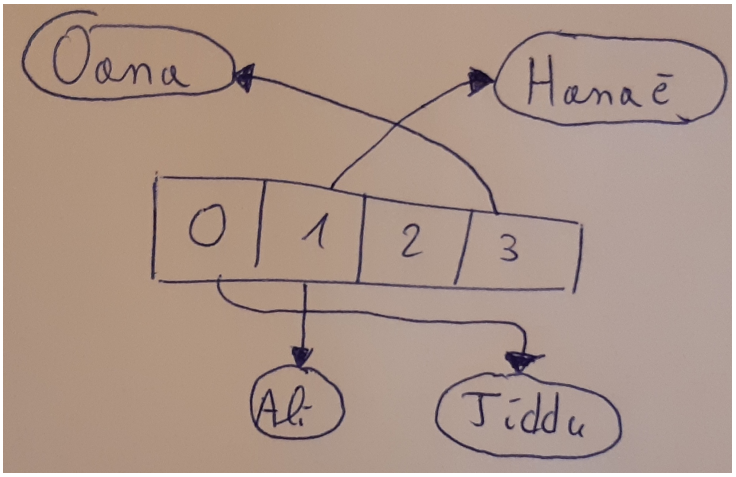


Figure 2.2 – Représentation schématique d'une liste : les 4 éléments sont associés à un indice (compris entre 0 et 3) et il est possible, connaissant un indice d'accéder directement à l'élément correspondant.

L'interface de la classe `ArrayList` fournit de nombreuses méthodes permettant d'accéder aux éléments stockés ou de modifier celle-ci. 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.

L'ajout d'un élément et l'accès à éléments d'une `ArrayList` sont deux opérations « 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 proportionnelle 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 s'écrire de manière plus compacte : `new ArrayList<>(Arrays.asList(s1.split(" ")))`.

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 65.

Cet exercice a uniquement un but pédagogique : l'utilisation d'un ensemble (cf. §6) 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 65.

Composition de structures de données Une `ArrayList` peu stocker des éléments de n'importe quel type complexe. Il est donc tout à fait possible de construire des `ArrayList` d'`ArrayList`. Ainsi, un mot peut être modélisé par une instance de `String`, une phrase comme une liste de mots c'est-à-dire un `ArrayList<String>` et un document comment

```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.

une liste de phrases et donc un `ArrayList<ArrayList<String>>`. Le code suivant montre comment il est possible de construire une telle structure à partir d'un fichier texte contenant une phrase par ligne et dont les mots sont séparés par des espaces :

```java
public static ArrayList<ArrayList<String>> readText(String text) {
  ArrayList<ArrayList<String>> doc = new ArrayList<>();

  for (String line : text.split("\n")) {
    ArrayList<String> sentence = new ArrayList<>();
    for (String word : line.split(" ")) {
      sentence.add(word);
    }

    doc.add(sentence);
  }

  return doc;
}
```

Il est essentiel de noter que cette fonction crée bien une nouvelle instance d'`ArrayList<String>` pour chaque nouvelle ligne du fichier qui est lue (c.-à-d. pour chaque phrase).

Pour vous convaincre que vous avez bien compris le principe des objets, vous pouvez vous demandez ce qui arriverait si la variable `sentence` était déclarée avant la première boucle `for` (p. ex. à la ligne 3).

```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`.

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].

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].

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.

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

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.

```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`.

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.

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

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

à une clé donnée ; mais plusieurs clés peuvent être associées à des valeurs identiques. En pratique, il est possible de considérer les `HashMap` comme 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 :

```
1 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 ;

```
1 if (dico.containsKey(word)) {
2   translatedSentence += " " + dico.get(word);
3 } else {
4   translatedSentence += " *" + word + "*";
5 }
```

