
Algorithmique
&
Programmation Orientée Objet

L3 MIAGE

PL

Université Grenoble Alpes
Celine.Fouard@univ-grenoble-alpes.fr

Copyright ©2010–2025 Céline Fouard, PhD

Ce cours a été rédigé par Céline Fouard.

Il est très largement inspiré du cours d'Emmanuel Promayon, PhD donné à Polytech'Grenoble
Les derniers chapitres ont été rédigés par Gilles Serasset

Il est également inspiré du polycopié de cours de programmation orientée objet d'Yvan Maillot.

Les chapitres 13, 21 et 7 sont repris de Fiches Ensimag 2A- Programmation Orientée Objet, avec l'aimable
autorisation de Sylvain Bouveret, Nicolas Castagne, Matthieu Chabanas, Catherine Oriat and Sébastien
Viardot pour l'Ensimag.

Si vous souhaitez utiliser ce document, merci de contacter Celine.Fouard@univ-grenoble-alpes.fr

Contents

A Bases de la Programmation en Java	1
1 Introduction	3
2 Présentation de Java	5
1 Le langage Java	5
2 Exécution d'un programme Java	8
3 Classe et Instance	13
1 Introduction	13
2 Exemple détaillé	14
3 Du statique au dynamique: exécution d'un programme Java	15
4 Exemple d'exécution et d'instanciation	16
5 Exécution pas à pas	17
4 Le diagramme UML de Classes	23
1 Pourquoi dessiner des diagrammes de classes UML ?	23
2 Les éléments de base du diagramme de classes UML	24
3 Les relations entre les classes	26
4 Autres éléments	29
5 Déroulement d'un programme: Le diagramme APO	31
1 Pourquoi dessiner des diagrammes APO ?	31
2 Catégories et familles de données	32
3 Squelette d'un diagramme APO	35
4 Les éléments d'un diagramme APO	35
6 Types prédéfinis en Java	39
1 Introduction	39
2 Booléen	40
3 Entier	41
4 Réels	43
5 Caractères	44
6 Chaîne de caractères	44
7 Conversions automatiques entre types primitifs	47
8 Autres opérateurs	48
7 Les Exceptions en Java	49
1 Exceptions Intégrées	50
2 La gestion des Exceptions	51
3 Créer ses propres Exceptions	54

B Rappel des Structures Algorithmiques	57
8 Instructions conditionnelles	59
1 Introduction	59
2 Principe algorithmique	60
3 Instruction conditionnelle à deux branches	61
4 Else if	63
5 Instruction à choix multiple	64
9 Constantes et énumérations	67
1 Les constantes	67
2 Énumérations	68
10 tableaux de données	71
1 Introduction	72
2 Déclaration et instanciation	72
3 Accès à une case du tableau	74
4 Remarques sur les tableaux	76
5 Tableau à n dimensions	78
11 Structure algorithmique de parcours	81
1 Introduction	81
2 Déroulement	82
3 En Java	84
12 Structure algorithmique de recherche	87
1 Introduction	87
2 Déroulement	88
3 En Java	89
4 Boucle Faire	90
5 Parcours ou recherche ?	90
C Programmation Orientée Objet en Java	93
13 Les paquetages en Java	95
1 Les paquetages (<i>packages</i>)	95
2 Paquetages et compilation / exécution	97
3 Archives JAR	99
14 Java: La classe !	101
1 Le principe d'encapsulation	101
2 Les modificateurs d'accès	103
3 La classe	103
4 Les attributs	104
5 Les méthodes	105
6 Le modificateur final	108
7 Les objets immuables	109
15 Les classes Numériques	115
1 Introduction	115
2 La famille Number	116
3 Conversion automatique	116
16 Vide de famille: Héritage et Polymorphisme	119
1 les principes de l'héritage	119
2 Héritage en Java	121
3 Les 11 points clés de l'héritage	125
4 Transtypage	126

CONTENTS	v
17 La classe Object	129
1 Introduction	129
2 Quelques méthodes de la classe Object	130
18 Classes Abstraites	135
1 Introduction	135
2 Classe abstraite en UML	137
3 Classe et méthode abstraite en Java	137
4 Exemple	138
19 Interfaces	139
1 Définition	139
2 Exemple : l'interface Comparable	140
3 Exemple d'utilisation de l'interface Comparable	141
4 Interface et polymorphisme	142
5 Hiérarchie d'interfaces et généricité	142
20 Le principe de la généricité	143
1 Principe	143
2 Intérêt	144
3 Généricité et héritage	147
21 API Collection	149
1 Principe	149
2 Les séquences: List	151
3 Files et piles: Queue, Deque	152
4 Files à priorités	153
5 Ensembles: Set	156
6 Dictionnaires: Map	158
22 Méthodes et variables statiques	161
1 Exemple: la classe Math	161
2 Méthodes statiques	161
3 Variables statiques	163
D Ordre supérieur en Java	167
23 Fonctions: l'ordre supérieur en java	169
1 Principe	169
2 Intérêt	170
3 Ordre supérieur en java avant java 8	170
4 Ordre supérieur en java depuis java 8	171
24 API des Streams: l'ordre supérieur et les collections Java	179
1 Introduction	179
2 Un exemple préliminaire commenté : l'abréviation majuscule	179
3 Les Streams en Java	186
4 Création d'un Stream	187
5 Opérations intermédiaires	188
6 Opérations terminales	190

Part A

Bases de la Programmation en Java

Introduction

Définition



Un **algorithme** est une *suite d'instructions à faire exécuter* qui décrit la marche à suivre pour réaliser un calcul ou une transformation de manière systématique et déterministe.

Un algorithme doit spécifier :

- les données en entrée
- les données produites (les données en sortie)
- les prédicts : les propositions logiques qui décrivent l'état de la mémoire (plus généralement de la machine) avant et après l'exécution de l'algorithme.
- la suite d'instructions algorithmiques (c'est-à-dire utilisant un vocabulaire avec un nombre limité de mot). Cette suite d'instructions décrit l'enchaînement des traitements (séquentiel, boucle,...) en utilisant un langage spécifique pour décrire ces traitements élémentaires.

Note



Nous utiliserons le langage de programmation Java, mais tous les algorithmes que nous (re)verrons pourront s'appliquer à *tous les langages de programmation contenant le paradigme objet*, comme C++, python, javascript, etc.

La difficulté d'un algorithme réside dans le passage du faire à faire faire. En l'occurrence, celui à qui l'on fait faire est dénué de toute connaissance a priori : il s'agit de l'ordinateur et plus particulièrement dans notre cas du langage Java. On peut donc lui parler avec un ensemble fini et restreint de mots.



La compétence visée par cette partie du cours est la maîtrise des structures de base de **l'algorithmique**. L'objectif est de comprendre et de savoir exprimer en termes algorithmiques les traitements à faire faire à la machine.

L'objectif n'est pas de faire uniquement de la programmation d'un traitement, mais de savoir exprimer ce traitement dans les *bons termes*.

On s'intéresse donc aux **principes** de l'algorithmique et il s'agit d'exprimer correctement le traitement dans un langage de programmation (ici le langage Java), et il ne **s'agit pas seulement** de faire un programme qui marche, mais d'adopter une démarche rigoureuse pour faire un programme qui non seulement donne le résultat attendu, mais offre également des qualités de robustesse, lisibilité et efficacité.

Tous les exercices, TD, TP et examens sont des traductions dans le langage Java de concept de modélisation et d'organisation des traitements dans le langage Java.

Présentation de Java

Fiche

<u>Prérequis</u>	<ul style="list-style-type: none"> Avoir déjà compilé et exécuté un programme
<u>Objectif pédagogique</u>	<ul style="list-style-type: none"> Connaître l'historique et les évolutions du langage Java Connaître les conditions de compilation et d'exécution d'un programme Java
<u>Intérêt</u>	<ul style="list-style-type: none"> Comprendre l'intérêt de l'étude du langage Java pour la Programmation Orientée Objet Connaître les espaces mémoires en jeu dans l'exécution d'un programme Java
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> Connaître les étapes de l'écriture à l'exécution d'un programme Java Connaître les environnements de développement et d'exécution d'un programme Java Comprendre les bases du fonctionnement de la JVM

1 Le langage Java

1.a Historique et évolutions

Le langage Java a été développé dans les années 1990 par l'entreprise Sun Microsystems (https://fr.wikipedia.org/wiki/Sun_Microsystems). Après un départ chaotique et quelques déboires, il s'impose de nous jours comme un des langages orientés objets phares, notamment en programmation web mais aussi comme langage généraliste.

Java est destiné initialement à être implanté dans des appareils *intelligents*, il a été prévu pour être simple et indépendant de l'architecture. Il est exploité ensuite pour développer un navigateur capable d'exécuter du code sur n'importe quelle machine, indépendamment de l'architecture, sécurisé et fiable. Il devient finalement un langage généraliste ([https://fr.wikipedia.org/wiki/Java_\(langage\)](https://fr.wikipedia.org/wiki/Java_(langage))).

Les différentes versions de java peuvent être trouvées sur le site https://en.wikipedia.org/wiki/Java_version_history et sont résumée dans la figure 2.1.

On peut noter les évolutions du langage au cours des années:

Version	Release date	End of Free Public Updates ^{[8][9]}	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	January 2019 for Oracle (commercial) December 2020 for Oracle (personal use) At least September 2023 for AdoptOpenJDK	March 2025
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	September 2022 for AdoptOpenJDK	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA

Legend: Old version Older version, still supported Latest version Latest preview version Future release

Figure 2.1: Différentes versions de Java au cours du temps

JDK 1.0 aussi appelé Java 1, sorti en mai 95. Cette version comprend 8 *packages*. Aujourd’hui, Java SE8 en comporte plus de 400.

JDK 1.1 sorti en mars 97 ajoute de nombreuses fonctionnalités:

- les classes internes (que nous verrons en fin de semestre)
- la sérialisation (que vous verrez dans une autre UE)
- l’internationalisation
- l’introspection (utilisée pour évaluer vos labs Caseine)
- l’accès aux bases de données avec JDBC
- etc.

JDK 1.2 sortie en décembre 98

- apparition des Java Foundation Classes (JFC) avec Swing, Java2D, drag & drop...
- le framework collections (que nous verrons plus loin)

JDK 1.2 et JDK 1.4 améliorent la rapidité et ajoutent des fonctionnalités, notamment pour la gestion de XML

J2SE 5.0 sortie en septembre 2004 apporte d'énormes améliorations dans le but de simplifier l'écriture du code:

- la généricité (que nous verrons plus loin)
- les grands nombres
- les annotations
- les énumérations (dont vous avez vu l'intérêt au premier semestre)
- la boucle `for each` (qui doit devenir la structure de parcours par défaut pour les tableaux)
- une simplification dans la synchronisation des threads (le multi-threading ne sera pas vu dans ce cours)
- etc.

Java SE 7 sortie en juillet 2011

- possibilité d'avoir des `String` dans les `switch`
- etc.

Java SE8, SE9, SE10 ajoutent notamment les lambda-expressions, ainsi qu'une interface fonctionnelle que nous n'utiliserons pas dans ce cours (pour se concentrer sur l'aspect objet)

1.b Caractéristiques de Java

Les caractéristiques de Java font de lui un langage facile à apprêhender et sûr.

Orienté objet En effet, Java fait intervenir peu de paradigmes (ce qui évite de tous les mélanger) et est totalement orienté objet:

- Tout est objet
- Tout est basé sur les classes

Fortement typé ce qui le rend très lisible et qui oblige à tout spécifier

- Tout objet a un type bien défini
- Le typage est statique mais il existe des conversions de type dynamique

Compilé Le compilateur `javac` compile le code source en un *ByteCode* (à partir de fichiers `.java` on obtient des fichiers `.class`)

Interprété La machine virtuelle Java (JVM) interprète le *ByteCode*

Portable

- Java est indépendant de la machine

- Java est indépendant du système d'exploitation (pourvu qu'il y ait une JVM)
- l'une des maximes de Java est *Write once, run anywhere*

Sécurisé

- Sécurisation du développement, en effet, Java interdit beaucoup de comportement de développement dangereux

- Sécurisation de l'exécution avec 2 niveaux de sécurité (que nous verrons plus loin)

Simple La syntaxe est inspirée de C/C++, mais en interdisant les principales erreurs de ce langage très permissif:

- pas d'arithmétique de pointeurs, mais une gestion simple des références
- pas de débordement possible
- gestion des instances créées grâce à un *garbage collector*, pas besoin de libérer la mémoire (`delete` source de beaucoup de *crashes* en C/C++)
- pas d'héritage multiple, souvent signe d'une mauvaise conception ou conception trop complexe
- les tableaux, les énumérations, les `String` sont des objets
- tout objet hérite d'une seule et même classe commune: la classe `Object`

2 Exécution d'un programme Java

2.a Au début était le `main`...