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 fi-

```java
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.

chier 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

Les programmes d'aujourd'hui (surtout ceux de TAL) doivent être capables de traiter des textes écrits dans n'importe quelle langue. D'un point de vue informatique, manipuler une grande variété d'alphabets pose de nombreux problèmes qui sont en partie résolus par l'utilisation du standard Unicode. Cette section a pour objectif de présenter les principales notions de ce standard et leur implémentation en java.

Principe de représentation des chaines de caractères

Pour comprendre les différentes entités définies dans le standard Unicode, il est nécessaire de comprendre comment les caractères sont représentés et manipulés par un ordinateur. Un ordinateur ne sait manipuler que des bits (des 0 et des 1) qui peuvent être regroupés pour représenter des nombres. Pour représenter un caractère, il est donc nécessaire d'associer celui-ci à un nombre. C'est le principe des pages de codes *(code page)* qui peuvent être vues comme des dictionnaires (au sens de `HashMap`) associant à un caractère un nombre. La figure 2.3

Une séquence de n bits permet de représenter les entiers compris entre 0 et $2^n - 1$.

montre la totalité de la page ASCII, l'une des première page de code standardisée et la table 2.1 un court extrait de la page Unicode.

Comme le montre ces deux exemples, la représentation d'un caractère met en œuvre trois entités :

— le *caractère* à proprement parlé, une entité abstraite représentant une partie d'un mot. On peut, par exemple, vouloir manipuler le caractère représentant le `diagramme soudé oe`[10] en majuscule.
— le *code* qui lui est associé : pour Œ, ce code est `U+0152` dans la page Unicode, `234` dans la page `ISO 6937`. C'est ce code qui est stocké et manipulé par l'ordinateur.
— le *glyphe* qui est utilisé pour afficher le caractère à l'écran. Un même code peut être rendu (représenté) par plusieurs glyphes : Œ, *Œ*, Œ, œ, ...

10 : C'est la dénomination du caractère o-e entrelacé dans la norme Unicode.

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Figure 2.3 – La table ASCII (en totalité). source : `https://fr.wikipedia.org/wiki/Fichier:ASCII-Table.svg`

La page de code ASCII ne permet de décrire que les 26 lettres de l'alphabet latin, divers signes de ponctuations, des caractères « blancs » (espace, tabulations, retour à la ligne, ...) et des *caractères de contrôle*. Cette page de code ne contient donc que le strict nécessaire pour utiliser un ordinateur en anglais : il ne permet pas de représenter ni les accents, ni les caractères d'autres alphabets (le eszett ß, les caractères hébreux ou cyrilliques, ...). Pour palier ces limites, de très nombreuses pages de code qui ont été définies pour étendre la page ASCII par différents pays (pour représenter les caractères propres à chaque langue ou alphabet) ou différentes entreprises (Microsoft a ainsi développé des pages de code spécifiques à Windows).

Les caractères de contrôle sont des caractères qui ne représentent pas un symbole. Ils sont notamment utilisés pour la mise en page (saut de ligne, tabulation, ...).

La plupart des pages définies sont des *extensions* de la page ASCII qui peut être vue comme le plus petit dénominateur commun des pages existantes (c.-à-d. que les caractères ASCII sont généralement présents dans toutes les pages et associés au même code).

code point	caractère	description
U+0061	a	Latin Small Letter A
U+0062	b	Latin Small Letter B
U+0063	c	Latin Small Letter C
	...	
U+007B	{	Left Curly Bracket
	...	
U+2167	VIII	Roman Numeral Eight
U+2168	IX	Roman Numeral Nine
	...	
U+1F600	😀	Grinning Face
U+1F609	😉	Winking Face
	...	
U+265E	♞	Black Chess Knight
U+265F	♟	Black Chess Pawn

Table 2.1 – Extrait de la table Unicode. Suivant les conventions, le code de chaque lettre est donné en hexadécimal et précédé du préfixe U+.

Pendant longtemps, l'existence des ces standards multiples a posé de nombreux problèmes compatibilités : comme un même code représentait des caractères différents d'une page de code à l'autre, la personne recevant un message ne lisait pas forcément la même chose que celle l'ayant écrit dès que celles-ci n'utilisaient pas la même page de code (p. ex. parce qu'elles utilisaient des systèmes d'exploitation différents ou n'étaient pas dans la même zone géographique). Ces problèmes avaient deux sources principales :

— lorsqu'un ordinateur reçoit un texte (une suite de bits représentant des caractères) il n'a aucun moyen de déterminer quelle page de code il doit utiliser pour interpréter celle-ci. Il faut donc que la page soit explicitement spécifiée (et que la valeur indiquée soit correcte !)

— même lorsque la page de code est connue, il faut disposer d'une police[11] capable d'afficher les caractères décodés

Par exemple, la figure 2.5 représente une page de Wikipédia affichée avec la page de code `Windows Cyrillique` alors que le document a été écrit avec la page de code Unicode : si les caractères « de base » (c.-à-d. ceux présents dans la page ASCII) sont toujours lisibles, les caractères accentués (comme « chinois simplifié » dans la première phrase de la page), certains signes de ponctuations et les caractères chinois sont remplacés par des caractères sans queue ni tête.

Figure 2.4 – Texte affiché avec le mauvais encodage. source : `http://sdz.tdct.org/sdz/asser-du-latin1-a-l-unicode.html`

11 : Une police de caractère contient l'ensemble des glyphes nécessaires à la représentation de l'ensemble des caractères d'un langage, complet et cohérent.


Naxi

Cette page contient des caractГ¯res spГ©ciaux ou non latins. Si certains caractГ¯res de cet article sβ€™affichent mal (carrГ©s vides, points dβ€™interrogation, etc.), consultez la page dβ€™aide Unicode.

Pour les articles homonymes, voir Naxi (homonymie).

Les **Naxi**[1] (chinois simplifiГ© : ɳˈɪ³θ¥ΩζΜ ; chinois traditionnel : ɳˊθ¥ΩζΜ ; pinyin : *nГ xΔ« zГ*) sont l'un des 56 groupes ethniques de Chine. Ils vivent dans le Yunnan.

Au recensement de 2010, ils vivaient principalement dans la prГ©fecture de Lijiango'O240 580) et, dans une moindre mesure, les prГ©fectures voisines : la prГ©fecture autonome tibГ©taine de Diqing (46 402), la PrГ©fecture autonome bai de Dali (4 686) et la PrГ©fecture autonome yi de Chuxiong (759). Certains rГ©sident Г©galement dans la province du Sichuan voisine[2], la PrГ©fecture autonome yi de Liangshan (5 639) et la PrГ©fecture autonome tibГ©taine de GarzГª (771).

Jadis, ce peuple utilisait plusieurs appellations pour s'autodГ©signer[n 1] : *Naxi* ɳˈɪ³θ¥Ω, *Nari* ɳˈɪ²ζΜ¥, *Naheng* ɳˈɪ³ɳ½• ou

Figure 2.5 – La page Wikipédia sur les Naxi (`https://fr.wikipedia.org/wiki/Naxi`) affiché avec une page de code différente de celle avec laquelle le texte a été écrit.

Aujourd'hui la quasi totalité des textes sont encodés en utilisant le standard Unicode. Ce standard permet de représenter les caractères des alphabets de toutes les langues existantes ou ayant existé (y compris le Klingon) mais également des notes de musique, des symboles mathématiques et les émoticônes. Il est régulièrement mis à jour pour prendre en compte les demandes de création de nouveaux caractères[12]. Plusieurs caractéristiques d'Unicode expliquent pourquoi il a réussi à s'imposer et à remplacer progressivement tous les autres standards :

— il permet de représenter tous les caractères possibles et imaginables et de nouveaux caractères sont régulièrement ajoutés ;
— il est compatible avec un grand nombre de pages de code existantes.

12 : Un article daté du 1er février 2020 dans *Le Monde* explique pourquoi et comment un nouveau caractère représentant la fondue a été ajouté en 2020 au standard Unicode et pourquoi le consortium a refusé d'introduire un caractère représentant la raclette. Cet article, ainsi que le dossier déposé pour justifier la nécessité de définir ce caractère (consultable ici) donne un exemple de la teneur des discussions au sein du consortium.

Caractéristiques d'Unicode

Caractères et propriétés Le standard Unicode peut donc être vu comme une grande table associant des caractères à des points de code *(code points)*. La spécification Unicode inclut également des informations sur ces derniers : les *propriétés* du caractère[13]. Pour chaque point de code défini, il est possible d'accéder au nom du caractère, à sa catégorie... mais également à des propriétés liées à l'affichage telles l'utilisation du point de code dans un texte bidirectionnel[14] ou pour le changement de casse du caractère. La figure 2.6 donne un exemple des propriétés associées à un caractère dans le standard Unicode.

13 : Une description détaillée des propriétés est disponible sur la page Wikipédia « Unicode character property »

14 : Ces propriétés permettent de savoir comment afficher un document mélangeant, par exemple, français (qui s'écrit de gauche à droite) et arabe (qui s'écrit de droite à gauche) et comment interagir avec ces documents (par exemple pour sélectionner du texte).

La catégorie d'un caractère décrit la nature de celui-ci. Ces catégories permettent, par exemple, d'identifier les « Lettres », les « Nombres », les « Ponctuations » ou les « Symboles » ; ces catégories sont à leur tour divisées en sous-catégories.

Unicode Data	
Name	LATIN SMALL LIGATURE OE
Block	Latin Extended-A
Category	Letter, Lowercase [Ll]
Combine	0
BIDI	Left-to-Right [L]
Mirror	N
Old name	LATIN SMALL LETTER O E
Index entries	o e, latin small letter ethel e, latin small letter o LATIN SMALL LIGATURE OE SMALL LIGATURE OE, LATIN LIGATURE OE, LATIN SMALL OE, LATIN SMALL LIGATURE
Upper case	U+0152
Title case	U+0152
Comments	ethel (from Old English eðel) French, IPA, Old Icelandic, Old English, ...
See Also	latin small letter ae U+00E6 latin letter small capital oe U+0276
Version	Unicode 1.1.0 (June, 1993)

Figure 2.6 – Un exemple des propriétés Unicode associée au caractère « œ ».

Encodage Une chaine Unicode est une séquence de points de code, qui sont des entiers compris entre 0 et 1 114 111 (`0x10FFFF` en hexadécimal). Cette séquence de points de code doit être stockée en mémoire. Les règles de traduction d'une chaine Unicode en une séquence d'octets sont appelées un *encodage de caractères* ou simplement un *encodage*[15]. Une représentation directe de ces nombres (chaque point de code peut être décrit par un entier codé sur 32 bits) serait extrêmement inefficace : la quasi totalité des textes courants n'utilise qu'un petit sous-ensemble de tous les caractères définis dans le standard Unicode qui correspondent aux premiers points de code et leur représentation sur 32 bits sera essentiellement constituée de 0. Par exemple, avec un codage sur 32 bits, la chaine `java` sera encodée de la manière suivante :

15 : D'un point de vue purement formel et contrairement à une croyance répandue, Unicode n'est pas un encodage mais uniquement une association entre un caractère et un nombre qui doit être encodé pour être représenté en mémoire.

j				a				v				a			
106	0	0	0	97	0	0	0	118	0	0	0	97	0	0	0

et nécessitera donc 16 octets pour être représentée soit quatre fois plus qu'une représentation de la chaine ne stockant pas les octets non nuls (qui correspond dans ce cas à l'encodage de la chaine en ASCII puisque la chaine ne comporte que des lettres de l'alphabet anglais).

Pour éviter ce problème, la norme Unicode définit plusieurs encodage. L'encodage le plus utilisé aujourd'hui est l'UTF-8. Dans cet encodage, les premiers caractères (correspondant à ceux de la table ASCII) sont représentés sur un octet, les suivants sur 2, 3 voire 4 octets.

UTF signifie `Unicode Transformation Format` et regroupe tous les encodages Unicode.

Cette représentation de taille variable complique la plupart des méthodes d'accès aux donnais et de nombreux algorithmes de traitement de chaines. L'UTF-8 présente toutefois plusieurs propriétés intéressantes :

- il peut gérer n'importe quel point de code Unicode ;
- les caractères sont codés exactement de la même manière en UTF-8 et en ASCII ;
- UTF-8 est assez compact. La majorité des caractères couramment utilisés peuvent être représentés avec un ou deux octets.

Ces opérations sont aujourd'hui implémentée en standard dans la plupart des langages de programmation et ce problème n'en n'est plus vraiment un.

Cette propriété permet à de vieux programmes qui n'ont pas été conçus pour un autre encodage que l'ASCII de fonctionner encore si on leur passe du texte en UTF-8.

Il existe d'autres encodage de l'Unicode comme l'UTF-16 ou l'UTF-32 mais ceux-ci sont plus rarement utilisés.

Un ordinateur ne voit pas une chaine de caractères mais une suite de bits. Il a besoin de connaitre l'encodage utilisé, à la fois pour savoir comment interpréter (décoder) cette suite et pour savoir quel caractère associé à celle-ci. Comme pour toutes les pages de code, l'encodage doit être spécifié explicitement pour éviter de voir apparaître des caractères bizarres comme ceux de la Figure 2.5.

Homoglyphes Les homoglyphes sont des caractères dont les glyphes sont visuellement indiscernables mais qui correspondent à des codes différents. Par exemples, le standard Unicode définit les caractères suivants :

- `Latin Capital Letter A` (`U+0041`);
- `Cyrillic Capital Letter A` (`U+0410`);
- `Greek Capital letter Alpha` (`U+0391`);
- `Cherokee Letter Go` (`U+13AA`);
- `Canadian Syllabics Carrier Gho` (`U+15C5`);

Figure 2.7 – Glyphes des différentes lettres représentant un `A` en Unicode. Les lettres sont données dans l'ordre du texte. (source : `http://www.fileformat.info/info/unicode/char/search.htm`)

— `Latin Small Letter Capital A` (`U+1D00`);
— `Lisu Letter A` (`U+A4EE`);
— `Carian Letter A` (`U+102A0`);
— `Mathematical Sans-Serif Capital A` (`U+1D5A0`);
— `Mathematical Monospace Capital A` (`U+1D670`).

Comme le montre la figure 2.7, tous ces caractères sont cependant visuellement très proches. Il est même plus que probable que dans certaines polices une même glyphe soit associée à plusieurs de ces codes. L'existence des homoglyphes est en partie due au fait que les concepteurs d'Unicode ont voulu maintenir une certaines compatibilité avec les pages de code existantes mais également parce qu'il se trouve simplement que dans de nombreux cas des caractères différents ont une représentation proche.

La présence d'homoglyphes posent de nombreux problèmes pour le traitement des textes. Par exemple, pour n'importe quel utilisateur les chaines `voce` et `voce` d'une part et `À` et `À` d'autre part sont parfaitement équivalentes alors, qu'en fait, elles n'ont aucun code en commun : le premier `voce` est composé des caractères latins `U+0076`, `U+006F`, `U+0063`, `U+0065` and `U+0073`; le second mélange des caractères cyrilliques et grecs : `U+03BD`, `U+03BF`, `U+0441` and `U+0435`. Si cet exemple peut paraitre artificiel (et il l'est effectivement), ce type de problème apparait fréquemment lorsqu'un utilisateur saisie un mot ou une phrase dans un alphabet qui ne correspond pas à celui de son clavier.

L'exemple des deux représentations possibles de `À` illustre une particularité du standard Unicode : la possibilité de *combiner* des caractères. Ainsi `À` peut être écrit comme une séquence de deux caractères :

— `A` : `Latin Capital Letter A` (`U+0041`)
— `◌̀` : `Combining Grave Accent` (`U+0300`)

Dans le second cas, c'est le programme gérant l'affichage qui sera chargé de construire la glyphe « à la volée » en combinant la glyphe de l'accent grave avec la glyphe du `A`. Tous les points de code devant être combinés à d'autre caractères sont regroupés dans la catégorie « `Combiner` ». Cette catégorie regroupe les diacritiques usuels mais également différent symbole. La figure 2.8 montre le premier *bloc* de point de code de cette catégorie.

La possibilité de combiner des caractères a plusieurs avantages : elle simplifie la conception des polices de caractères[16] ou de définir des caractères non usuels. Par exemple, le symbole peut facilement être combiné pour donner . Ce dernier symbole correspond aux deux points de code `U+027B` (Latin Small Letter Turned R with Hook) et `U+030D` (Combining Vertical Line Above).

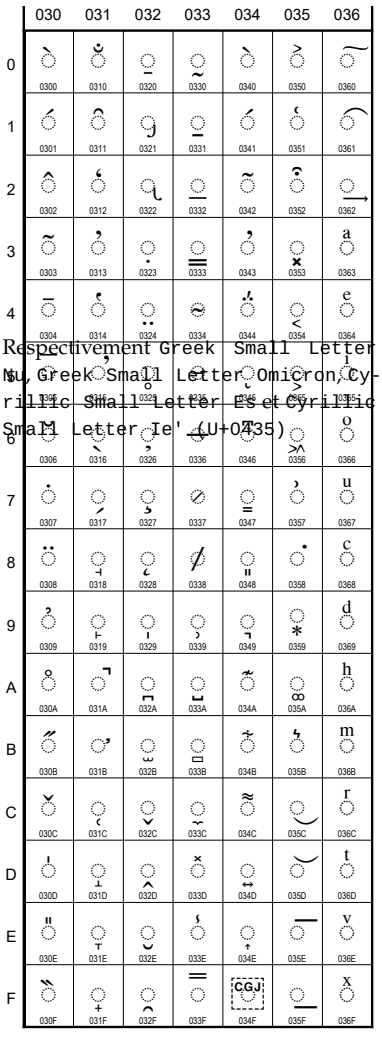


Figure 2.8 – Points de code du bloc *Combining Diacritical Marks*. Extrait du standard Unicode

Normalisation Pour palier les problèmes soulevés par les homoglyphes (ou du moins une partie de ceux-ci), le standard Unicode définit la notion d'*équivalence canonique* : le standard liste explicitement des points

16 : les glyphes des caractères `À`, `È` et `Ò` n'ont pas besoin d'être définie explicitement, mais peuvent être « construits » automatiquement à partir des glyphes des caractères `A`, `E`, `O` et du glyphe représentant un accent grave

caractère	NFD	NFC
Singleton		
Å	A + ◌̊	Å
U+212B	U+0041 U+030A	U+00C5
Ω	Ω	Ω
U+2126	U+03A9	U+03A9
Composition canonique		
ô	o + ◌̂	ô
U+00F4	U+006F U+0302	U+00F4
Composition canonique multiple		
ṩ	s + ◌̣ + ◌̇	ṩ
U+1E69	U+0073 U+0323 U+0307	U+1E69
ḍ̇	d + ◌̣ + ◌̇	ḍ + ◌̇
U+1E0B U+0323	U+0064 U+0323 U+0307	U+1E0D U+0307
q̣̇	q + ◌̣ + ◌̇	q + ◌̣ + ◌̇
U+0071 U+0307 U+0323	U+0071 U+0323 U+0307	U+0071 U+0323 U+0307

Table 2.2 – Exemple de transformations mises en jeu lors de la normalisation vers la forme normale canonique.

de code ou des combinaisons de points de code qui décrivent le même caractère[17]. Il définit également un algorithme de *normalisation* qui transforme une chaine Unicode vers une *forme normale* dans laquelle les formes équivalentes sont transformées en un même point de code (ou série de points de code). La forme normale repose sur deux types de transformations :

- les caractères qui décrivent le même symbole sont tous transformé vers un point de code unique ;
- les caractères de la classe Combiner (décrivant notamment les diacritiques) sont soit *décomposés* soit *composé* quand c'est possible.

17 : Pour être plus précis : le même concept abstrait de caractère

Il existe donc deux formes de normalisation :

- la *forme normale C* (NFC) dans laquelle les points de code de la catégorie `Combiner` sont remplacés par des formes composées dès que possible ;
- la *forme normale D* (NFD) dans laquelle, au contraire, toutes les formes pouvant être décomposées le sont.

La norme Unicode propose un autre type de normalisation reposant sur la notion de *compatibilité* : deux points de code (ou séquence de points de code) seront définis comme compatibles si leur sémantique est jugée suffisamment « proche ». C'est par exemple le cas pour :

- les ligatures : fi (`U+FB01`) est compatible avec f i (`U+0066` + `U+0069`) ;
- les exposants : 2^5 (`U+0032` + `U+2075`) est compatible avec 2 5 (`U+0032` + `U+0035`)
- ſ (`017F`) (s long) est compatible avec s (`U+0073`).

Comme pour la normalisation vers des formes canoniques, il y a deux types de normalisation vers des formes compatibles :

- la forme normale NFKD qui correspond à la forme NFD dans laquelle les points de code compatibles ont tous été transformés vers une même forme ;
- la normalisation NFKC qui correspond à la forme NFC dans laquelle les points de code compatibles ont été transformés vers une même forme.

Le nom des différentes formes normales repose sur les conventions suivantes :

- D : décomposition
- C : composition
- K : comptabilité

La conversion vers une forme normale est nécessaire dès que l'on cherche à comparer des chaines de caractères Unicode (y compris si l'on utilise des méthodes comme `startsWith` ou `endsWith`). Le choix de la forme normale dépend par contre de l'application. Une recommandation courante est d'utiliser la forme NFC qui, plus compacte, permet d'utiliser moins de mémoire et de réduire les temps de traitement.

Représentation des chaines de caractères en java

La classe `String` représente une chaine de caractères Unicode en stockant une liste de points de code. Elle offre une série de méthodes pour manipuler les chaines de caractères ainsi que des méthodes permettant d'accéder aux *code points* et, avec l'aide de la classe `Character`, de manipuler directement ceux-ci.

Pour des raisons historiques[18], les *code points* peuvent être représentés soit par un type spécifique, des `char` soit par un entier (c.-à-d. un `int`). Seule cette dernière solution permet de représenter la totalité des caractères Unicode existant aujourd'hui. Il est **absolument impératif** de ne jamais utiliser de méthodes utilisant des `char` aussi bien comme argument que comme type de retour pour garantir que la présence de caractères « spéciaux » causent une erreur ou un comportement indéterminé du programme.

18 : Le langage java a été défini à une époque où tous les caractères Unicode pouvait être représentés sur 16 bits et où les encodages de taille variable comme l'UTF-8 n'avaient pas encore été inventés. Les `char` ont donc été définis sur 16 bits et ne peuvent donc représenter que les 2^{16} soit les 65 536 premiers caractères Unicode (la norme en contient plus de 245 000 en 2020). Pour des raisons de *compatibilité ascendante* il est impossible de changer cette définition et il existe, pour la quasi totalité des méthodes manipulant des *code points* une version « historique » prenant en argument un `char` et une version « actuelle » dont les arguments sont de type `int`.

Création d'une chaine de caractères Il existe deux moyens de créer une chaine de caractères :

— soit en l'initialisant directement :

```
1    String s = "bébé";
```

L'utilisation de cette syntaxe suppose toutefois d'indiquer au compilateur l'encodage dans lequel l'éditeur sauvegarde le fichier `.java` et que celui-ci permette de représenter tous les caractères utilisés.
Il également possible de spécifier directement un caractère par sont *code point* (en précédant la valeur hexadécimale de celui-ci du « `\u` ») s'il n'existe pas de moyen simple de saisir celui-ci. L'instruction suivante :

Dans `eclipse`, l'encodage du code source est spécifié par la propriété « `Text File Encoding` » accessible dans le menu `Preferences > General > Workspace`. Le choix de l'encodage est automatiquement passé au compilateur.

```
1    String s = "Fa\u00F1ch";
```

permet, par exemple, de définir une chaine de caractères dont la valeur est `Fañch`.

— en lisant les chaines à partir d'un fichier texte (cf. §2.1). Comme expliqué au paragraphe précédent, il est alors nécessaire de connaitre l'encodage du fichier et de spécifier celui-ci explicitement lors de la lecture.

Manipulation des *code points* Unicode Il est possible, étant donné une chaine de caractères `s` d'accéder aux *code points* constituant celle-ci de la manière suivante :

La méthode `codePoints` renvoie un `Stream` (cf. 4). La conversion de ce `Stream` en tableau permet le parcours de celui-ci à l'aide d'une boucle `for` sans avoir à connaitre les méthodes spécifiques à la manipulation des `Stream`

```java
for (int codePoint : s.codePoints().toArray()) {
  // ...
}
```

Plusieurs méthodes de la classe `Character` peuvent être utilisés pour :

- tester les propriétés d'un *code point* : `isDigit`, `isAlphabetic`, `isUpperCase`, `isWhitespace`, ...
- avoir des informations sur le *code point* : `getName` permet d'obtenir le nom du caractère Unicode ; `getType` la catégorie du caractère.

Normalisation et comparaison de `String` La classe `Normalizer` implémente les différentes méthodes de normalisation décrite dans la section précédente. Le listing 13 donne un exemple d'utilisation de cette classe. Ce programme illustre également les problèmes que peut soulever l'utilisation de différentes normalisations Unicode. La sortie de ce programme est :

```
sans normalization :
98 -> LATIN SMALL LETTER B
233 -> LATIN SMALL LETTER E WITH ACUTE
98 -> LATIN SMALL LETTER B
233 -> LATIN SMALL LETTER E WITH ACUTE
taille = 4
--------


avec normalization NFKD:
98 -> LATIN SMALL LETTER B
101 -> LATIN SMALL LETTER E
769 -> COMBINING ACUTE ACCENT
98 -> LATIN SMALL LETTER B
101 -> LATIN SMALL LETTER E
769 -> COMBINING ACUTE ACCENT
taille = 6
--------
égalité de bébé et bébé --> false
bébé commence par 'bé' : false
```

L'exécution de ce programme montre à quel point les méthodes de manipulation des chaines de « haut niveau » sont *fragiles* (les résultats du `equals` et du `startsWith` peuvent paraitre en premier abord erronés) et que l'absence de normalisation des chaines peut entrainer des erreurs pas toujours faciles à détecter.

```java
import java.text.Normalizer;

public class Unicode {

  // pour réduire la longueur des lignes
  public static void print(String s) {
    System.out.println(s);
  }

  public static void main(String[] args) {
    String str = "bébé";

    print("sans normalization :");
    for (int codePt : str.codePts().toArray()) {
      print(codePt + " -> " + Character.getName(codePt));
    }
    print("taille = " + str.length());
    print("--------");

    print("\n\navec normalization NFKD: ");
    String nStr = Normalizer.normalize(str, Normalizer.Form.NFKD);
    for (int codePt : nStr.codePts().toArray()) {
      print(codePt + " -> " + Character.getName(codePt));
    }
    print("taille = " + nStr.length());

    print("--------");
    print(str + " == " + nStr + " = " + str.equals(nStr));
    print(nStr + " commence par 'bé' : " + nStr.startsWith("bé"));
  }
}
```

Listing 13 – Manipulation de chaine Unicode en java.

2.4 Expressions régulières

Définition(s)

Les expressions régulières sont un langage spécialisé permettant de décrire des ensembles de chaines de caractères. Une expression régulière est composée de caractères et de méta-caractères (`. ^ $ * + ? { } [ ] \ | ( )`) dont nous verrons progressivement la signification[19].

l'expression régulière la plus simple est une chaine constituée uniquement de caractères (sans aucun méta-caractère) et décrit un ensemble constitué d'un seul élément : la chaine elle-même. Par exemple, la chaine `meuh` est une expression réguilière décrivant l'ensemble de chaines `{meuh}`. Le méta-caractère `.` permet de représenter n'importe quelle lettre. Ainsi `me.h` représente l'ensemble `{meah, mebh, mech, ..., meAh, meBh, ... me#h, ...}`. Il est possible d'indiquer certains caractères peuvent être répétés à l'aide d'un *quantificateur*. Les quantificateurs les plus répandus sont :

— `?` qui indique un caractère qui existe zéro ou une fois : l'expression régulière `meuh?` décrit l'ensemble `{meuh, meu}`;

Cette section ne donne qu'un aperçu très rapide de la syntaxe des expressions régulières. Une description plus complète est disponible dans la javadoc de la classe `Patter`

19 : Pour qu'un méta-caractère soit interprété comme un caractère normal, il faut *protéger* (en anglais : *escape*) celui-ci en le faisant précéder du symboles `\\`. Ainsi `+` sera interprété comme un quantificateur, alors que `\\+` comme le caractère représentant l'addition.

— `*` qui indique un caractère qui existe zéro ou plusieurs fois : l'expression régulière `meu*h` correspond à l'ensemble (de taille infinie) `{meh, meuh, meuuh, meuuuh, ...}`.
— `+` qui définit un caractère qui existe une ou plusieurs fois : `meu+h` correspond à `{meuh, meuuh, meuuuh, ...}` (mais pas `meh`).

Pour appliquer un quantificateur à plusieurs caractères, il suffit de placer ceux-ci entre parenthèses. Ainsi `c(ab)*d` décrit l'ensemble des chaines `{cd, cabd, cababd, ...}`.

Le symbole `|` permet d'indiquer un choix entre plusieurs alternatives : `(b|m)euh` décrit l'ensemble `{beuh, meuh}`.

Les différents opérateurs peuvent être combinés : `(m|b)eu+h` correspond à l'ensemble `{beuh, meuh, beuuh, meuuh, ...}` et `a.*a` correspond à toutes les chaines commençant et se terminant par `a` (y compris la chaine `aa`).

Utilisation des expressions régulières en java

L'utilisation des expressions régulières en java se fait toujours en deux étapes : une première étape consiste à *définir* l'expression régulière en *compilant* celle-ci. L'expression régulière est alors représentée par une instance de la classe `Pattern`. Par exemple :