Pour effectuer des traitements, il faut un point d'entrée c'est-à-dire un endroit qui contient la première instruction à exécuter. En Java, on l'appelle le programme **principal**, qui donne en anglais `main`. En Java, pour dérouler un programme, on va passer par une classe à part (souvent appelée `Test`) qui contiendra une méthode particulière: la méthode `main`.

Définition



Le `main` est une action spécifique (une *méthode* en Java) qui contient la première instruction à exécuter quand on lance un programme (exécute le programme / l'application).

C'est une opération spécifique et **unique** pour un programme / une application.



Attention, Java étant un langage **pur objet**, toutes les méthodes sont dans des classes, y compris la méthode `main`. Il faut donc créer une classe spéciale pour le programme principal.

Le `main` permet de formuler *la première instruction*, c'est-à-dire la première ligne de code à exécuter. Une fois la première instruction exécutée, on peut également avoir d'autres instructions. Les instructions suivantes sont écrites à la suite, dans les lignes suivantes de l'opération `main`.

La **signature** de la méthode `main` est toujours la même: `public static void main(String[] args)`. La classe qui contient le `main`, elle peut avoir le nom souhaité par le programmeur.

Par exemple, on pourra avoir, dans le fichier `Test.java`, le code suivant:

```

1 class Test {
2     public static void main(String[] args) {
3         // Ici, les instructions du programme principal...
4     }
5 }
```

Evidemment, un programme `main` peut appeler différentes méthodes de différentes classes, chacune dans un fichier `.java` portant le nom de la classe. On se retrouve donc avec un certain nombre de fichiers `.java` contenant des classes.

2.b Java est un langage compilé

Dans le cas d'un langage compilé classique, le modèle d'exécution consiste à :

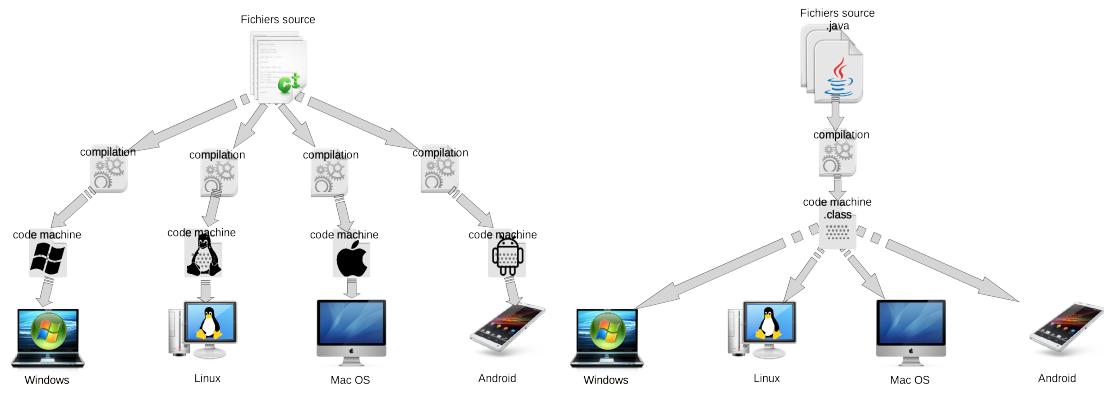
- écrire le code source d'un programme dans un langage (par exemple C++)
- le compiler sur une machine et son système d'exploitation (par exemple, PC sous Windows, un Mac sous OSX, un PC sous Linux, une machine Silicon sous Unix, un smartphone sous Android, etc.)
- l'exécuter sur la machine sur laquelle on l'a compilé, comme illustré figure 2.2(a)

Ce principe offre une certaine portabilité. Le programme source est écrit une fois pour toute. Il reste à le compiler sur la machine voulue.

Le modèle d'exécution choisi par Java est encore plus portable que le modèle classique: il consiste à écrire le code source d'un programme et à le compiler. La compilation génère un fichier *bytecode* exécutable sur n'importe quelle machine et n'importe quel système pourvu que ce système soit muni d'une *machine virtuelle java (JVM)*, comme illustré figure 2.2(b).

Ce principe offre une grande portabilité et permet notamment de transporter à travers le web du code (lourd) exécutable sur n'importe quelle machine munie d'une JVM.

Pour compiler un programme Java, on utilise l'utilitaire `javac` disponible dans le JDK (*Java Development Kit* en anglais) Java, c'est-à-dire l'ensemble des outils qui permettent de développer en Java.



(a) Dans le cas des langages uniquement compilés, la compilation doit être faite sur la même machine que l'exécution.

(b) Dans le cas de Java, le programme une fois compilé peut s'exécuter sur n'importe quelle machine munie d'une JVM.

Figure 2.2:

2.c Java est un langage interprété...

...Par la machine virtuelle java (JVM)

En effet, une fois le programme lancé par l'exécution de la ligne de commande `java NomDeLaClasseQuiContientLaMethodeMain` le main est exécuté et on commence à manipuler des informations dans la mémoire vive, aussi appelée RAM (pour *Random Access Memory* en anglais: mémoire à accès aléatoire).

En java, la mémoire utilisée pour l'exécution des programmes est séparée de la mémoire principale par l'utilisation de la machine virtuelle Java. La JVM (*Java Virtual Machine* en anglais) est un environnement de programmation pour les applications Java¹.

Il existe deux types de machines virtuelles, les VM système et les VM applicatives. La JVM fait partie de la seconde catégorie. Elle est exécutée comme n'importe quelle application sur un système d'exploitation hôte [...]. Son but est de fournir un environnement de programmation indépendant de la plate-forme qui fait abstraction du matériel sous-jacent et/ou du système d'exploitation, et permet à un programme de s'exécuter de la même manière sur n'importe quelle plate-forme. La JVM utilise un bytecode, un langage intermédiaire entre Java (le langage utilisateur) et le langage machine².

La JVM va charger et *interpréter* les fichiers `.class` pour exécuter le programme. Elle va donc charger toutes les classes définies dans les fichiers `.class` et rechercher celle qui contient la méthode `main`. La JVM, qui interprète les fichiers `.class` fait partie d'un ensemble d'outils qui permet d'exécuter des programmes Java: le JRE (*Java Runtime Environment*) qu'il faut avoir installer sur son système ou son navigateur pour pouvoir exécuter des programmes Java. Ces étapes sont résumées sur la figure 2.3³

Les espaces mémoire de la JVM

Comme illustré dans la figure 2.4, la JVM définit différentes zones de données d'exécution qui sont utilisées durant l'exécution d'un programme. Certaines de ces zones de données sont créées lorsque la JVM est lancée et sont détruites lorsqu'elle s'arrête. Les autres zones sont propres à chaque fil d'exécution (thread). Les zones de données par fils d'exécution sont créées lorsque le fil d'exécution est créé et sont détruites lorsqu'il se termine⁴.

Fils d'exécution

Une méthode Java peut lancer dans la JVM plusieurs fils d'exécutions en parallèle. Il s'agit alors de programmation parallèle (*multi-thread*) que nous n'aborderons pas dans ce cours. Nous nous intéresserons uniquement au fil d'exécution principal d'un application, c'est-à-dire celui de la méthode `main`.

¹extrait de <http://www.jmdoudoux.fr/java/dej/chap-jvm.htm>

²extrait de <https://soat.developpez.com/tutoriels/java/jvm/decouverte-machine-virtuelle-java/?page=intro-jvm>

³figure extraite de http://www.ntu.edu.sg/home/ehchua/programming/java/J2_Basics.html

⁴extrait de <https://soat.developpez.com/tutoriels/java/jvm/decouverte-machine-virtuelle-java/?page=intro-jvm>

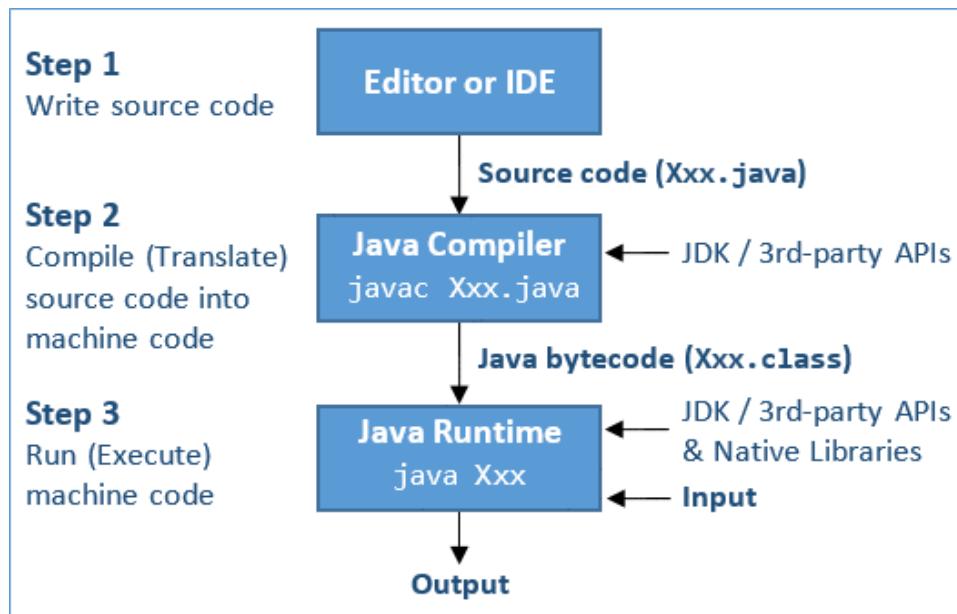


Figure 2.3: Différentes étapes d'un programme Java

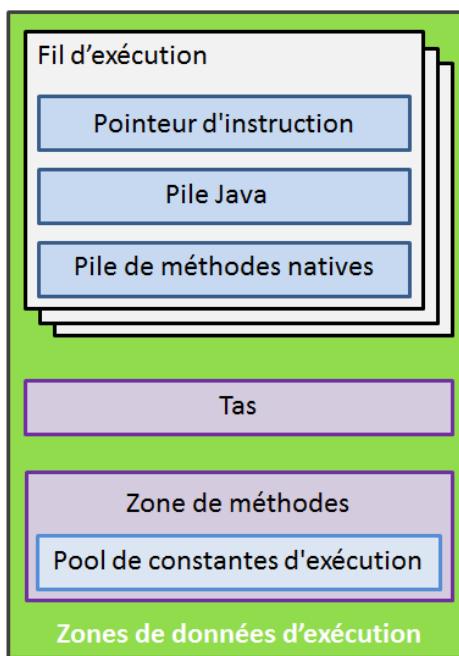


Figure 2.4: Illustration des types de champ mémoire de la JVM.

Pour chaque fil d'exécution, est stocké dans la mémoire:

- un pointeur d'instruction qui référence des instructions dans la Pile Java,
- Une pile Java, dans laquelle se trouvent les données locales aux méthodes courantes (variables, paramètres, etc.)
- Une Pile de méthodes natives: à l'instar des méthodes Java qui sont exécutées dans les piles Java, les méthodes natives (généralement en C/C++) sont exécutées dans les piles de méthodes natives. Les piles de méthodes natives sont généralement allouées par fil d'exécution lorsqu'il est créé.

Tas

La JVM a un tas partagé par tous les fils d'exécution. Le tas est une zone de données d'exécution dans laquelle toutes les instances de classes et les tableaux sont stockés. Le tas est créé au démarrage de la JVM. Les objets n'étant pas explicitement désalloués de la mémoire, lorsqu'ils ne sont plus référencés le ramasse-miettes les supprime du tas.