```
1  Pattern p = Pattern.compile("a*b|c");
```

Il est alors possible d'utiliser cette expression régulière pour vérifier si une chaine donnée appartient à l'ensemble des chaines représentées par l'expression régulière. Les vérifications sont mises en œuvre par une instance de la classe `Matcher` qu'il est possible de créer en appelant la méthode `matcher` de la classe `Pattern`.

La classe `Matcher` offre deux types de méthodes de recherche :

— la méthode `matches` : qui vérifie que l'*ensemble* de la ligne corresponde au motif décrit par l'expression régulière
— la méthode `find` : qui vérifie si une *partie* de la chaine correspond au motif.

Le listing 14 montre comment utiliser les expressions régulières pour vérifier qu'un fichier stocke bien un dictionnaire en respectant la syntaxe suivante :

— il y a un entrée par ligne
— chaque entrée décrit une clé et une valeur séparée soit par un double point soit par un signe égal.

Groupes Les méthodes `find` et `matches` permettent d'obtenir plus d'informations que le simple fait de savoir si la chaine appartient ou non à l'ensemble des chaines décrites par une expression régulière. En particulier il est possible d'identifier des *groupes* dans une expression régulière en plaçant les caractères correspondant entre parenthèses. Par exemple, l'expression régulière `a(bc)d(ef)` identifie 2 groupes.

Les méthodes de recherche permettent d'obtenir les indices du texte auxquels correspondent les groupes. Ainsi, en utilisant la méthode

```java
public class CheckFile {

  public static void main(String[] args) throws IOException {
    Pattern p = Pattern.compile(".*(:|=).*");

    int lineCount = 1;
    int nError = 0;
    for (String line : Files.readAllLines(Paths.get(args[1]))) {
      Matcher m = p.matcher(line);
      if (!m.matches()) {
        System.out.println("la ligne " + lineCount + " n'est pas conforme");
        nError += 1;
      }
      lineCount += 1;
    }
    System.out.println("il y a " + nError + " erreurs");
  }

}
```

Listing 14 – Programme vérifiant qu'un fichier stocke bien des associations (clé, valeur).

`find` sur la chaine `aabcdefgg`, il est possible de déterminer que le premier groupe[20] commence à la position 2 de la chaine, et le second à la position 5 :

20 : Le numéro d'un groupe correspond au nombre de parenthèses ouvrantes que l'on a rencontré en parcourant la chaine de gauche à droite.

```java
  Pattern p = Pattern.compile("a(bc)d(ef)");
  Matcher m = p.matcher("aabcdefgg");
  System.out.println("début groupe 1 : " + m.start(1));
  System.out.println("début groupe 2 : " + m.start(2));
```

Les groupes sont particulièrement utiles pour capturer des parties d'une chaine lorsqu'ils utilisés avec des quantificateurs ou le méta-caractère « `.` ». Ainsi, dans l'exemple du listing 14, il est possible de récupérer les valeurs de la paire en utilisant l'expression régulière `(.*)(=|:)(.*)` : la clé est capturée par le groupe d'indice 1 et la valeur par le groupe d'indice 3.

Quantificateurs non gloutons Lorsqu'ils sont utilisés pour capturer des groupes, le comportement par défaut des quantificateurs est de capturer autant de caractère que possible. Ce comportement peut être problématique

Par exemple, l'expression régulière `<(.*)>` ne permet pas de capturer le nom de toutes les balises dans `"<html><head><title>Title</title>"` : le premier caractère de l'expression régulière sera mis en correspondance avec le chevron ouvrant du `html` et le `.*` consomme le reste de la chaine jusqu'au chevron fermant du `title`.

Pour capturer le nom des différentes balises, la solution consiste à rendre les quantificateurs non gloutons en les faisant suivre du méta-caractère `?` de manière à ce qu'ils effectuent une correspondance aussi petite que possible. Le listing 15 illustre la différence entre quantificateurs gloutons et non gloutons. La sortie du programme est :

```java
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class TestRegexp {

  public static void main(String[] args) throws IOException {
    Pattern pGreedy = Pattern.compile("<(.*)>");
    Pattern pNonGreedy = Pattern.compile("<(.*?)>");
    String line = "<head><title>Title</title>";

    System.out.println("Greedy");
    System.out.println("------");
    Matcher mGreedy = pGreedy.matcher(line);
    while (mGreedy.find()) {
      System.out.println(mGreedy.group(1));
    }

    System.out.println("\nNon-Greedy");
    System.out.println("----------");
    Matcher mNonGreedy = pNonGreedy.matcher(line);
    while (mNonGreedy.find()) {
      System.out.println(mNonGreedy.group(1));
    }
  }

}
```

Listing 15 – Illustration de la différence entre quantificateurs gloutons et non gloutons.

```
Greedy
------
head><title>Title</title

Non-Greedy
----------
head
title
/title
```

Exemple Le listing 16 montre comment il est possible de définir une expression régulière identifiant les références dans un document LaTeX[21]. La définition de l'expression régulière appelle plusieurs commentaires :

- il est nécessaires d'*échapper* les méta-caractères \, { et } pour que ceux-ci soient interprète comme des caractères normaux ;
- le quantificateur * est utilisé en mode glouton pour sans quoi, il capturerait l'ensemble des caractères compris entre la première accolade ouvrante et la dernière[22].

21 : Les références sont indiquées par la commande LaTeX `\ref{id}` où `id` est un identifiant composé des caractères « usuels ».

22 : En mot glouton, la capture s'arrête dès qu'une accolade ouvrante est rencontrée

```java
public class TestRegexp {

  public static void main(String[] s) {
    Pattern p = Pattern.compile("\\\\ref\\{(.*)\\}");
    String content = "La figure \ref{fig:fig_name} que nous avons vu à la section \ref{sec:pouet}"

    Matcher m = p.matcher(content);
    while (m.find()) {
      System.out.println(m.group(1));
    }
  }

}
```

Listing 16 – Exemple d'utilisation des expressions régulières en java.

Programmation orientée objet 3

La programmation orientée objet est un *paradigme*[1] de programmation qui offre une nouvelle manière d'analyser un problème et de concevoir le code pour le résoudre. Ce chapitre est organisé de la manière suivante : les deux premières sections exposeront les problèmes soulevés par les méthodes de développement classiques (c.-à-d. non-orienté objets) et les principes mis en œuvre par la programmation orientée objet pour y répondre. La section suivante présentera les éléments de syntaxe java utilisés pour définir de nouvelles classes. La section 3.4 expliquera comment ces éléments permettent de résoudre les problèmes identifiés au début du chapitre. Finalement la section 3.5 détaillera certains aspects particuliers de la syntaxe java relatifs à la définition des objets et des classes.

1 : Les deux autres « grands » paradigmes de programmation sont la programmation impérative qui correspond, en première approximation, aux programmes vus jusqu'à présent et la programmation fonctionnelle que l'on retrouve dans des langages comme Haskell. De plus en plus de langages, comme Python et, dans une certaine mesure, java sont multi-paradigmes.

3.1 Pourquoi la programmation orientée objet ?

Tous les programmes vus et écrits jusqu'à présent étaient constitués de *fonctions* qu'il était possible de combiner pour réaliser un « traitement » et résoudre un problème donné. Ces fonctions sont un moyen de décrire (à l'aide des instructions qui la composent) et de nommer (par le nom de la fonction) une « méthode de calcul » *(computational process)*. La définition d'une fonction augmente l'expressivité du langage en permettant de définir de nouvelles instructions et fournit ainsi un premier niveau d'*abstraction* en autorisant une programmation à un plus haut niveau conceptuel dans lequel le programmeur ou la programmeuse indique uniquement ce qu'elle ou il fait (p. ex. calculer la racine carré d'un nombre ou chercher le plus grand éléments d'une liste) plutôt que comment il ou elle le fait (en détaillant explicitement les instructions nécessaires pour réaliser ces deux actions).

La programmation orientée objet offre un second moyen d'améliorer l'expressivité d'un langage et d'atteindre un nouveau niveau d'abstraction en offrant la possibilité d'enrichir le langage par de nouveaux types de données et des outils garantissant que ces types sont utilisés correctement et limitant le risque d'erreur lors de la conception d'un programme.

Il est possible d'illustrer l'intérêt de la programmation orientée objet en considérant l'exemple d'un programme devant manipuler des nombres rationnels[2]. Plusieurs manières de représenter un rationnel dans un programme sont imaginables : un rationnel étant composé de deux entiers, le numérateur et le dénominateur, il suffit de stocker ceux-ci soit dans deux variables distinctes soit dans une liste ou un tableau. Dans ce dernier cas[3], l'addition entre deux rationnels est définie par :

2 : Un rationnel est, en mathématique, un nombre qui peut s'écrire comme le quotient entre deux entiers relatifs (une fraction dans le langage courant). Les nombres rationnels possèdent un ensemble de règle de calcul. Par exemple : $\frac{a}{b} + \frac{c}{d} = \frac{a \times d + c \times b}{b \times c}$.

3 : Ce cas a été choisi car il est difficile, d'écrire une fonction additionnant deux rationnels lorsque ceux-ci sont représentés par deux variables distinctes dans la mesure où une fonction ne peut retourner qu'une seule valeur en java.

```java
public static int[] addRationnal(int[] r1, int[] r2) {
  int[] res = new int[2];
  res[0] = r1[0] * r2[1] + r2[0] * r1[1];
  res[1] = r1[1] * r2[1];

  // simplification de la fractionx
  BigInteger b1 = BigInteger.valueOf(res[0]);
  BigInteger b2 = BigInteger.valueOf(res[1]);
  int gcd = b1.gcd(b2).intValue();

  res[0] = res[0] / gcd;
  res[1] = res[1] / gcd;

  return res;
}
```

Ces deux représentations posent de nombreux problèmes et leur utilisation risque d'entraîner de nombreuses erreurs ou, au moins, de demander au programmeur ou à la programmeuse une attention particulière pour éviter celles-ci. Par exemple, dans la représentation par deux entiers, l'association entre le numérateur et le dénominateur n'étant pas explicite, la·le programmeur·euse devra s'assurer en permanence de maintenir celle-ci et de toujours considérer le dénominateur correspondant au bon dénominateur lors de la manipulation d'un rationnel ; dans la représentation par une liste, elle ou il devra se souvenir en permanence de la position (indice) du numérateur et du dénominateur. De plus, les rationnels étant décrits par des types standards de java, le compilateur n'a aucun moyen de vérifier que l'utilisation d'une variable est valide : il est, par exemple, possible d'appeler la fonction `addRationnal` avec n'importe quel tableau d'entiers et le compilateur ne peut pas garantir que lors de l'appel de cette fonction, l'argument est bien un tableau comportant deux éléments et que l'élément décrivant le dénominateur n'est pas nul.

Le risque de faire des erreurs de ce type (qui peut de premier abord paraître extrêmement faible) est exacerbé par le fait que le développement d'un logiciel fait aujourd'hui généralement intervenir plusieurs·es programmeurs·euses sur une longue période qui peut se compter en mois ou en années : s'il parait évident pour la personne prenant cette décision que le dénominateur sera toujours stocké à l'indice 0, ce n'est pas nécessairement le cas pour un·e développeur·euse qui rejoindrait le projet en cours de route ou même pour la personne ayant pris la décision si celle-ci doit modifier son code plusieurs mois après l'avoir écrit. La programmation orientée objet a pour objectif de permettre au programmeur ou à la programmeuse de définir ses propres types afin que le compilateur puisse vérifier la validité des opérations et de limiter les erreurs de manipulation des données en fournissant des mécanismes de *protection de l'information*.

La représentation d'un rationnel par deux entiers ou par un tableau d'entiers présente un troisième inconvénient majeur : la *propagation des modifications*. Imaginons qu'il ait été décidé au début du projet que les rationnels seraient représentés par un tableau de deux entiers `int[]`.

Tou·t·es les programmeur·euse·s peuvent alors commencer à travailler sur leur partie du programme et écrire le code dont ils ou elles sont responsables. Plusieurs mois plus tard, le programme est prêt et les premiers tests « réels » sont effectués. Il apparaît alors que dans de rares cas, qui n'avaient pas été détectés lors de la réflexion initiale, les nombres manipulés sont tellement grands qu'ils ne peuvent pas être représentés par des `int`[4]. Il est alors nécessaire de modifier la manière dont les rationnels sont représentés, ce qui implique de changer tous les endroits du code où un rationnel est créé (pour changer le type de la variable stockant celui-ci) mais également la signature de toutes les fonctions utilisant un rationnel (il faut que les arguments des fonctions soient adaptés à la nouvelle définition) et bien souvent le code de celle-ci. En effet, comme la manière dont les données sont stockées est exposée (connue) les programmeurs·euses vont directement manipuler celles-ci et créer des *dépendances* entre le code qu'ils·elles écrivent et la représentation des données ; dès lors que cette dernière est modifiée, les changements devront être propagés à l'ensemble du code.

4 : Il y a en java trois types de données permettant de représenter les entiers : `short` qui permet de représenter les entiers entre $[\![-2^{15}, 2^{15}+1]\!]$, `int` entre $[\![-2^{31}, 2^{31}+1]\!]$ et `long` entre $[\![-2^{63}, 2^{§3}+1]\!]$. Les `int` couvrent la quasi totalité des cas courant.

En résumé, la programmation orientée objet a trois objectifs principaux :

— permettre à la programmeuse ou au programmeur de définir de nouveaux types de sorte que le compilateur puisse vérifier la validité des opérations effectuées dans le programme ;
— assurer la cohérence et la protection des informations ;
— permettre l'évolution de la représentation des données (pour corriger une erreur ou ajouter une nouvelle fonctionnalité) sans que celle-ci nécessite de modifier tout le programme.

3.2 Encapsulation

La solution apportée par la programmation orientée objet aux problèmes décrits dans la section précédente repose sur la notion d'*encapsulation*[5]. L'encapsulation a pour objectif de *séparer* la manière dont un type est manipulé de la manière dont il est construit, généralement en combinant d'autres types. Cette approche correspond à un changement de point de vue fondamental : un type n'est plus défini par la manière dont il est représenté (p. ex. un rationnel par un tableau de deux entiers, un·e étudiant·e par un nom, un prénom, une adresse mail et la liste des cours qu'il ou elle suit), mais par les opérations qui peuvent être effectuées sur celui-ci (p. ex. envoyer un mail à l'étudiant·e, déterminer le nombre de cours qu'il ou elle a validé, ...).

5 : Il est tout à fait possible de mettre en œuvre l'encapsulation dans des langages non orientés objets. L'intérêt des langages orientés objets est de fournir une syntaxe permettant au compilateur de vérifier que les principes de l'encapsulation sont bien respectés.

L'ensemble des opérations qu'il est possible de réaliser sur un type est appelé *interface*. L'interface doit être distinguée de *implémentation* de celle-ci, qui désigne le code permettant de réaliser ces opérations. L'interface peut être vue comme une spécification ou un contrat passé entre la personne utilisant un type et la personne concevant celui-ci détaillant les services fournis par le type et son comportement. Comme illustré à la figure 3.1, un langage orienté objet permet de garantir que seule l'interface est *visible* et que les détails de l'implémentation restent *cachés* : un·e programmeur·euse utilisant un nouveau type ne manipulera alors que l'interface de celui-ci et n'aura accès ni aux données ni à la manière dont elles sont stockées.

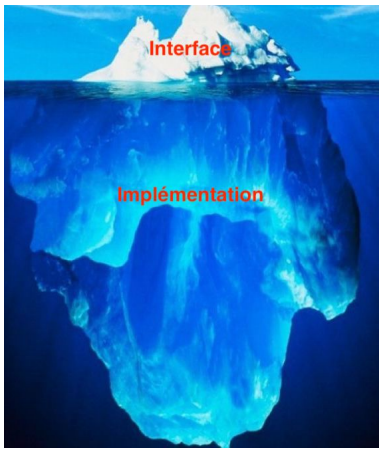


Figure 3.1 – Illustration du principe de l'encapsulation : l'interface et implémentation sont séparées et seule l'interface est visible.

Distinguer l'interface de l'implémentation offre plusieurs avantages : cette séparation permet notamment à un·e programmeur·euse souhaitant utiliser un type de s'abstraire de détails de fonctionnement de celui-ci. Il en sera plus nécessaire pour utiliser un type de comprendre la manière dont les données sont stockées ou comment fonctionnent certaines fonctions, mais uniquement de maîtriser l'interface de celui-ci : comme pour la définition de fonctions, limiter l'accès à l'interface accroît la lisibilité du code et permet une programmation de plus haut niveau.

De plus comme le·a programmeur·euse n'aura pas accès directement aux données, mais uniquement à des méthodes permettant de manipuler celle-ci, la cohérence des données sera plus facile à assurer : il suffit de s'assurer que les méthodes respectent bien la définition des données pour garantir que celles-ci seront correctement utilisées dans l'ensemble du programme. Par exemple, dans la définition d'un type représentant un·e étudiant·e, il sera possible de garantir que le nom sera toujours stocké en majuscule (puisqu'il sera impossible d'accéder directement à celui-ci et donc de changer sans casse), ce qui facilitera la recherche (plus besoin de prendre en compte la casse dans la comparaison) ou permettra d'uniformiser facilement la présentation.

L'encapsulation permet également d'éviter la propagation des modifications : un·e programmeur·euse ne pouvant plus accéder à l'implémentation d'un type mais uniquement à l'interface de celui-ci, le code qu'il ou elle écrira ne dépendra que de l'interface. Il est donc possible de faire autant de modifications que souhaitées à l'implémentation sans avoir d'impact sur le reste du code tant que l'interface du type n'est pas modifiée.

En programmation orientée objet, l'encapsulation est mise en œuvre grâce à la notion de classe : un nouveau type sera définie par l'écriture d'une classe le représentant et qui permettra de regrouper dans une même unité (de manière pratique, dans un même fichier) la représentation des données, la définition de l'interface et l'implémentation de celle-ci. Les langages orientés objet offrent différentes construction permettant de garantir la séparation de l'interface et de l'implémentation ainsi que la protection des données qui seront détaillés dans la section suivante.

3.3 Définition d'une classe en java

Java peut être étendu en définissant de nouveaux de types à l'aide de classes. Même si leurs syntaxes est très proche, les classes définissant de nouveaux types doivent être distinguées des classes définissant des unités de compilation comme toutes celles vues jusqu'à présent.

Pour illustrer les différents éléments de la syntaxe java permettant de définir un nouveau type, nous allons considérer deux exemples : la définition d'un type définissant une paire d'entiers et d'un type décrivant un·e étudiant·e. L'interface du premier type comportera trois opérations :

— une méthode pour ajouter un entier aux deux composantes de la paire (c.-à-d. obtenir la paire $\langle a+\delta, b+\delta\rangle$ à partir de la paire $\langle a, b\rangle$) ;
— une méthode pour savoir si une paire est plus grande ou égale à une autre (suivant les conventions python la paire $\langle a, b\rangle$ est plus grande que la paire $\langle c, d\rangle$ si, et seulement si $a > c$ ou $a = c \wedge b \geq d$) ;
— une méthode testant l'égalité de deux paires ;
— une méthode représentant une paire sous forme d'une chaine de caractère (par exemple pour afficher celle-ci à l'aide de la méthode `System.out.prinln`).

L'interface du type représentant un·e étudiant·e permettra de lui envoyer un mail et de vérifier si celui-ci ou celle-ci a validé son année.

Déclaration de classe La première étape pour définir une classe consiste à déclarer celle-ci en spécifiant dans un fichier :

```
1 public class IntPair {
2 }
```

Comme d'habitude, une classe doit être définie dans un fichier portant le même nom que la classe (dans ce cas : `IntPair.java`).

La déclaration d'une classe est suffisante pour définir un nouveau type dont le nom sera le nom de la classe : dès que la classe est déclarée (c.-à-d. dès qu'un fichier comportant le code précédent est créé), il est possible de définir, dans toutes les autres classes du projet, des variables de type `IntPair` (ou `Etudiant`) à l'aide de la syntaxe habituelle : `IntPair p`. Il est également possible de définir des fonctions prenant en argument des variables de ce type : une fonction permettant d'envoyer un mail à une liste d'étudiants aura, par exemple, pour signature :