Zone des méthodes

Lorsque le chargeur de classes charge une classe, il stocke sa structure (le pool de constantes, les données des champs et des méthodes, et le code des méthodes et des constructeurs) dans la Zone de Méthodes. Il s'agit d'un espace mémoire partagé par tous les fils d'exécution et la JVM n'impose pas qu'il soit libéré par le ramasse-miettes et donc qu'il fasse partie du tas (heap).

Classe et Instance

Fiche

<u>Prérequis</u>	Algorithmique impérative de base
<u>Objectif pédagogique</u>	<p>Introduire par l'exemple les notions qui permettront de représenter les différentes étapes de programmation de la conception au débogage :</p> <ul style="list-style-type: none"> les notions de base de classe (en Java et en diagramme UML de classes) l'instanciation et la définition d'une instance et d'une référence (en Java et en diagramme APO)
<u>Intérêt</u>	Partir sur les bonnes bases pour approfondir l'algorithmique en Java.
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> Comprendre les définitions et différences entre classes et instances Savoir représenter une classe en diagramme UML Savoir écrire une classe simple en Java Savoir représenter des variables et une instance en diagramme APO.

1 Introduction

1.a Algorithmique en Java

de L'objectif des parties **A** et **B** est de consolider et d'approfondir les bases de l'algorithmique en Java. Elle devrait vous permettre d'une part de maîtriser les structures algorithmiques de base et leurs cas d'utilisation et d'autre part d'être capable de mettre en œuvre une stratégie efficace de résolution de problèmes complexes par la décomposition en problèmes simples et dont la résolution est connue.

La mise en œuvre des algorithmes vus se fera en Java. Lors de ces premières parties, les spécificités d'un langage orienté objet ne seront pas abordées puisque vues et approfondies au deuxième semestre.

Cependant, pour implémenter efficacement des algorithmes complexes en Java, nous allons voir ici les structures de base d'une classe et de l'instanciation.

1.b De la conception à l'exécution d'un programme

Dans ce cours, nous allons voir comment concevoir et réaliser un programme en Java. La programmation n'étant pas qu'une affaire de clavier, des **étapes de réflexion** sont à mettre en œuvre **avant, pendant et après l'étape de programmation**. Pour nous guider dans ces étapes de réflexion, nous allons utiliser 2 types de diagramme tout au long de cette année:

- Le diagramme UML de classes pour concevoir la structure de nos classes Java (détaillé dans le chapitre 4)
- Le diagramme APO pour représenter le déroulement du programme dans la mémoire (détaillé dans le chapitre 5) et ainsi mieux appréhender les comportements attendus et inattendus d'un programme.

Le diagramme de classes UML

Le langage UML (pour *Unified Modeling Language*) permet de modéliser différentes étapes de design logiciel. Dans ce cours, nous utiliserons exclusivement le diagramme de classes qui permet de décrire le squelette et l'organisation des classes entre elles. La section suivante aborde la construction d'un diagramme de classes UML par l'exemple. La construction détaillée dans le cas général d'un diagramme UML de classes sera abordé chapitre 4. D'autres diagrammes UML seront vus en M1 MIAGE.

Le diagramme APO

Le diagramme APO (pour *Algorithmique et Programmation Objet*), permet de représenter la mémoire de l'ordinateur pendant l'exécution d'un programme Java. Cela permet non seulement de bien comprendre ce qui se passe **exactement** lors de l'exécution de son programme, mais également de prévoir des comportements non désirés. La section suivante aborde la construction d'un diagramme APO par l'exemple. Le chapitre 5 explique comment il doit être construit dans le cas général.

2 Exemple détaillé : modélisation d'un article en Java

2.a Problématique utilisée

On considère la modélisation d'articles vendus sur un site marchand. Un **Article** est représenté par son **nom**, une chaîne de caractères, une **codeBarre**, de type entier et un **prix** de type réel. Une instance de document XML représentant un article pourrait être le suivant :

```

1 <?xml version="1.0" encoding="utf8"?>
2 <article>
3   <nom>Ordinateur HP 17 inch</nom>
4   <codeBarre>186942</codeBarre>
5   <prix>399.99</prix>
6 </article>
```

ou encore

```

1 <?xml version="1.0" encoding="utf8"?>
2 <article>
3   <nom>XP-Pen Artist 15.6Pro</nom>
4   <codeBarre>185924</codeBarre>
5   <prix>429.99</prix>
6 </article>
```

2.b Le modèle utilisé: la classe

En Java, une classe est la description d'un type complexe (en XML, on peut la comparer à XMLSchema). Les sous-éléments d'une classe sont *des attributs*¹.

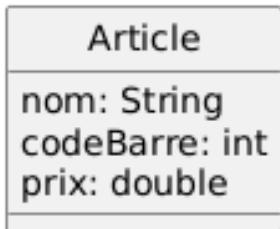
Dans notre exemple, la classe **Article** aura 3 attributs :

¹Nous verrons plus tard qu'il y a d'autres membres d'une classe, les méthodes.

- nom de type *chaîne de caractères*
- codeBarre de type *entier*
- prix de type *réel*

On peut représenter le *modèle* de cette classe en UML² et en Java comme ci-dessous:

En UML:



En Java:

```

1 class Article {
2     String nom;
3     int codeBarre;
4     double prix;
5 }
  
```

En UML, une classe est représentée par une boîte. Au sommet est écrit le nom de la classe. Après un trait horizontal, on note les attributs sous la forme **nom: type**. On note un seul attribut par ligne.

En Java, on écrit, dans un fichier **qui a le même nom que la classe**, et comme extensions **.java**; le mot clé **java class** qui indique que l'on va déclarer une classe, suivi du nom de la classe. Le corps de la classe est délimité par des accolades **{ }**. A l'intérieur des accolades, on définit les attributs sous la forme **type nom;**. On définit également un seul attribut par ligne.



Au cours des parties **A** et **B** de ce cours, nous ne mettrons aucune décoration aux attributs. C'est-à-dire qu'ils seront définis uniquement avec leur type, leur nom et un **;**. Nous verrons et utiliserons les modificateurs d'accès dans les partie **C** et **D** uniquement.

Nous venons de définir le *modèle*, c'est-à-dire la classe d'un article. Les documents XML précédents représentent deux exemples, on dit aussi **instances** d'article différents.

3 Du statique au dynamique: exécution d'un programme Java

3.a Le **main** et la chaîne de traitements

Comme vu au chapitre **2**, en Java, pour dérouler un programme, on va passer par une classe à part (souvent appelée **Test**) qui contiendra une méthode particulière: la méthode **main**.

Définition



Le **main** est une action spécifique (une *méthode* en Java) qui contient la première instruction à exécuter quand on lance un programme (exécute le programme / l'application).

C'est une opération spécifique et **unique** pour un programme / une application.

Le **main** permet de formuler la *première instruction*, c'est-à-dire la première ligne de code à exécuter. Une fois la première instruction exécutée, on peut également avoir d'autres instructions. Les instructions suivantes sont écrites à la suite, dans les lignes suivantes de l'opération **main**. La chaîne de traitements décrit la marche à suivre pas à pas. Une fois le programme *lancé*, le **main** est exécuté et on commence à manipuler les informations dans la mémoire vive (RAM pour *Random Access Memory*).

La chaîne de traitements est définie par les algorithmes du programme.

²Dans ce cours UML désignera uniquement *diagramme UML de classes*. D'autres diagrammes UML seront vus en M1.



Il y a une différence entre *algorithmique* et *programmation*. Un algorithme s'exprime:

- en langage naturel (ex: recette de cuisine)
- en langage dédié à l'algorithmique (abstrait) (cf D. Knuth *The Art of Computer Programming*)
- en langage de programmation. Ici, nous utiliserons le langage Java.

Nous représenterons l'exécution d'un programme, c'est-à-dire l'exécution de la chaîne de traitements grâce au diagramme APO (Algorithmique et Programmation Objet).

Définition



Les *diagrammes APO* représentent l'état de la mémoire vive c'est-à-dire les données telles qu'elles sont stockées dans la mémoire vive à un moment précis de l'exécution de la chaîne de traitements.

Note



Le diagramme APO, comme le diagramme de classes UML, peuvent être utilisés quelque soit le langage de programmation (Java, C++, C#).

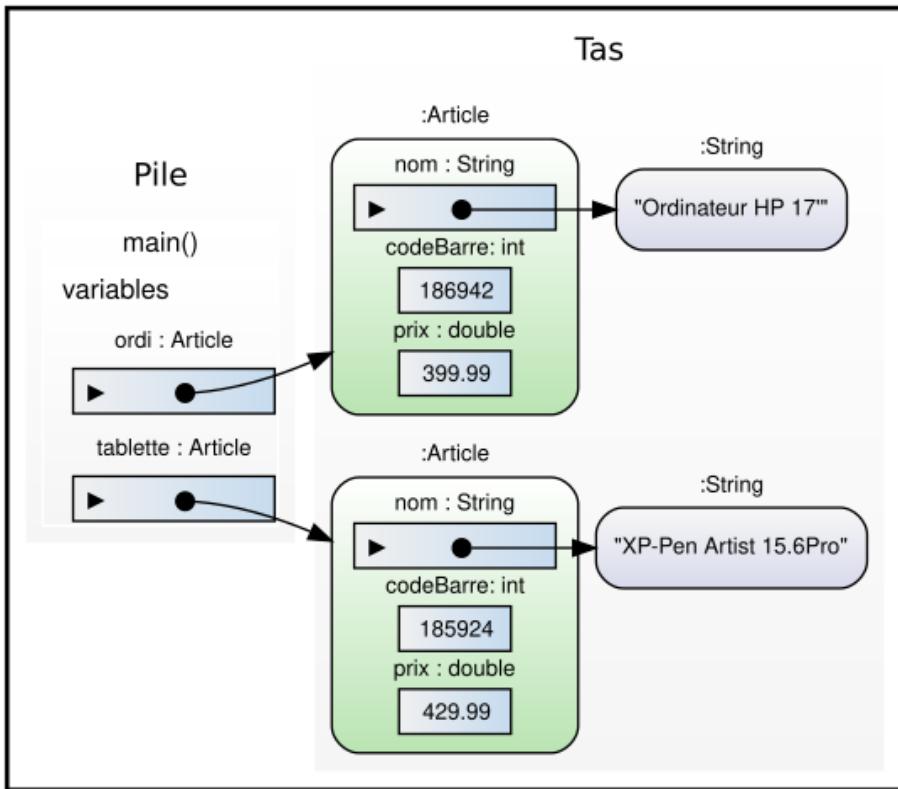
4 Exemple d'exécution et d'instanciation

Si l'on considère les exemples de la section 2.a, on va créer dans le `main` deux exemples, on dira deux **instances d'Article**. On obtiendra, dans la classe `Test`, la méthode `main` suivante:

```

1 class Test {
2     public static void main(String[] args) {
3         Article ordi;
4         ordi = new Article();
5         ordi.nom = "Ordinateur HP 17";
6         ordi.reference = 186942;
7         ordi.prix = 399.99;
8
9         Article tablette;
10        tablette = new Article();
11        tablette.nom = "XP-Pen Artist 15.6Pro";
12        tablette.reference = 185924;
13        tablette.prix = 429.99;
14    }
15 }
```

A la fin de ce programme, nous aurons l'état de la mémoire représentée par le diagramme APO ci-dessous:



5 Exécution pas à pas

5.a Déclaration

En Java, la déclaration d'une variable se fait par l'expression :

type nom;

Cela crée une **case** en mémoire qui est nommée `nom` et qui est prête pour prendre comme valeur quelque chose de type `type`.

Le code du `main` ci-dessus donne 2 exemples de déclaration de variable:

- ligne 3: `Article ordi;`
- ligne 9: `Article tablette;`

Ces 2 variables sont de type `Article`, l'une s'appelle `ordi` et l'autre `tablette`.

On peut citer d'autres exemples:

```

1 int i;           // c'est une déclaration de i de type int (primitif)
2 Article t;       // c'est une déclaration de t de type référence vers un Article
3 t j;             // c'est une déclaration de j de type t (primitif)
4 T k;             // c'est une déclaration de k de type référence vers le type T

```

Note

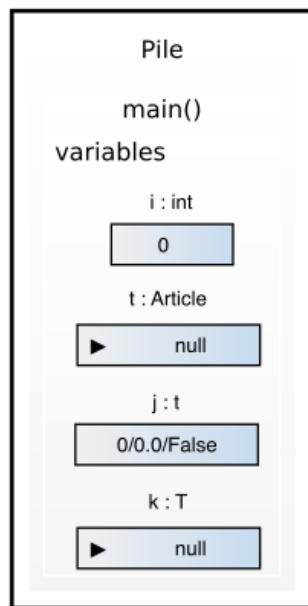
En mathématique, on déclare également des variables, par exemple, soit $x \in \mathbb{R}$.