```
1 public static void sendMail(ArrayList<Etudiant> lst)
```

Définition des attributs Les classes ainsi définies sont des coquilles vides ne permettant ni de stocker des informations, ni de définir des opérations sur celles-ci. La deuxième étape dans la définition d'une classe consiste à définir les *attributs* de celle-ci. Les attributs sont des variables attachées à une instance de la classe et permettant de stocker toutes les informations nécessaires pour décrire une variable de ce type. Par exemple, pour une variable de type `IntPair` il sera nécessaire de représenter et de stocker les deux entiers constituant la paire ; pour une variable de type `Etudiant`, il faudra stocker le nom, le prénom, l'adresse mail de l'étudiant·e ainsi que la liste des cours qu'elle ou il suit. Il s'agit là, bien entendu, d'une modélisation simplifiée : il est, en pratique, possible de définir autant d'attributs que nécessaire au bon fonctionnement du programme (p. ex. le numéro d'étudiant, la première année d'inscription à l'Université, l'adresse postale, ...).

Les attributs d'une classe sont déclarés au début de la classe, directement après la déclaration de celle-ci en spécifiant trois éléments : le nom de l'attribut, son type et son *mode d'accès*. Par exemple pour la classe `IntPair` :

```java
public class IntPair {
  private int first;
  private int second;
}
```

Ce code permet de définir un type `IntPair` pouvant stocker deux entiers appelés `first` et `second`. Le mode d'accès, dans ce cas `private` indique que les attributs ne sont ni visibles ni accessibles à l'extérieur de la classe : il est impossible d'écrire du code lisant ou modifiant la valeur de l'attribut si ce code n'est pas situé dans la définition de la classe

Par exemple, le fragment de code suivant :

```java
public class PairTest {
  public static void main(String[] s) {
    IntPair p = new IntPair(2, 3);
    // accès en lecture à un attribut
    System.out.println(p.first);
    // accès en écriture : affecte une valeur
    // arbitraire à un attribut
    p.first = 7;
  }
}
```

générera deux erreurs de compilation :

```
PairTest.java:4: error: x has private access in IntPair
        System.out.println(p.first);
                           ^
PairTest.java:5: error: x has private access in IntPair
        p.first = 5;
         ^
2 errors
```

Le compilateur interdit aussi bien de fixer la valeur d'un attribut (avec l'instruction `p.first = 7`) que d'accéder à sa valeur pour l'afficher, ces deux instructions n'étant pas dans le code de la classe `IntPair`. De manière général, les attributs ont toujours `private` comme mode d'accès : comme expliqué à la section 3.4, ce choix permet de garantir l'encapsulation des données.

Pour la classe `Etudiant`, en supposant qu'il existe une classe `Lecture` et une classe `Email` décrivant, respectivement, un cours et une adresse mail, la définition des attributs pourra être :

```java
public class Etudiant {
  private String firstName;
  private String lastName;
  private Email mail;
  private ArrayList<Lecture> lectures;
}
```

Comme l'illustre cet exemple, les attributs peuvent être de n'importe quel type, y compris être des instances d'autres classes définies dans le programme.

Constructeur Les constructeurs constituent le 3e élément d'une classe. Ils permettent de spécifier la manière dont les différents attributs d'une classe doivent être initialisés. La définition d'un constructeur est nécessaire puisque le mode d'accès des attributs (`private` comme expliqué dans le paragraphe précédent) interdisent à la programmeur ou à la programmeuse de fixer leur valeur directement.

Les constructeurs sont des méthodes pouvant avoir un nombre de paramètres arbitraires et ayant exactement le même nom que la classe[6] et n'ayant pas de type de retour. Comme toutes les méthodes ils sont précédés d'un mode d'accès (en général `public`). Il est possible de définir plusieurs constructeurs si ceux-ci ont des paramètres dont les types différent.

6 : C'est le seul cas où un nom de fonction peut commencer par une majuscule dans les conventions java.

Un premier constructeur de la classe `IntPair` est[7] :

7 : La définition du constructeur se place généralement juste après la définition des attributs dans le bloc correspondant à la définition de la classe

```
1 public IntPair(int x, int y) {
2   this.first = x;
3   this.second = y;
4 }
```

Ce constructeur permet à un·e utilisateur·rice de la classe de créer une paire d'entiers en spécifiant la valeur de ceux-ci :

```
1 IntPair p = new Pair(2, 3);
```

Le mot-clé `new` permet d'appeler le constructeur comme une méthode « normale ». Ce constructeur initialise la valeur des attributs en « copiant » les valeurs passées en paramètre. Un constructeur doit toujours initialiser tous les attributs d'une classe soit en leur donnant une valeur fixée par la personne créant l'instance, soit en leur donnant une valeur par défaut. La définition du constructeur montre que les attributs peuvent être manipulés comme n'importe quelle variable. Il est courant (c.-à-d. obligatoire pour la programmeuse ou le programmeur souhaitant rendre son code lisible) d'utiliser la syntaxe `this.attrib` pour distinguer l'accès à un attribut de l'accès une variable locale. Une manière plus courante de définir ce constructeur est d'ailleurs :

```
1 public IntPair(int first, int second) {
2   this.first = first;
3   this.second = second;
4 }
```

Ce constructeur initialise l'attribut désigné par `this.first` à partir de la valeur de la variable locale `first`[8] : même s'ils ont des noms très proches, `first` et `this.first` correspondent bien à deux éléments distincts du programme.

8 : Les paramètres d'une fonction sont des variables locales dont la valeur est fixée lors de l'appel.

Comme indiqué précédemment, il est tout à fait possible de définir plusieurs constructeurs dans une classe. Un deuxième constructeur de la classe `IntPair` est :

```java
public IntPair(IntPair other) {
  this.first = other.first;
  this.second = other.second;
}
```

Ce constructeur, appelé *constructeur de copie* initialise une nouvelle instance de `IntPair` en copiant la valeur des attributs d'une instance existante. Ce constructeur illustre la possibilité d'accéder aux attributs d'une autre instance d'une classe mais uniquement dans la définition de celle-ci.

Un constructeur possible pour la classe `Etudiant` est :

```java
public Etudiant(String nom, String prenom, Email mail) {
  this.lastName = nom;
  this.firstName = prenom;
  this.mail = mail;
  this.lectures = new ArrayList<>();
}
```

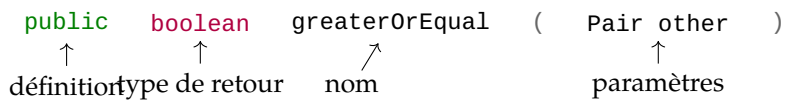
L'objectif d'un constructeur est d'initialiser tous les attributs, ce qui signifie d'attribuer une valeur à toutes les valeurs de type primitif et d'assurer qu'il y ait bien eu une instanciation (un appel au constructeur) pour toutes les variables de type complexe.

L'exemple précédant illustre les deux différents types d'initialisation possible :

— soit à partir d'une valeur spécifiée par l'utilisateur·rice ;
— soit à une valeur « par défaut ».

Dans le cas de la classe `Etudiant` la valeur de tous les attributs, excepté `lectures`, devra être spécifiée par l'utilisateur·rice (il y n'y a aucun moyen de connaitre leurs valeurs autrement qu'en demandant à la personne créant l'instance). Le dernier attribut sera initialisé à une valeur par défaut, ici une liste vide. Si l'ajout des cours peut être fait par une ou plusieurs autres méthodes de la classe, il est nécessaire que l'initialisation soit faite dans le constructeur afin que chaque méthode n'ait pas à vérifier que le constructeur de la classe `ArrayList` ait bien été appelé et le fasse le cas échéant.

Méthodes Les méthodes sont le dernier élément de la définition d'une classe. Elles permettent de définir les opérations qu'il est possible de réaliser sur les instances de la classe. Les méthodes sont définies comme une fonction (cf. §1.4) sauf que leur déclaration ne contient pas le mot clé `static` :



Le mode d'accès des méthodes peut être soit `private` soit `public`. Une méthode publique est visible (c.-à-d. peut être appelée) dans n'importe quelle classe du projet, alors qu'une méthode privée ne peut être

appelée que dans le code de la classe. Les méthodes privées sont utilisées pour améliorer la lisibilité du code.

Il est possible de définir autant de méthodes nécessaires pour implémenter toutes les opérations associées au type. En pratique, l'interface d'un type est généralement définie par l'ensemble des méthodes publiques de la classe.

Il est possible, dans le code d'une méthode, d'utiliser toutes les instructions java et d'accéder aux attributs comme le montre l'exemple de la méthode `toString` :

```
1 public String toString() {
2   return "<" + this.first + ", " + this.second + ">";
3 }
```

La définition de cette méthode est nécessaire pour afficher les valeurs des attributs, ceux-ci n'étant pas directement accessibles. Une fois le code de cette méthode ajoutée à la classe, il est possible de l'appeler à l'aide de la syntaxe habituelle :

```
1   IntPair p1 = new IntPair(1, 98);
2   IntPair p2 = new IntPair(12, 2);
3   System.out.println(p2.toString());
```

Même si cette méthode n'a pas d'arguments, elle est *attachée* à une instance particulière (ici `p`) et ce sont les attributs de cette instance qui seront utilisés pour produire le résultat : `<12, 2>`.

Il est possible de définir des méthodes avec des paramètres, comme la méthode `inc` :

```
1   public void inc(int howMuch) {
2     this.first += howMuch;
3     this.second += howMuch;
4   }
```

Cette méthode sera appelée de la manière suivante :

```
1   IntPair p1 = new IntPair(1, 98);
2   IntPair p2 = new IntPair(12, 2);
3   p1.inc(1);
```

Après l'exécution de ces lignes, la référence `p1` désignera la paire $\langle 2, 99 \rangle$. La méthode `inc` peut modifier sans problèmes les attributs puisque celle-ci est également défini dans la classe.

Cette méthode peut également être définie de la manière suivante :

```
1   public IntPair inc(int howMuch) {
2     return new IntPair(this.first + howMuch, this.second + howMuch);
3   }
```

Cette implémentation crée une nouvelle paire dont les éléments correspondent au résultat de l'opération. Pour incrémenter une référence donnée, il faut désormais utiliser le code :

```
p1 = p1.inc(1);
```

Cette définition de la méthode `inc` permet de ne pas modifier les valeurs des attributs. Une classe dont les attributs ne sont jamais modifiés après la création de celle-ci (c'est-à-dire qu'aucune des méthodes de la classe ne modifie les attributs) est qualifiée de *immuable* (*immutable* en anglais) et *variable* (*mutable* en anglais) dans le cas contraire. Utiliser des objets immuables permet d'améliorer la lisibilité du code et les performances de celui-ci. Ils peuvent également être utilisés sans problème dans des programmes *multi-threadés*. De manière générale toutes les classes doivent être immuables sauf s'il y a une bonne raison de modifier la valeur des attributs.

Une méthode peut prendre en argument une autre instance de la classe, comme pour le cas de la méthode `greaterOrEqual` :

```
public boolean greaterOrEqual(IntPair other) {
  if (this.first == other.first) {
    return this.second >= other.second;
  }
  return this.first > other.first;
}
```

L'appel de cette méthode mettra en jeu deux références, comme montré dans l'exemple ci-dessous :

```
IntPair p1 = new IntPair(1, 98);
IntPair p2 = new IntPair(12, 2);
p1.inc(p2);
```

Dans le code de la méthode, la référence `p2` sera désignée par la variable `other` et la référence `p1` par `this`. Cette méthode n'étant pas symétrique, il faut faire attention à la manière dont elle est appelée : l'appel `p1.greaterOrEqual(p2)` ne donnera pas le même résultat que l'appel `p2.greaterOrEqual(p1)` et les deux références ne jouent pas le même rôle dans le code de la fonction. En pratique, une méthode est toujours *attachée* à la référence dont le nom précède le point lors de l'appel (`ref.methode(...)`) et accède aux attributs décrivant cette référence avec la syntaxe `this.attr` ; d'autres références peuvent être passées en paramètres et, dans ce cas, la méthode peut accéder aux attributs décrivant cette référence avec la syntaxe `nomParametre.attr`.

Le listing 17 donne le code complet de la classe `IntPair`.

Différences entre les deux types de classes Il est possible de définir en java deux types de classes différentes :

— des classes comme `IntPair` qui définissent des types ;
— des classes « unité de compilation ».

Le listing 17 illustre les principales différences entre ces deux types de classes : les classes décrivant des types ne contiennent ni méthode `main`, ni méthode `static` mais définissent des attributs et des constructeurs.