On remarque qu'il existe 2 familles de données (elles seront détaillées chapitre 5, section 2.b):

- les données de type primitif
- les données de type référence

Dans le diagramme APO, on distingue les 2 :

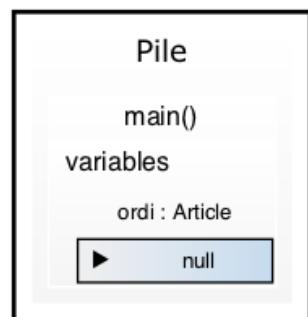


On peut remarquer qu'**Article** commence par une Majuscule, ce qui signifie que l'on est en train d'utiliser la classe **Article**. Si c'est une classe, c'est qu'elle contient d'autres informations. Lorsque l'on utilise une classe par l'intermédiaire d'une instance de classe, la taille allouée au stockage en mémoire est variable et dépend de la classe. Certaines classes contiennent beaucoup d'informations (beaucoup de sous-éléments = beaucoup d'attributs) et d'autres classes contiennent peu d'informations (exemple: la classe **Article** n'a que 3 attributs). Chaque instance de la classe **Article** doit contenir 3 attributs. Une instance d'une classe qui contient beaucoup plus d'information aura besoin de plus d'espace mémoire.

Il faut bien distinguer :

- la variable **t** qui est une référence (la taille est toujours la même, quelque soit le type de la référence)
- et l'instance qui est en mémoire et qui permet de stocker les informations dans les attributs.

Dans notre exemple, la déclaration de la variable **ordi** de type **Article** se fait de la manière suivante :



Notez la flèche dans la case qui indique que l'on a affaire à une référence.

5.b Instanciation

En Java, l'instanciation, c'est-à-dire la création d'une instance se fait en 3 phases.

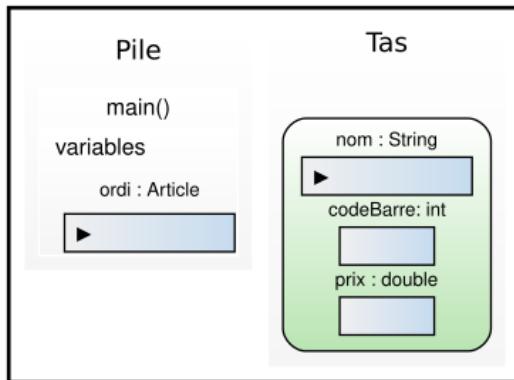
```
4     ordi = new Article();
//      [3] [1]      [2]
```

L'opérateur `new` permet d'**instancier** (c'est-à-dire créer une instance) une classe. Grâce à l'instruction `new NomDeLaClasse`, on va réserver un nouvel emplacement dans la mémoire (allocation mémoire) qui est capable de stocker toutes les données définies par la classe `NomDeLaClasse`.

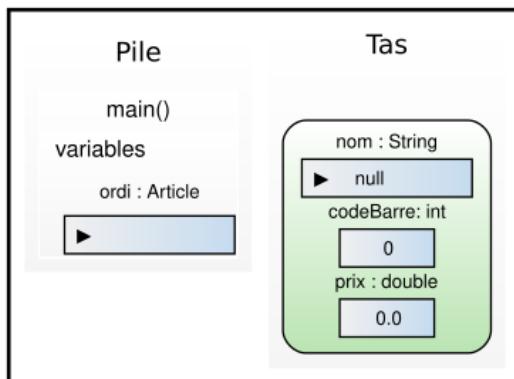
En effet, une instance est fabriquée à partir du **modèle** défini dans la classe. Dans la classe `Article` on a défini 3 attributs ; on a modélisé le concept d'`Article` comme étant quelque chose qui doit comporter 3 éléments : un nom, un code barre et un prix. C'est-à-dire qu'au moment où on a défini la classe `Article`, on a prévu qu'un article aurait forcément ces 3 éléments. Le modélisateur du système d'information a défini ce qu'était un `Article`. Lorsque l'on utilise une instance de cette classe, elle aura forcément 3 parties.

Sur la ligne 4, il se passe 3 choses:

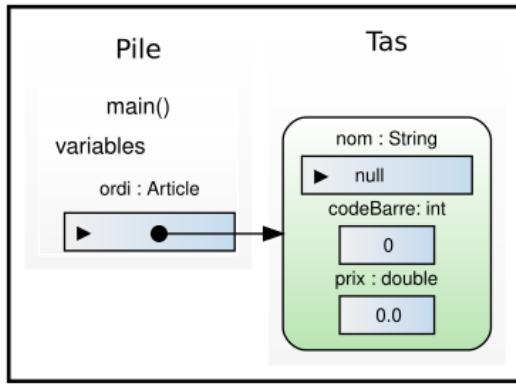
1. **new: instantiation** / création de l'emplacement dans la mémoire.



2. **Article()** : appel à une action spécifique qui permet d'initialiser les attributs à leur valeur par défaut (le **constructeur** par défaut)



3. **= : affectation** de l'emplacement mémoire (adresse mémoire) à la variable de type référence.



Définition



Un constructeur est une action spécifique d'une classe qui permet d'initialiser les attributs lors de l'instanciation.

Un constructeur est une action qui a le même nom que la classe.

Dans l'exemple ci-dessus, on n'a pas défini d'action dans la classe `Article`. Lorsqu'il n'y a pas de constructeur défini dans une classe Java, Java crée automatiquement et silencieusement (implicitement) un constructeur.

Définition



Le constructeur par défaut est une action définie dans la classe, qui a le même nom que la classe et qui n'a pas de paramètre. Lorsque le constructeur par défaut est défini implicitement, il initialise les attributs à leurs valeurs par défaut.

Définition



En Java, une **instance** est un objet en mémoire créé à partir du modèle défini dans une classe.

Pour créer une instance, il faut:

- une classe
- un constructeur
- l'opérateur `new`

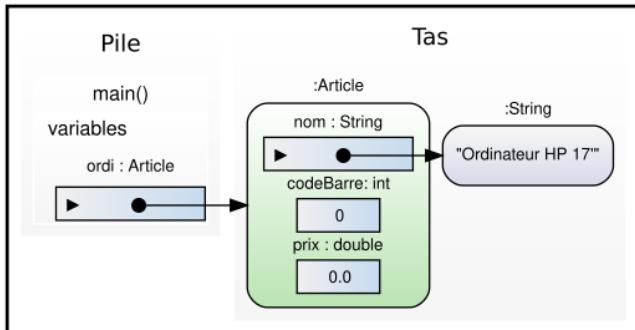
5.c Initialisation

```

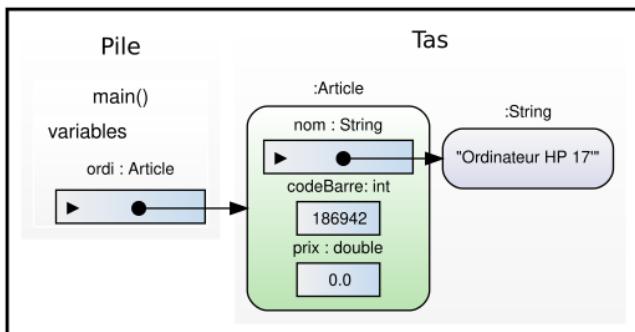
5     ordi.nom = "Ordinateur HP 17";
6     ordi.codeBarre = 186942;
7     ordi.prix = 399.99;

```

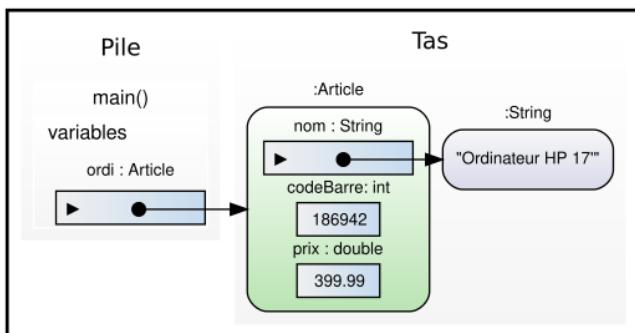
- ligne 5 : `ordi.nom = "Ordinateur HP 17";`



- ligne 6 : `ordi.codeBarre = 186942;`



- ligne 7 : `ordi.prix = 399.99;`



L'initialisation est indispensable à toute utilisation. On utilise généralement l'opérateur `=` qui permet d'affecter une valeur à un attribut. On peut également demander des valeurs à l'utilisateur, à un web service sur une autre machine, à une base de données, etc.

Le diagramme UML de Classes

Fiche

Prérequis	Chapitre 3: Classes et instances
Objectif pédagogique	Se donner des outils pour communiquer entre concepteurs/développeurs et enseignants/étudiants afin d'acquérir des concepts avancés de Programmation Orientée Objet
Intérêt	<ul style="list-style-type: none"> Représenter les concepts simples et avancés de la POO de manière explicite et compréhensible par tous. Faciliter la capacité d'abstraction.
Compétences à acquérir	<ul style="list-style-type: none"> Être capable d'écrire une/des classe(s) Java à partir d'un diagramme UML de classes Être capable d'écrire un diagramme UML de classes à partir d'une/plusieurs classe(s) Java Être capable de concevoir des diagrammes de classes UML simples à partir d'un énoncé général (cette compétence sera approfondie en M1).

1 Pourquoi dessiner des diagrammes de classes UML ?

Nous utiliserons, au cours de l'UE APO, 2 types de diagrammes:

- le diagramme UML de classes pour réfléchir à la modélisation avant d'implémenter et ainsi mieux modéliser
- le diagramme APO pour comprendre ce qui se passe à l'exécution et ainsi mieux programmer.

Il ne s'agit pas seulement de faire des dessins pour *faire des dessins*, mais de représenter les concepts les plus complexes et avancés de la programmation orientée objet de manière explicite et compréhensible et faciliter ainsi la capacité d'abstraction.

Il existe de nombreux types de diagrammes UML qui permettent de spécifier, puis modéliser un système avant de l'implémenter. Des cours de Master MIAGE sont dédiés à la spécification et la modélisation à partir d'un cahier des charges abstrait et font références à plusieurs types de diagrammes UML. L'un des objectifs pédagogiques de ces cours de M1 et M2 est de savoir concevoir un système (en Java ou autre langage) totalement à partir d'un cahier des charges en construisant notamment des diagrammes de classes UML (et bien sûr de l'implémenter ensuite).

L'objectif pédagogique de cette UE concernant les diagrammes UML de classes est de savoir lire, comprendre et implémenter en Java un diagramme de classes UML complet (avec toutes les subtilités qu'il peut comporter). C'est-à-dire de maîtriser les concepts de programmation orientée objet et leur représentation aussi bien en diagrammes de classes UML qu'en Java. Vous devrez être capables, à la fin de cette UE de traduire n'importe quel diagramme de classes UML objet en Java et réciproquement.

Ce chapitre intervient avant que nous ayons vu tous les concepts avancés de programmation orientée objet (héritage, polymorphisme, interfaces, etc.). Aussi, il présente en détail les représentations des concepts du chapitre 3 "Classe et instance" qui le précède, et présente succinctement dans la dernière section, la représentation UML des concepts qui seront vus dans les chapitres suivants. On reviendra donc à ce chapitre au fur et à mesure de l'avancement du cours.

2 Les éléments de base du diagramme de classes UML

2.a Modificateurs d'accès

Le tableau suivant énumère les modificateurs d'accès possibles en Java.

Modificateur d'accès UML	Modificateur d'accès Java	Description
+	<code>public</code>	Visible partout (autres classes, autres packages)
-	<code>private</code>	Visible uniquement dans la classe elle-même
#	<code>protected</code>	Visible uniquement dans la famille.
	<code>Ø (rien)</code>	Visible dans tout le package



Vous ne devrez implémenter les modificateurs d'accès en Java qu'une fois que vous les maîtriserez, c'est-à-dire à partir du chapitre 14.

2.b La classe

En diagramme de classes UML (on écrira juste UML par la suite pour des raisons de rapidité même s'il s'agit d'un abus de langage étant donné que dans ce cours, seuls les diagrammes de classes UML seront abordés), une classe est représentée par un rectangle qui contient 3 compartiments.

Le premier compartiment contient le nom de la classe, le deuxième les attributs et le troisième, les méthodes.

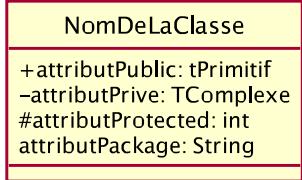
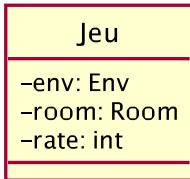
	En UML	En Java
Syntaxe		<code>class NomDeLaClasse { }</code>
Exemple		<code>public class Jeu { }</code>

2.c Les attributs

En programmation orientée objet, c'est-à-dire en Java comme en UML, un attribut est défini par un nom, un type et un modificateur d'accès. En UML, un attribut est déclaré par

- son modificateur d'accès
- son nom
- deux points ‘::’ (semicolon en anglais)

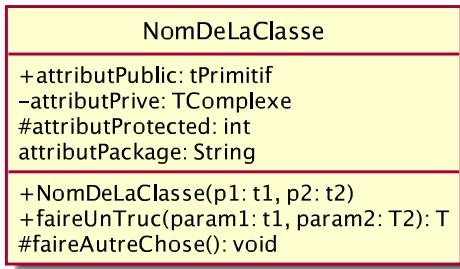
- son type

	En UML	En Java
Syntaxe	 <pre>+attributPublic: tPrimitif -attributPrive: TComplexe #attributProtected: int attributPackage: String</pre>	<pre>class NomDeLaClasse { public tPrimitif attributPublic; private TComplexe attributPrive; protected int attributProtected; String attributPackage; }</pre>
Exemple	 <pre>-env: Env -room: Room -rate: int</pre>	<pre>public class Jeu { private Env env; private Room room; private int rate; }</pre>

2.d Les méthodes

En programmation orientée objet, les méthodes sont définies par leur signature, c'est-à-dire leur nom, leur type, leur modificateur d'accès et leurs paramètres.

En UML, on donne d'abord le modificateur d'accès, puis le nom de la méthode, suivi de deux points ‘:’ et le type. Les paramètres sont déclarés dans les parenthèses de la méthodes, séparés par des virgules et déclarés par leur nom, ‘:’ et leur type.

	En UML	En Java
Syntaxe	 <pre>+attributPublic: tPrimitif -attributPrive: TComplexe #attributProtected: int attributPackage: String +NomDeLaClasse(p1: t1, p2: t2) +faireUnTruc(param1: t1, param2: T2): T #faireAutreChose(): void</pre>	<pre>class NomDeLaClasse { public tPrimitif attributPublic; private TComplexe attributPrive; protected int attributProtected; String attributPackage; public NomDeLaClasse(t1 p1, t2 p2) { // Implémentation du // constructeur } public T faireUnTruc(t1 param1, T2 param2) { // Implémentation de la méthode faireUnTruc } protected void faireAutreChose() { // Implémentation de la méthode faireAutreChose } }</pre>

	En UML	En Java
Exemple	<pre> classDiagram class Jeu { -env: Env -room: Room -rate: int +Jeu() +execute(): boolean #demarrePartie(partie: Partie): void } </pre>	<pre> public class Jeu { private Env env; private Room room; private int rate; public Jeu() { // Implémentation du // constructeur } public boolean execute() { // Implémentation de la méthode } protected void demarrePartie(Partie partie) { // Implémentation } } </pre>



Une méthode aura toujours des parenthèses à la fin de son nom, même si elle n'a pas de paramètres.

Note



Les constructeurs n'ont pas de types, on les déclare uniquement avec leur modificateur d'accès, leur nom et leurs paramètres.

3 Les relations entre les classes

Une grande partie de ce cours est extraite du site <https://www.emse.fr/~boissier/enseignement/aco/pdf/UML.Classe.4pp.pdf>.

3.a Associations

Une association est une relation entre plusieurs classes. Elle est modélisée par un trait entre ces classes. Le plus souvent, les associations ont une *arité binaire*, c'est-à-dire qu'elles mettent 2 classes en relation.

Note



Dans ce cours, nous ne nous intéresserons qu'aux associations binaires.

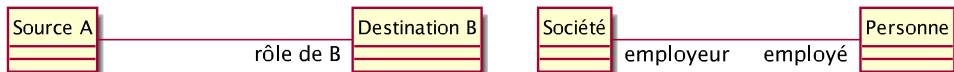
Les associations peuvent être nommées, c'est-à-dire identifiées par un texte unique décrivant la sémantique de l'association.



S'il y a ambiguïté, on peut indiquer le sens de lecture avec le signes de flèche pleine.

Rôle des extrémités

Les extrémités des associations peuvent être qualifiées par des rôles. Un rôle indique comment une classe Source voit une classe Destination.



Multiplicité

La multiplicité d'une association précise le nombre d'instances pouvant être liées par une extrémité à une instance de l'autre extrémité.

Sous sa forme générale, elle s'exprime par une suite d'intervalles disjoints sur l'ensemble des entiers naturels.



En UML, on a les multiplicités suivantes:

Multiplicité	Définition
1	un et un seul (multiplicité par défaut)
0..1	zéro ou 1
N	exactement N
M..N	de M à N
*	zéro ou plus (de 0 à +∞)
0..*	zéro ou plus (de 0 à +∞)
1..*	un ou plus (de 1 à +∞)

Cas particuliers

Note



On peut également qualifier ou contraindre des associations, et créer des classe-association. Ces notions ne seront pas traitées dans ce cours, vous pouvez pour cela vous reporter à <https://www.emse.fr/~boissier/enseignement/aco/pdf/UML.Classe.4pp.pdf> ou à <https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes>.

Dans ce cours, nous utiliserons exclusivement **3 formes spéciales d'association** : la composition, l'aggrégation et la généralisation (l'héritage ou l'implémentation) qui sont détaillées ci-dessous.

3.b Aggrégation ◊—

L'aggrégation, est une forme spéciale d'association qui exprime une relation de composition entre aggrégats. L'aggrégation n'est ni réflexive ni symétrique et l'une des extrémités joue un rôle prédominant.

A travers une aggrégation, il est possible de représenter:

- la propagation des valeurs d'attributs d'une classe vers l'autre
- une action sur une classe qui implique une action sur une autre classe
- une subordination des objets d'une classe à ceux d'une autre



3.c Composition ↵

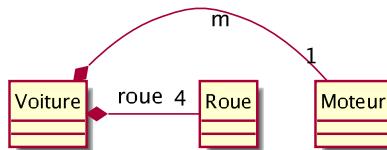
La composition est un cas particulier d'aggrégation. La classe ayant le rôle prédominant est la classe *composite* ou classe conteneur.

La composition implique :

- la durée de vie des composants est la même que celle du composite
- la multiplicité du côté du composite prend ses valeurs dans 0 ou 1



est équivalent à



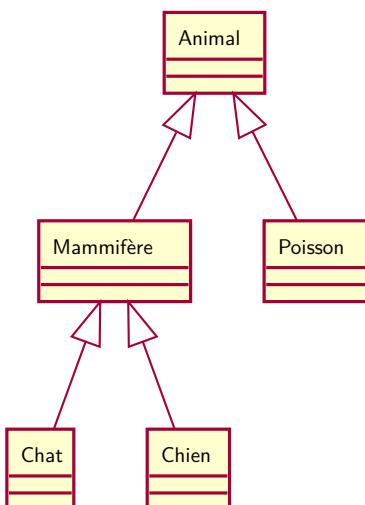
Dans le cas des classes Java auxquelles on s'intéressera dans ce cours, on considérera systématiquement les attributs comme composites d'une classe. La décision quant au choix de l'aggrégation ou de la composition se fait lors de la modélisation à partir du cahier des charges, qui est au delà des objectifs de ce cours.

3.d Généralisation ↵

La généralisation est une relation entre une classe plus générale et une classe plus spécifique. Elle peut généralement être exprimée par *est un* ou *est une sorte de...*



En Java et dans ce cours, la généralisation sera exprimée par l'héritage. L'héritage et le polymorphisme qui en découlent seront vu au chapitre 16.



Note

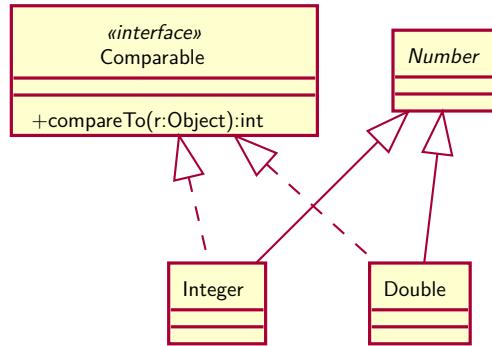


Il existe dans certains langages des généralisations (héritages) multiples. Ce n'est pas le cas en Java et cela ne sera pas abordé dans ce cours.

3.e Interface

Une interface est la description d'un ensemble d'opérations utilisées pour spécifier un service offert par une classe. Les interfaces en Java seront vues au chapitre 19.

En UML, une interface est représentée par une classe ayant le *séthéotype* «interface». La flèche d'une classe qui implémente une interface vers l'interface en question est une flèche d'héritage en pointillés.

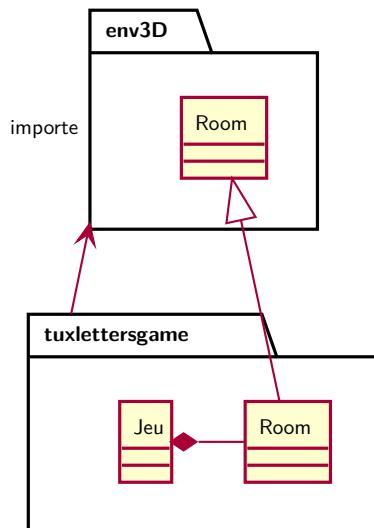


4 Autres éléments

4.a Le paquetage

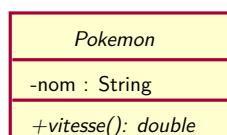
Le paquetage est un mécanisme de partitionnement des modèles et de regroupement des éléments de modélisation. Le paquetage en Java sera vu au chapitre 13.

Chaque paquetage peut contenir un ensemble de diagrammes et/ou de paquetage. Chaque élément d'un paquetage possède un nom unique dans ce paquetage. En UML, il y a la possibilité de définir des relations entre les paquetages.



4.b Classes et méthodes abstraites

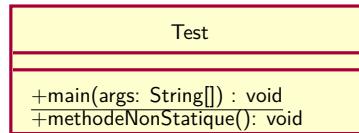
En UML, une méthode ou une classe abstraite est représentée en italique. Les classes abstraites seront vues au chapitre 18.



Lorsqu'il n'est pas possible d'utiliser un italique (quand on fait le diagramme à la main par exemple), on peut rajouter la mention `{abstract}` à la suite de la méthode ou du nom de classe.

4.c Variables et méthodes statiques

En UML, une variable ou une méthode statique est représentée soulignée. Les méthodes et variables statiques seront vues au chapitre 22.



Déroulement d'un programme: Le diagramme APO

Fiche

<u>Prérequis</u>	Chapitre 3 : Classe et Instance
<u>Objectif pédagogique</u>	Se donner des outils pour communiquer entre concepteurs/développeurs et enseignants/étudiants afin d'acquérir des concepts avancés d'algorithmique et de Programmation Orientée Objet
<u>Intérêt</u>	<ul style="list-style-type: none"> Comprendre ce qu'il se passe pendant l'exécution d'un programme Java Etre capable de prévoir (et ainsi de prévenir) les erreurs d'un programme que l'on est en train d'écrire Maîtriser les différences de comportement entre les variables, attributs, paramètres, instances, etc.
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> Etre capable de dessiner un diagramme APO à partir d'un programme Java simple Etre capable d'écrire un programme Java à partir d'un diagramme APO Etre capable d'expliquer et de prévoir le comportement des variables, attributs, paramètres, instances et références d'un programme Java donné en justifiant sa réponse.