```java
public class IntPair {

    private int first;
    private int second;

    public IntPair(int first, int second) {
        this.first = first;
        this.second = second;
    }

    public IntPair(IntPair p) {
        this.first = p.first;
        this.second = p.second;
    }

    public boolean greaterOrEqual(IntPair other) {
        if (this.first == other.first) {
            return this.second >= other.second;
        }
        return this.first > other.first;
    }

    public boolean equal(IntPair other) {
        return this.first == other.first && this.second == other.second;
    }

    public String toString() {
        return "<" + this.first + ", " + this.second + ">";
    }

}
```

Listing 17 – Le code source complet de la classe `IntPair`

Exercice 3.3.1

Écrire une classe `Vector` permettant de manipuler des vecteurs de $\mathbb{R}^3$. Les composantes du vecteur seront représentées par des `double`. L'interface de cette classe comporte :

- un constructeur à trois arguments ;
- un constructeur permettant de créer un vecteur nul ;
- d'une méthode retournant une représentation du vecteur sous forme d'une chaine de caractères de la forme : `<composante nº 1, composante nº 2, composante nº 3>` ;
- d'une méthode fournissant la norme du vecteur ;
- d'une méthode renvoyant *une nouvelle instance* représentant la somme de deux vecteurs
- d'une méthode renvoyant le produit scalaire de deux vecteurs.

Le code suivant permettra de tester le code développé :

```java
public class TestVector {

  public static void main(String[] s) {
    Vector v1 = new Vector(1, 2, 3);
```

```
    Vector v2 = new Vector(3, 4, 5);
    Vector v3 = v1.add(v2);

    System.out.println(v3.toString());
    System.out.println(v1.dot(v2));
    System.out.println(v1.norm());
  }
}
```

3.4 Mode d'accès et encapsulation

Les classes java regroupent dans une même unité syntaxique les variables décrivant un type complexe et les méthodes manipulant celles-ci en interdisant tout accès (en lecture ou en écriture) à ces variables. Cette interdiction est vérifiable par le compilateur qui générera une erreur de compilation dès qu'elle n'est pas respectée. C'est cette interdiction qui va permettre de mettre en œuvre les principes de l'encapsulation.

Ainsi, l'initialisation se faisant toujours par l'intermédiaire d'un constructeur, il est possible de garantir que les objets soient toujours dans un état « cohérent ». Le constructeur de la classe `Etudiant` peut, par exemple, être modifié pour garantir que :

- le nom est toujours en majuscule ;
- le prénom est toujours en minuscule sauf la première lettre ;
- l'adresse mail est bien une adresse mail valable[9].

9 : Par soucis de clarté, une expression régulière simplifiée a été utilisée pour vérifier la validité d'une adresse mail. Une version plus réaliste est décrite ici

Le constructeur modifié sera :

```
public Etudiant(String nom, String prenom, Email email) {
  this.nom = nom.toUpperCase();
  this.prenom = prenom.substring(0, 1).toUpperCase()
              + prenom.substring(1).toLowerCase();
  String pattern = "[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+"
  Matcher m = Pattern.compile(pattern).matcher(s);
  if (!m.matches()) {
    throw new RuntimeException("Invalid Email");
  }
  this.email = email;
}
```

Comme les attributs de la classe sont déclarés en `private`, seules les méthodes de la classe pourront les modifier et il est possible de supposer que ces contraintes sont toujours respectées et d'utiliser celles-ci pour simplifier l'implémentation de la classe.

L'interdiction de la modification des attributs limite également la propagation des modifications. Si les spécifications de la classe `IntPair` sont modifiées et qu'il faut désormais supprimer la comparaison entre paires et considérer que les paires $\langle 1, 2\rangle$ et $\langle 2, 1\rangle$ sont égales (c.-à-d. ne plus tenir compte de l'ordre des éléments dans le test d'égalité des paires), il suffit de :

— supprimer la méthode `greaterOrEqual` de la définition de la classe ;
— modifier la méthode `equals` de la manière suivante :

```
    public boolean equals(Pair p) {
      return (this.first == p.first && this.second == p.second) || \
             (this.first == p.second && this.second == p.first);
    }
```

La première modification aura un impact sur l'ensemble du code puisqu'elle modifie l'interface de la classe. La seconde sera par contre transparente pour les utilisateurs·rices de la classe. Mieux encore : dans la mesure où il n'est pas possible d'accéder directement aux attributs pour comparer leurs valeurs, la seule manière pour un·e programmeur·euse de tester l'égalité de deux paires est d'utiliser la méthode `equals`. L'encapsulation permet donc de garantir que l'ensemble du code du programme utilise bien la nouvelle définition de l'égalité.

L'encapsulation permet également de simplifier l'implémentation de la nouvelle définition de l'égalité : dans la mesure où les attributs ne sont accessibles que dans la définition de la classe, le·a programmeur·euse est libre de choisir de la définition des attributs la plus adaptée aux besoins du code. Il ou elle peut par exemple choisir de toujours stocker la plus petite valeur dans l'attribut `first` en modifiant le constructeur de la manière suivante :

```
public IntPair(int x, int y) {
  if (x < y) {
    this.first = x;
    this.second = y;
  } else {
    this.first = y;
    this.second = x;
  }
}
```

Comme les attributs ne peuvent plus être modifiés (ou alors uniquement par des méthodes de la classe pour lesquels il est facile de vérifier que la contrainte d'ordre est respectée), il est possible d'utiliser cette nouvelle contrainte pour simplifier l'implémentation de la méthode `equals` :

```
public boolean equals(Pair p) {
  return this.first == p.first && this.second == p.second;
}
```

Si l'intérêt de la simplification est faible dans le cas des paires, il devient non négligeable dans le cas des triplets puisqu'il n'y aura pas besoin de tester les 6 permutations possibles de celui-ci.

3.5 POO & Java

3.6 Interfaces

Motivation Un langage orienté objet permet au programmeur ou à la programmeuse de définir facilement de nouveau type pour augmenter l'expressivité du langage tout en limitant les problèmes de propagation des modifications et ceux liés à la cohérences des données. La définition de nouveaux types peut toutefois soulevée des difficultés lorsque deux entités « proches » sont représentées par des types distincts dans le programme.

Ainsi deux modélisations d'une phrase sont possibles. Une première modélisation ne met en jeu qu'une seule classe `Word` pouvant posséder, par exemple, une méthode `getPronunciation` qui renvoie la prononciation du mot (en alphabet phonétique internationale), une méthode `corrected` renvoyant la forme corrigé du token (cette méthode renverra, par exemple, `hippopotame` si le mot est `hipopotame`) et une méthode `toPlural` renvoyant le pluriel du mots. Dans une seconde modélisation, chaque catégorie morpho-syntaxique est modélisée par une classe et il y a une classe `Noun`, une classe `Verb`, une classe `Preposition`, ...

Si la seconde méthode peu sembler plus complexe de premier abord, distinguer une classe par catégorie morpho-syntaxique permet de définir des méthodes spécifiques à chaque catégorie morpho-syntaxique : il est, par exemple, possible de définir une méthode `isPresent` dans la classe `Verb` qui n'aurait aucun sens pour un nom ou une préposition. Ajouter cette méthode à la classe `Word` nécessite de gérer tous les cas où la méthode est appelée dans un contexte dans lequel elle n'a pas de sens (c.-à-d. lorsque le mot représenté n'est pas un verbe). La distinction des classes permet donc, *in fine*, d'obtenir une modélisation plus fine et, par conséquent, un code plus simple, mais soulève, en contrepartie, plusieurs problèmes : même si les instances de `Verb` et les instances de `Noun` représentent toutes les deux des mots et ont donc certains comportement communs (ces instances peuvent, par exemple, être combinées pour former des phrases), elles seront de types différents et ne pourront donc pas toujours être utilisées dans le même contexte.

Ainsi il ne sera pas possible de définir d'ajouter une instance `Noun` et une instance de `Verb` à une même liste et il faudra définir une méthode pour chaque catégorie morpho-syntaxique même si le corps de la méthode est exactement le même, une méthode ayant un paramètre de type `Noun` ne pouvant être appelée avec une instance de `Verb` en paramètre. Pour résoudre ce type de problème, la programmation orientée objet définit la notion d'interface[10].

10 : Suivant les langages, la notion introduite dans cette section est appelée soit interface, soit classe virtuelle pure

Élément de syntaxe Une interface permet de définir un nouveau type et de formaliser la notion, introduite dans la section 3.2, de contrat entre le·la programmeur·euse de la classe et la personne utilisant celle-ci en listant l'ensemble des opérations qu'il est possible d'effectuer sur ce type sans préciser la dont celles-ci fonctionnent : en première

```java
public interface Word {

    // retourne une chaine de caractères décrivant
    // la prononciation du mot (en IPA)
    public String getPronunciation();
    // retourne le mot orthographié correctement
    public String corrected();

}
```

Listing 18 – Interface décrivant le comportement d'un mot.

approximation, une interface est simplement une énumération de signature de méthode. Le listing 18 donne un exemple d'une interface décrivant les opérations qu'il est possible d'effectuer sur un mot (cf. la description donnée dans le paragraphe précédent).

Il n'est pas possible de créer directement une instance dont le type est défini par une interface : une instruction du type :

```java
  Word w = new Word();
```

entrainera une erreur de compilation :