1 Pourquoi dessiner des diagrammes APO ?

Le diagramme de classes UML permet de représenter les classes et leur organisation lors de la conception d'un programme. La vue d'ensemble de l'organisation d'un programme est une étape importante avant le début de la programmation. Cependant, ce diagramme ne permet pas de représenter les instances, ni ce qui se passera en mémoire **pendant l'exécution du programme**.

Le diagramme APO (pour *Algorithmique et Programmation Objet*) permet de représenter et donc de comprendre / maîtriser le comportement des variables, attributs, paramètres, instances et références lors

de la phase d'exécution d'un programme. Il est nécessaire pour la rédaction et la correction d'algorithmes car il permet de prédire, éviter ou corriger des erreurs sur des programmes.

2 Catégories et familles de données

Dans la plupart des langages de programmation, on manipule des variables *typées*, c'est-à-dire où le type est défini lors de la déclaration de la donnée¹.

Définition



Une donnée dans un programme est définie par

- son nom
- son type
- sa valeur

Le nom et le type sont indiqués lors de la **déclaration**. Le type va déterminer la taille de l'emplacement mémoire de la donnée.

Les données sont utilisées à différents endroits d'un programme. On distingue les données utilisées en fonction:

- de l'endroit où elles sont déclarées, ce qui définit leur **catégorie**
- de la façon dont elles sont stockées en mémoire, ce qui définit leur **famille**

2.a Catégories

Il existe 3 catégories qui correspondent à 3 endroits distincts d'une classe ou d'un programme dans lesquels on peut déclarer une donnée :

- les variables
- les paramètres
- les attributs

Variable

Une variable est une donnée nécessaire aux calculs effectués à l'intérieur d'un algorithme. On se sert d'une variable notamment pour :

- diriger l'algorithme
- faire des calculs intermédiaires

Les algorithmes sont définis dans les actions / méthodes / opérations d'une classe.

Note



La portée d'une variable est la méthode ou l'instruction algorithmique dans lequel elle a été déclarée.

Conséquence : à la fin d'une méthode, si le résultat du calcul ou de l'algorithme est important (c'est-à-dire si on veut pouvoir s'en servir), il faut :

- soit transférer le résultat dans un attribut
- soit renvoyer le résultat à celui qui a demandé son calcul

Dans un bloc d'une méthode la dernière instruction peut donc être une instruction qui permet de renvoyer une valeur :

```
return _____;
// _____ est le nom d'un variable
```

¹Il existe des langages dans lesquels le type est déduit automatiquement de la valeur de la variable, par exemple en Python ou Javascript

Paramètre

Un paramètre est une donnée **supplémentaire** indispensable au bon déroulement d'un algorithme et qui ne peut pas être déduite des données déjà disponibles dans l'algorithme.

C'est une donnée *d'entrée* fournie par celui qui a demandé l'exécution de l'algorithme.

Exemple en maths : on a l'algorithme "racine carrée", on lui fourni le nombre à partir duquel il faut faire le calcul. Il s'écrit: $\begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \mapsto & f(x) = \sqrt{x} \end{array}$. Dans ce cas, x est le paramètre de la fonction $f = \sqrt{}$ pour lequel on veut effectuer le calcul.

Exemple en java :

```
class TestMath {
    public static void main(String args[]) {
        // ici le "main" fait appel à l'opération "Math.sqrt(...)"
        Math.sqrt(45.0); // appel de la méthode "Math.sqrt(...)" avec le paramètre
        45
    }
}
```

```
// quelque part dans le JDK on a une class Math
class Math {
    ...
    ... sqrt(double x) {
        // Ici utiliser un développement limité pour calculer la racine
        // carrée de x
        // Au moment où la ligne 5 du programme principal ci-dessus s'exécute,
        // dans le bloc sqrt(..), x aura la valeur 45.0
    }
}
```

On a besoin de définir un paramètre :

- quand on ne dispose pas de la donnée sous forme d'attribut
- ni de moyen pour calculer cette donnée à partir de rien (*ex nihilo*)
- ni de moyen d'y accéder

Un paramètre est toujours associé aux données d'entrée d'un algorithme / d'une méthode. Les paramètres font partie de la définition des méthodes. Ils sont déclarés entre les parenthèses suivant le nom de la méthode. S'il y a plusieurs paramètres, leurs déclarations sont séparées par des virgules.

En UML, comme en Java, on a

`nomMéthode(..., ...)` // ici il y a 2 paramètres

La déclaration d'un paramètre se fait comme une déclaration de variable :

En UML	En Java
nom: Type	Type nom

Cette déclaration produit le même effet : un espace de la mémoire est réservée pour stocker la donnée qui correspond au paramètre. La portée d'un paramètre est la méthode dans laquelle ce paramètre est déclaré.

Attribut

Un attribut (on utilise aussi le terme *variable d'instance*) sert à stocker les données permettant de caractériser l'état d'une instance.

Exemple : une instance de la classe **Complexe** a deux attributs (**r** et **i**) qui de part leur valeur caractérisent l'état du nombre complexe.

La déclaration d'un attribut se fait *directement* à l'intérieur du bloc de la classe, c'est-à-dire en *dehors* de tout bloc d'actions / de méthodes. La portée d'un attribut *au minimum* l'ensemble de la classe : un attribut est accessible dans toutes les méthodes / actions de la classe (voir chapitre 01).

2.b Famille

Il y a deux familles possibles pour déclarer une variable, un paramètre ou un attribut:

- la famille des types primitifs
- la famille des types références

Type primitif

Il existe 9 types primitifs en Java : `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean` et `void`. On ne peut pas créer de nouveaux types primitifs en Java. Les types primitifs seront détaillés dans le chapitre 6.

Type référence

Définition



Une *instance* est un objet dans la mémoire issu d'une classe.

Définition



Une **référence** est une donnée (un attribut, une variable ou un paramètre) dont la valeur

- est une adresse/emplacement dans la mémoire où se situe une instance ou
- est la valeur `null`

Une référence ne peut référencer qu'**une seule instance**. Une instance peut être référencée par plusieurs références. Un type référence permet de stocker une *référence* vers une instance d'une classe. Bien que l'emplacement mémoire d'une variable de type référence occupe une taille fixe, l'instance va occuper un emplacement en mémoire de taille variable (qui dépend des attributs déclarés dans l'instance).

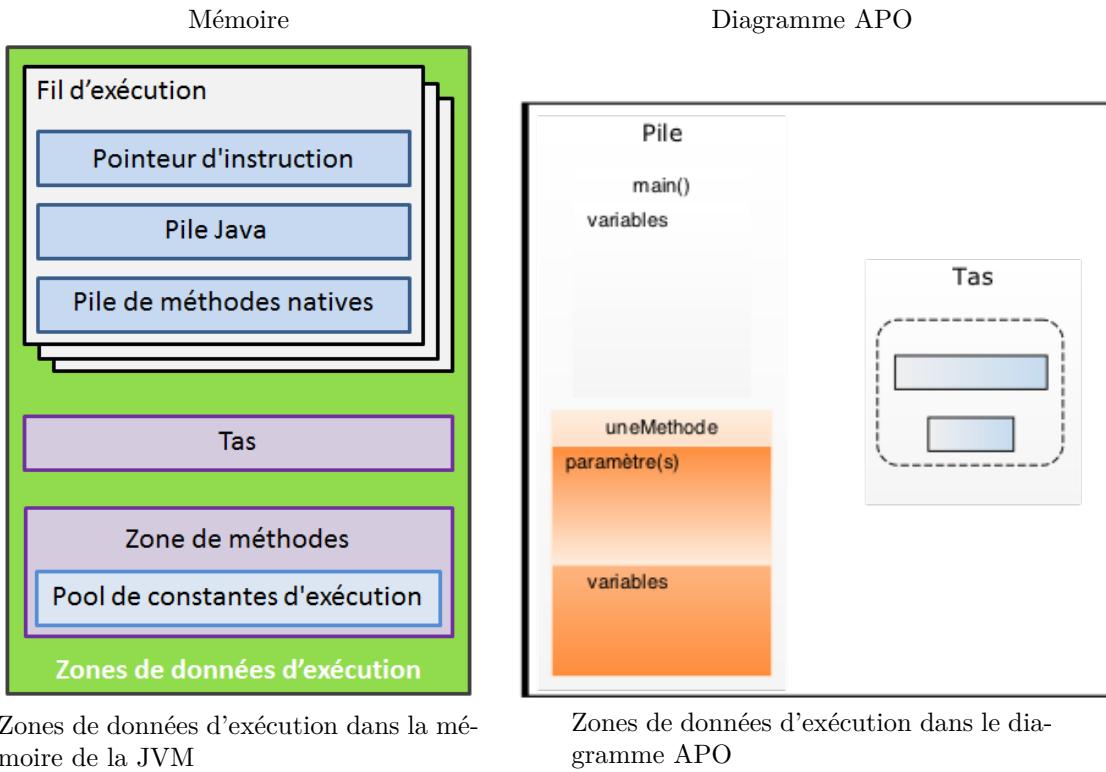
Note



En Java, il y a une famille qui n'existe pas, mais qu'on retrouve dans d'autres langages (comme C, C++, etc) : les *pointeurs*. Un pointeur est une donnée dont la valeur est un emplacement mémoire (même s'il ne correspond pas à une instance). Nous n'aborderons pas les pointeurs dans ce cours.

3 Squelette d'un diagramme APO

Un diagramme APO va représenter les éléments en mémoire lors de l'exécution d'un programme. Comme illustré dans la figure ci-dessous, un diagramme APO représente la Pile d'exécution à gauche et le Tas (Heap en anglais) à droite².



Dans la partie gauche du diagramme APO, on représente la **Pile** du fil principal d'exécution (on ne représentera ici qu'un seul fil d'exécution, le *multi-threading* ne sera pas abordé dans ce cours). Cette pile contient les variables du programme principal (**main**). Lorsqu'une méthode est appelée dans le programme principal, elle est représentée dans la pile, sous le **main**. Dans la partie droite du diagramme APO, on représente le **Tas**, qui contient les instances créées lors du programme. Ces deux parties sont détaillées dans les sections suivantes.

4 Les éléments d'un diagramme APO

4.a Les variables

Comme rappelé dans les sections ci-dessus, une variable est définie par un type, un nom et une valeur. Dans le diagramme APO, une variable est représentée par une case au dessus de laquelle est mentionné le nom de la variable, puis deux points : et le type de la variable. La valeur de la variable est indiquée dans la case. En APO, comme en programmation orientée objet en général, on différencie la famille des variables primitives et des variables références. Une référence sera indiquée par une flèche dans la case.

²La Pile et le Tas d'exécutions sont des espaces mémoire réservés dans la mémoire vive lors de l'exécution d'un programme. Ils seront vus en détail au deuxième semestre.

	Variable de type Primitif	Variable de type Référence
Syntaxe	nomDeVariable : typePrimitif ? 	nomDeVariable : Référence ► ? 
Exemple	nombreEtudiants : int 42 	enseignant : Enseignant ► null 

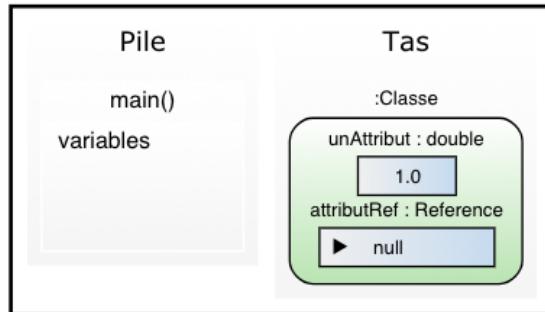
4.b Les instances

Instances

Dans le tas de la JVM comme du diagramme APO, sont représentés les instances. Une instance est un objet créé à partir d'une classe. En diagramme APO, on représente une instance comme une boîte qui contient des attributs. Au dessus de la boîte, on donne le type de l'instance précédé de ::.

Attributs

En diagramme APO, les attributs sont représentés à l'intérieur des instance, comme les variables (nom: type). Tout comme pour les variables, on distingue les attributs de type référence par une flèche à gauche de la case.