```
> javac Test.java
Test.java:4: error: Word is abstract; cannot be instantiated
        Word w = new Word();
                 ^
1 error
```

En effet, l'interface ne spécifiant que le contrat (ce que fait le type) et non pas l'implémentation[11] (comment il le fait), la machine virtuelle ne saura pas quelles instructions exécutés pour réaliser une opération donnée.

11 : Une interface ne comporte ni attribut, ni constructeur ni aucune instruction « exécutable ».

Cette interface doit ensuite entre implémenter par une (ou plusieurs classe). Ces classes sont qualifiées de *classes concrètes* par opposition aux interfaces pour indiquer qu'elles comportent l'implémentation des méthodes et pas uniquement leur déclaration. Comme le montre le listing 19, une classe doit faire deux choses pour implémenter une interface :

- signifier au compilateur qu'elle implémente l'interface en ajoutant à sa définition le mot-clé `implements` suivit du nom de l'interface implémentée ;
- définir l'ensemble des méthodes (c.-à-d. donner leur code) définies dans l'interface. Ces méthodes doivent être précédées de l'annotation `@Override`.

Une classe qui implémente l'interface doit définir l'ensemble des méthodes en respectant exactement la signature de celles-ci (même nombre et type de paramètres). Si ce n'est pas le cas, le compilateur générera une erreur de compilation, comme celle montrée dans le figure 3.2.

Il est possible de définir tous les attributs, constructeurs et autres méthodes nécessaires à la réalisation de la classe (l'interface n'impose une

```
> javac Verb.java
Verb.java:1: error: Verb is not abstract and does not override
abstract method corrected() in Word
public class Verb implements Word {
       ^
1 error
```

Figure 3.2 – Erreur générée lorsqu'une des méthodes de l'interface n'est pas définie (ici la méthode `corrected`.

Listing 19 – Une implémentation de l'interface `Word`.

```
1  public class Verb implements Word {
2
3      private String token;
4
5      public Verb(String token) {
6          this.token = token;
7      }
8
9      @Override
10     public String getPronunciation() {
11         // ...
12         return "";
13     }
14
15     @Override
16     public String corrected() {
17         // ...
18         return "";
19     }
20
21     public boolean isFuture() {
22         return true;
23     }
24
25 }
```

contrainte que sur les méthodes dont la signature est définie dans l'interface). Ainsi, dans le listing 19, une méthode spécifique aux verbes ont été ajoutée.

Une instance d'une classe implémentant une interface a deux types : le type de la classe et le type de l'interface. Les deux instructions suivantes seront donc parfaitement valables :

```
1 Verb va = new Verb();
2 Word v2 = new Verb();
```

Ainsi, suivant le contexte, une instance de la classe `Verb` pourra être utilisée comme argument d'une fonction prenant un paramètre de type `Verb` mais également d'une fonction prenant un argument de type `Word`; de la même manière une liste de type `ArrayList<Word>` pourra stocker aussi bien des instances de `Verb` que des instances de `Noun` (si cette dernière classe implémente bien l'interface). Il faut toutefois faire attention : lorsqu'une variable est déclarée avoir pour type le type d'une interface, seules les opérations définies par cette dernière pourront lui être appliquée. Ainsi, la fonction suivante qui cherche

à vérifier si une phrase (représentée par une liste de `Word`) est au futur :

```
1  public boolean isSentenceFuture(ArrayList<Word> sentence) {
2    for (Word w : sentence) {
3      if (w.isFuture()) {
4        return true;
5      }
6    }
7    return false;
8  }
```

entraînera une erreur de compilation :

```
> javac Test.java
Test.java:7: error: cannot find symbol
            if (w.isFuture()) {
                ^
  symbol:   method isFuture()
  location: variable w of type Word
1 error
```

En effet, dans le contexte de la fonction `isSentenceFuture`, les références stockées dans la liste seront considérées uniquement comme des variables de type `Word` quelque soit la classe concrète utilisée pour les instancier. Seules les méthodes de l'interface pourront être utilisées. Il est toutefois possible de *convertir (cast)* la référence pour pouvoir utiliser les méthodes spécifiques à la classe concrète. Ainsi, la version correcte de la fonction `isSentenceFuture` est :

Le terme correct en français est la *coercition de type* ou le *transtypage*.

```
1  public boolean isSentenceFuture(ArrayList<Word> sentence) {
2    for (Word w : sentence) {
3
4      if (!(w instanceof Verb)) {
5        continue;
6      }
7
8      Verb v = (Verb) w; //
9      if (v.isFuture()) {
10       return true;
11     }
12   }
13   return false;
14 }
```

Le transtypage à proprement parlé a lieu à la ligne 8. Celui-ci n'est possible que si l'instance décrite représente bien un verbe c'est-à-dire si c'est bien une instance de la classe `Verb`. Dans le cas contraire, la machine virtuelle lèvera une exception `ClassCastException`. Il est possible de vérifier que cette condition à l'aide du mot-clé `instanceof` et d'effectuer le transtypage uniquement lorsque celui-ci est possible. Une fois la conversion effectuée et une variable de type `Verb` définie, il est possible d'utiliser les méthodes spécifiques à cette classe sur celle-ci.

Interfaces de la bibliothèque standard Java La bibliothèque standard java définit plusieurs interfaces permettant d'intégrer les classes définies par un·e utilisateur·rice avec des constructions du langage (p. ex. les aboucles de type `foreach`) ou certaines fonctionnalités de la bibliothèque standard.

Par exemple, l'interface `Serializable` permet d'indiquer que les instances d'une classe peuvent être sauvegardées dans un fichier. Cette interface ne définit aucune méthode et a pour seul objectif d'indiquer à la machine virtuelle que les opérations nécessaires à la sauvegarde d'une instance sont possibles. Un exemple d'utilisation de cette interface est donné dans le listing 20 : si la classe `Person` n'était pas définie comme implémentant l'interface `Serializable`, la machine virtuelle lèverait une exception `NotSerializableException` lors de l'exécution du programme d'exemple.

La *sérialisation* est l'opération qui consiste à transformer une instance en une séquence d'octets. Celle-ci peut ensuite être stockée dans un fichier, une base de données ou être envoyée à travers un réseau. Elle est ensuite *désérialisée* pour « recréer » l'objet.

```java
import java.io.Serializable;

public class Person implements Serializable {

    String firstName;
    String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public boolean isSame(Person other) {
        return this.firstName.equals(other.firstName) &&\
            this.lastName.equals(this.lastName);
    }

}
```

```java
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

public class ExSerialization {

    public static void main(String[] args)\
        throws IOException, ClassNotFoundException {

        Person p1 = new Person("Chimamanda",
                               "Ngozi Adichie");
        Person p2 = new Person("Patsauq",
                               "Markoosie");

        ObjectOutputStream ostream = \
            new ObjectOutputStream(new FileOutputStream("ex.pkl"));
        ostream.writeObject(p1);

        ObjectInputStream istream = \
            new ObjectInputStream(new FileInputStream("ex.pkl"));
        Person p3 = (Person) istream.readObject();

        System.out.println(p1.isSame(p3));
        System.out.println(p2.isSame(p3));
    }
}
```

Listing 20 – Définition d'une classe pouvant être sérialisée et sérialisation de celle-ci.

Java avancé 4

4.1 Utilisation de bibliothèques

L'écosystème java contient de très nombreuses bibliothèques qui couvrent la plupart des sujets imaginables. Leur utilisation permet de réaliser très facilement des logiciels complets. Il existe ainsi des bibliothèques pour :

- ajouter des fonctionnalités manquante à la bibliothèque standard java (projet Apache Commons) ;
- construire des images 2D ou 3D (projet Java OpenGL) ;
- réaliser des calculs statistiques ou mathématiques (projet International Mathematics and Statistics Library) ;
- faire du TAL (projet CoreNLP).

Contrairement au classes et méthodes de la bibliothèque standard, ces bibliothèques ne sont pas diffusées avec le compilateur et la machine virtuelle standard. Elles doivent être explicitement installées par le programmeur.

Java définit un format, le format `jar`, pour stocker et diffuser des bibliothèques. L'utilisation des bibliothèques dans `eclipse` est particulièrement simple : il suffit de télécharger le `jar` de la bibliothèque à partir du site web de celle-ci ; ajouter ce fichier au projet dans `eclipse` (par exemple en faisant un glisser-déplacer du fichier à la racine du projet) puis indiquer au compilateur et à la machine virtuelle qu'il faut qu'ils considèrent les classes et les fonctions définies dans celui-ci en ajoutant le jar à la liste des bibliothèques à considérer (menu `Project > Properties > Java Build Path > Libraries > Add JAR`).

Le fichier jar peut être contenu dans une archive. Dans ce cas, il faut commencer par extraire celui-ci de l'archive.

Cette approche extrêmement simple souffre ne passe toutefois pas à l'échelle : si elle peut être utilisée lorsqu'un projet n'utilise qu'une ou deux libraires, il est inconcevable de demander à un programmeur de télécharger et d'installer manuellement une dizaine de bibliothèque. L'installation manuelle de bibliothèque soulève deux autres problèmes :

- une bibliothèque peut dépendre[1] et il est vite fastidieux de devoir lister et installer toutes les dépendances ;
- le code d'une bibliothèque peut évoluer très rapidement et le programme nécessiter une version précise de la bibliothèque. En plus de lister toutes les bibliothèques dont un programme dépend, il faut s'assurer de connaitre la version exacte de la bibliothèque dont il a besoin (et de pouvoir installer celle-ci).

1 : Une bibliothèque A *dépend* d'une bibliothèque B, si l'exécution de A n'est possible que si la bibliothèque B est installée

Il existe aujourd'hui des outils, comme Apache Maven qui automatise la gestion et l'installation des dépendances. La prise en main d'un tel outils dépasse toutefois les objectifs de ce document.

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 21.

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 21 – 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 :

```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 += "*";
    }
  }
```

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

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

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 22 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*.

Le code final (listing 23) 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

```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);
    }

}
```

Listing 22 – Code java pour l'exercice 1.4.1 avant refactoring.

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 23 – 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  ^^I^^I
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

```
1  import java.util.ArrayList;
2  import java.util.HashSet;
3
4  public class RepeatedArrayList {
5
6      public static boolean allUnique(ArrayList<String> lst) {
7          HashSet<String> set = new HashSet<String>(lst);
8          return lst.size() == set.size();
9      }
10
11     public static void main(String[] s) {
12
13         ArrayList<String> ex1 = new ArrayList<>();
14         ex1.add("a");
15         ex1.add("b");
16         ex1.add("a");
17
18         ArrayList<String> ex2 = new ArrayList<>();
19         ex2.add("a");
20         ex2.add("b");
21         ex2.add("c");
22
23         System.out.println("ex1 : " + allUnique(ex1));
24         System.out.println("ex2 : " + allUnique(ex2));
25     }
26 }
```

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;
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);
    }

}
```