Principe des Troations

Le principe des Troations doit **impérativement** être respecté pour tout algorithme, tout programme et tout système d'information. Il énonce qu'il faut absolument passer par les trois étapes suivantes et dans cet ordre pour toute création d'instance :

1. Déclaration
2. instanciation
3. Initialisation



Ne pas oublier l'étape numéro 3 !
 La valeur par défaut des attributs de la famille référence est **null**.
 Une manipulation erronée d'une référence dont la valeur est **null** génère une **Null Pointer Exception** (NPE). Une NPE arrête **brutalement** l'exécution du programme -> "ça plante".

4.c Les méthodes

Lorsqu'une méthode est appelée dans la méthode `main`, elle apparaît à gauche du diagramme APO, dans la Pile, en dessous de la méthode `main`. Une méthode est appliquée sur une instance via sa référence³. En Java, une méthode est appelée via `referenceVersInstance.nomMéthode(paramètres)`. La représentation dans le Tas de l'instance sur laquelle est appelée la méthode (l'instance référencée par `referenceVersInstance`) prend alors une couleur différente pour être associée à la méthode. Le titre du compartiment de la méthode est le nom de la méthode. En diagramme APO, une méthode comporte 2 sous-parties, l'une consacrée aux paramètres, l'autre aux variables. On peut noter que dans la méthode, on a également accès aux attributs de l'instance sélectionnée.

Les paramètres

En Java, les paramètres d'une méthode sont déclarés dans la signature de la méthode, entre les parenthèses. On les représente ici comme des cases qui ont un nom, :, un type (comme les variables) dans le compartiment paramètre. Leur nom correspond au nom qu'ils portent en tant que paramètre à l'intérieur de la méthode considérée. On peut noter, à gauche de chaque paramètre, un numéro qui correspond à sa position dans l'ordre des paramètres. Ce numéro permet de repérer d'où vient ce paramètre (et quelle est sa valeur) si l'on écrit à gauche des variables (généralement de la méthode `main`) le même numéro.

L'exemple ci-dessous montre l'appel à la méthode `somme()` sur une instance de `Entier` à partir de la méthode `main`. Le code est représenté ci-dessous:

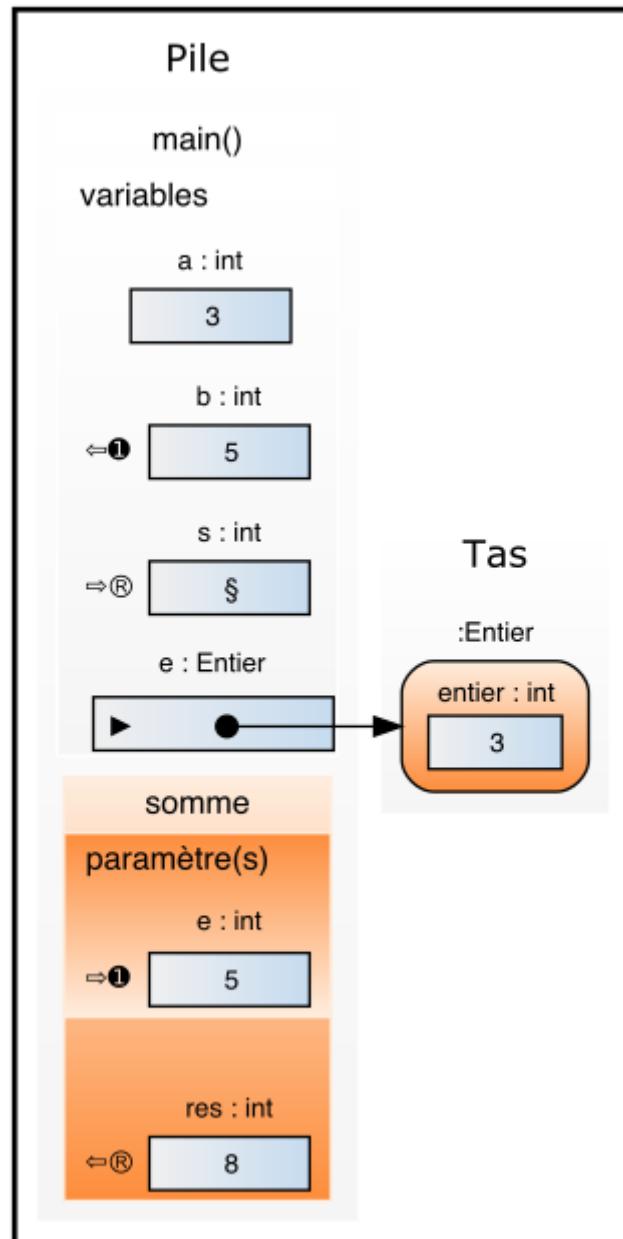
```

1 public class Entier {
2     private int entier;
3
4     public Entier(int e) {
5         entier = e;
6     }
7
8     public int somme(int entier) {
9         int res;
10        res = this.entier + entier;
11        return res;
12    }
13}
```

```

1 public class Test {
2     public static void main(String[] args) {
3         int a;
4         int b;
5         int s;
6         a = 3;
7         b = 5;
8         Entier e;
9         e = new Entier(a);
10        s = e.somme(b);
11    }
12}
```

³A part dans le cas des méthodes `static` (que nous verrons au deuxième semestre)



En Java les paramètres sont toujours passés par **valeur**. C'est-à-dire que la valeur du paramètre est une copie de la valeur de la variable initiale. En conséquence, si l'on change la valeur d'un paramètre à l'intérieur de la méthode, cela ne change pas la valeur de la variable dont il est issu. Cela est vrai pour les paramètres de type primitif **et** les références.

Types prédéfinis en Java

Fiche

<u>Prérequis</u>	Chapitre 3 : Classe et Instance, Chapitre 5 : Diagramme APO
<u>Objectif pédagogique</u>	Comprendre et maîtriser l'utilisation et les calculs sur les types prédéfinis en Java.
<u>Intérêt</u>	<ul style="list-style-type: none"> • Maîtriser les différents types prédéfinis en Java • Maîtriser les différences de comportement des opérations des différents types prédéfinis en Java • En déduire les erreurs de calculs possibles d'après les différents types de données
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> • Connaître les types prédéfinis en Java, savoir lesquels sont primitifs ou des références • Connaître les résultats des opérations possibles sur les différents types prédéfinis

1 Introduction



Attention à ne pas confondre type primitifs et types prédéfinis. En Java il y a 9 types *primitifs*, mais des centaines de types *prédéfinis*. Par exemple : les types `String`, `Date`, `[]`, ... Ce sont des types *prédéfinis* mais de type *référence* (on doit faire une déclaration de référence, une instanciation et une initialisation...)

Dans 99% des langages de programmation il existe des types prédéfinis, c'est-à-dire des types qui sont définis directement *dans* le langage (ils viennent avec le langage, leurs propriétés et les opérations possibles sont déjà définis dans le langage).

Chaque type prédéfini a un **nom** qui permet de désigner :

- un ensemble de valeurs possibles
- un ensemble d'opérations / d'opérateurs possibles

sur les variables/paramètres/attributs définis à l'aide de ce type.

2 Booléen

Nom du type en Java	<code>boolean</code>
Valeurs possibles	<code>true</code> et <code>false</code>
Valeur par défaut	<code>false</code>
Taille en mémoire	1 octet (ne devrait nécessiter qu'un bit)
Opérations possibles (ordre de priorité)	<code>!</code> (négation), <code>&&</code> (et puis), <code> </code> (ou bien)

Exemple :

```

1 boolean qcm;
2 boolean lacunes;
3 boolean nouvellesNotions;
4
5 // Ici, les 3 variables booléenne prennent des valeurs
6 // ...
7
8 // On utilise des expressions booléennes pour exprimer des conditions...
9 if (lacunes || nouvellesNotions && qcm) {
10     System.out.println("On révise l'AP0");
11 }
12 // Attention, il vaut mieux mettre les parenthèses de manières explicites
13 // A laquelle des deux expressions suivantes l'expression précédente est-elle é
14 // quivalente ?
14 if ((lacunes || nouvellesNotions) && qcm) {
15     System.out.println("On révise l'AP0");
16 }
17 // ou
18 if (lacunes || (nouvellesNotions && qcm)) {
19     System.out.println("On révise l'AP0");
20 }
```

Note



En programmation, `&&` correspond à l'expression `et puis` et `||` correspond à l'expression `ou bien`.

Pourquoi ?

En programmation si on a l'expression `(a && b)`, lors de l'exécution du programme :

- `a` sera évalué en **premier**
- si `a` est faux, l'expression `(a && b)` est forcément fausse et le programme **n'évalue pas** l'expression `b`
- si `a` est vrai, alors le programme évalue `b`
 - si `b` est faux, alors l'expression `(a && b)` est forcément fausse
 - si `b` est vrai, alors l'expression `(a && b)` est vraie

En programmation si on a l'expression `(a || b)`, lors de l'exécution du programme :

- `a` sera évalué en **premier**
- si `a` est vrai, l'expression `(a || b)` est forcément vraie et le programme **n'évalue pas** l'expression `b`
- si `a` est faux, alors le programme évalue `b`
 - si `b` est vrai, alors l'expression `(a || b)` est forcément vraie
 - si `b` est faux, alors l'expression `(a || b)` est fausse

Exemple : Table de Karnaugh pour (`a && b`).

Configuration	valeur de <code>a</code>	valeur de <code>b</code>	valeur de <code>a && b</code>
0	false	false	false
1	false	true	false
2	true	false	false
3	true	true	true

A titre d'exercice : table de Karnaugh du ‘ou bien’

Les variables de type `boolean` sont surtout utilisées dans des expressions qui vont permettre de diriger les flux du programme (exemple : `if`).

3 Entier

Il y a 4 types (qui font croire qu’ils sont des) "entiers" :

Nom du type	intervalle (valeurs possibles)	place mémoire occupée	valeur par défaut
<code>byte</code>	$[-2^7, \dots, +2^7 - 1]$	8 bits	0
<code>short</code>	$[-2^{15} \dots +2^{15} - 1]$	16 bits	0
<code>int</code>	$[-2147483648, \dots, +2147483647]$	32 bits	0
<code>long</code>	$[-9233372036854775808, \dots, 9233372036854775807]$	64 bits	0

Java propose un suffixe (non obligatoire) L¹ pour préciser qu’un entier est de type Long.

```
1 long population = 7800000000L;
2     long distanceToMoon = 384400000L;
3     System.out.println("World Population: " + population);
4     System.out.println("Distance to Moon in meters: " + distanceToMoon);
```

Note

Il n’existe pas de suffixe pour les types `byte`, `short` ou `int`.



3.a Opérations sur les entiers

- `++` : incrément de 1
- `--` : décrément de 1
- `*` : multiplication *entièr*e
- `/` : division *entièr*e
- `%` : modulo: reste de la division *entièr*e
- `+` et `-` : addition et soustraction *entiers*



Attention aux dépassements des bornes. Il y a plein de valeurs qui ne sont pas représentées...

¹Le suffixe l en minuscule existe également, mais est déprécié.

Note

Pour toutes les autres opérations (exposants, racine carrée, négation...): il faut utiliser une méthode / opération de la classe `Math`. Exemple : `Math.pow(x,y)` pour calculer x^y , `Math.sqrt(x)` pour calculer \sqrt{x} , etc.
On notera toutefois que $x^2 = x \times x$ et qu'il vaut mieux utiliser `x*x` que `Math.pow(x, 2)`.

Attention aux divisions: il s'agit de la division dans le domaine entier !



```
1 int i;
2 i = 7 / 2;
3 System.out.println(i); // affiche le quotient de la division entière : 3
4 i = 7 % 2;
5 System.out.println(i); // affiche le reste de la division entière : 1
```

Dans la division entière suivante :

$$\begin{array}{r} 7 \\ -6 \quad | \quad 2 \\ \hline = 1 \end{array}$$

- 3 est le quotient ($7/2$)
- 1 est le reste ($7 \% 2$)

3.b Entiers et bases

Par défaut, en mathématiques et en informatique (et donc en Java), on compte en base 10. Par exemple :

$$452 = 4 \times 10^2 + 5 \times 10^1 + 2 \times 10^0$$

Java permet d'encoder des entiers dans d'autres bases, à l'aide de différents préfixes.

- Le préfixe `0b` permet de coder des nombres en **binaire**. Par exemple :

```
1 int x = 0b1010; // => x = 10 en base décimale
```

$$0b1010 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 10$$

- Le préfixe `0` tout seul permet de coder des nombres en base **octale**. Par exemple :

```
1 int x = 012; // => x = 10 en base décimale
```

$$012 = (\text{préfixe } 0 \text{ puis}) 1 \times 8^1 + 2 \times 8^0 = 1 \times 8 + 2 \times 1 = 10$$

- Le préfixe `0x` permet de coder des nombres en base **hexadécimale**. Par exemple :

```
1 int x = 0x12; // => x = 18 en base décimale
```

$$0x12 = (\text{préfixe } 0x \text{ puis}) 1 \times 16^1 + 2 \times 16^0 = 1 \times 16 + 2 \times 1 = 18$$



Lorsque l'on préfixe un entier par 0, il est encodé en base 8 ! Attention à ne pas se faire avoir par exemple pour l'encodage d'une date. En effet, le code ci-dessous provoque une erreur de compilation puisque le nombre 09 n'existe pas (9 n'existe pas en base 8) !

```

1 public class Test {
2     public static void main(String[] args) {
3         System.out.println(getDate(09, 06, 1977));
4     }
5
6     public static String getDate(int jour, int mois, int année) {
7         return "" + jour + "/" + mois + "/" + année;
8     }
9 }
```

4 Réels

Il y a 2 types (qui font croire qu'ils sont des) "réels".

Nom du type	intervalle (valeurs possibles)	place mémoire occupée	valeur par défaut
float	$\pm[1,4 \times 10^{-45}, \dots, 3,4 \times 10^{38}]$	32 bits	0.0f
double	$\pm[4,9 \times 10^{-324}, \dots, 1,7 \times 10^{308}]$	64 bits	0.0d (ou 0.0)

Ces nombres sont stockés en mémoire conformément à la norme IEEE 754.



Il y a un gros problème d'intervalles : il y a plein de valeurs qui ne sont pas représentées.

Autour de zéro :

- les **float** sont utilisables pour 6 chiffres significatifs
- les **double** sont utilisables pour 15 chiffres significatifs

Ici, on utilisera systématiquement des **double**.

Note



La classe **Math** contient des opérations pour les calculs sur les **double**. La classe **Math** contient également des constantes, par exemple **Math.PI** ($\approx \pi$)

Il y a des valeurs remarquables :

- 0.0 et -0.0 existent séparément et sont égaux
- **Double.POSITIVE_INFINITY**, **Double.NEGATIVE_INFINITY** : le nombre le plus grand et le plus petit qui peut être encodé dans un double en Java.
- **NaN (Not A Number)** : résultat de la division de 0.0 par 0.0, **Math.sqrt(-1)** ou encore le reste de la division par 0².

²Dans ces cas particulier, Java ne lève pas une exception mais renvoie la *valeur NaN*

5 Caractères

Ce type n'existe pas en XML Schema, mais il est très utile en Java. En anglais : **character**.

Nom du type	intervalle (valeurs possibles)	place mémoire occupée	valeur par défaut
char	un seul caractère unicode UTF-16 (UTF-8 est inclus dans UTF-16)	16 bits	\u0000

On représente généralement un caractère par sa valeur entre guillemets simples' (*simple quote*). Note: \u permet d'utiliser le code unicode d'un caractère (exemple : `char c = '\u0381` donne le caractère 'α')



Attention à ne pas confondre un caractère et la valeur qu'il représente. par exemple `char c = 4` est le nombre 4 codé sur 16 bits, mais il ne représente **pas** le caractère '4' qui permet d'afficher 4.

De même, le caractère \u0000 est le caractère nul, mais il ne représente pas le caractère '0'

Il y a des valeurs remarquables :

- '\t': la tabulation horizontale
- '\n': aller à la ligne
- '\r': retour au début de la ligne
- ...

6 Chaîne de caractères

En XML Schema, on utilise le type prédéfini `xsd:string`. En Java, il **n'y a pas de type primitif correspondant**. En effet, un type primitif utilise un espace mémoire de taille prédéfinie / fixe. Pour une chaîne de caractères, on ne peut pas déterminer à l'avance combien il y aura de caractères. On ne peut donc pas stocker une chaîne de caractères dans un type primitif. On va être obligé de créer un espace mémoire spécifique pour chaque chaîne de caractères.

Comme le type *chaîne de caractères* est très utile dans toute application, Java a défini un type spécifique : le type **String** (Attention, il s'agit d'un S majuscule !).

Le type **String** est donc défini à l'aide d'une classe Java. On doit donc respecter le principe de construction: déclaration, instanciation, initialisation.



Il faut toujours garder à l'esprit qu'on manipule dans ce cas une *instance* à l'aide d'une *référence*.

On utilise le délimiteur " (double guillemet, *double quote*) (sur la touche 3 du clavier)

On utilise souvent les opérateurs + et += qui ont un sens particulier pour les chaînes de caractères :

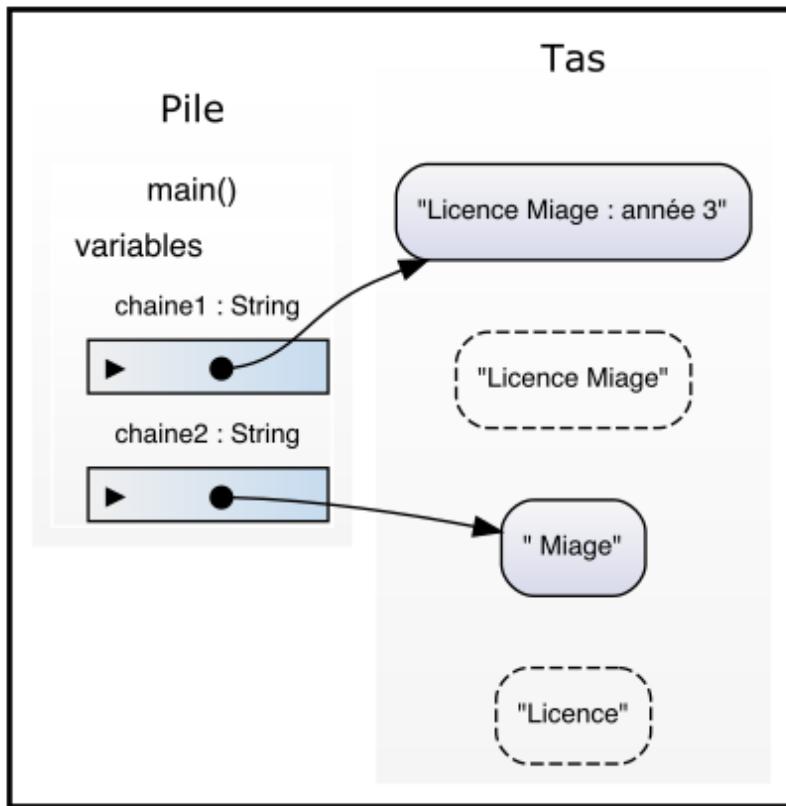
- + opération de concaténation de deux chaînes de caractères
- += opération de concaténation à la fin d'une chaîne de caractères.

A noter que les opérateurs + et += savent aussi transformer tout seul un valeur numérique ou booléenne en chaîne de caractères.

```

1 String chaine1;
2 chaine1 = new String("Licence");
3 String chaine2;
4 chaine2 = new String(" Miage");
5 chaine1 += chaine2; // équivalent de : chaine1 = chaine1 + chaine2
6 chaine1 += " : année " + 3; // la concaténation avec l'int 3
7 // 1- transforme 3 en "3"
8 // 2- concatène "3" avec ce qu'il y avait déjà dans chaine1

```



Comme `String` est une classe très usuelle, il existe de nombreuses méthodes qui permettent de simplifier l'utilisation de chaîne de caractères :

- On n'est pas obligé d'utiliser l'opérateur `new`: on peut écrire directement `chaine1 = "Technologies";` (équivalent de `chaine1 = new String("Technologies");`)
- On peut connaître la taille d'une chaîne de caractères (c'est-à-dire le nombre de caractères) en utilisant la méthode `length()`: `chaine1.length()` renvoie un `int` qui donne la taille de la chaîne.

Pour afficher une chaîne de caractères sur la sortie standard, on utilise `System.out.println(...)`

```

1 System.out.println("test");
2 // 1. création de l'instance "test"
3 // et 2. envoi de la référence à cette instance à sout
4 System.out.println(chaine1);
5 // 1. envoi de la référence à l'instance "Technologies de l'Information pour la
   Santé3" à sout
6 System.out.println("Technologies" + chaine2);
7 // 1. création d'une instance
8 // 2. envoi de cette nouvelle instance à sout
9 int i = 42;
10 System.out.println(i);
11 // 1. création de l'instance "42"
12 // 2. envoi cette nouvelle instance à sout

```

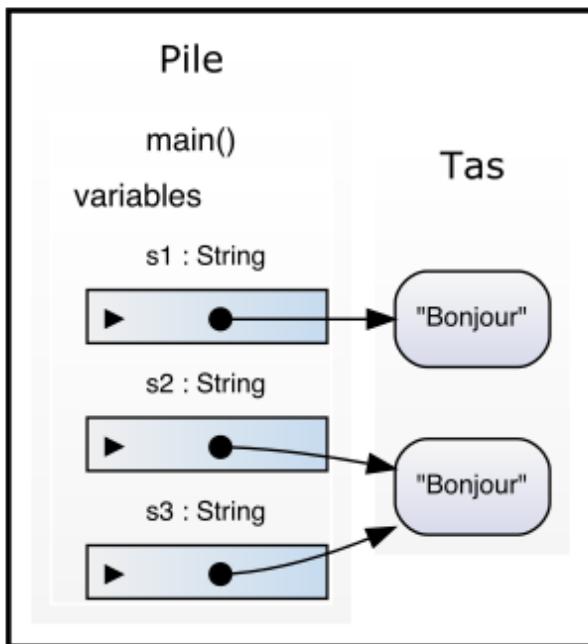


Attention cependant à toujours garder en mémoire le fait que les chaînes de caractères sont des types références et non des types primitifs. Notamment pour la comparaison entre 2 chaînes de caractères:

```

1 String s1 = new String("Bonjour");
2 String s2 = new String("Bonjour");
3 String s3 = s2;
4
5 boolean test1 = (s1 == s2);
6 System.out.println("s1 == s2 ? : " + test1);
// La ligne précédente affichera s1 == s2 ? : false
7 boolean test2 = (s2 == s3);
8 System.out.println("s2 == s3 ?: " + test2);
// La ligne précédente affichera s2 == s3 ?: true
10

```



Dans le code précédent, la variable booléenne `test1` compare les valeurs des 2 *références* `s1` et `s2`. Comme ces 2 références ne pointent pas vers la même instance, on a `s1 != s2`.

Par contre, `s2` et `s3` référencent bien la même instance et lorsque l'on compare ces 2 références, on obtient une égalité. Si l'on veut comparer le contenu des chaînes de caractères, on utilisera la méthode `equals` sur l'une des deux chaînes de caractères. En effet, `s1.equals(s2)` renverra `true`.



Lorsque l'on souhaite comparer 2 chaînes de caractères (ou 2 instances en général), on **n' utilisera jamais** la comparaison `==`, mais on **utilisera systématiquement** la méthode `equals`.



L'initialisateur par défaut (`String s1 = "Bonjour";` au lieu de `String s1 = new String("Bonjour");`) peut induire des erreurs d'interprétation sur la manière dont fonctionnent les `String` en Java, car les chaînes initialisées avec cet initialisateur par défaut sont traitées un peu différemment par le compilateur Java.

En effet, si dans un même code, plusieurs chaînes sont initialisées par défaut avec la même valeur, le compilateur ne créera qu'un objet `String` (cf code ci-dessous).

Cependant, comme dans les méthodes, on ne peut pas présupposer de l'initialisation des paramètres, on utilisera systématiquement la méthode `equals` pour les chaînes de caractères.

```

1 String s1 = "bonjour";
2 String s2 = "bonjour";
3 boolean test = s1 == s2;
4 System.out.println("s1 == s2 ? : " + test);
5 // La ligne précédente affichera s1 == s2 ? : true
6 // Parce que l'initialisateur par défaut de String n'aura créé qu'une seule
   instance.

```

Note

Depuis la version 9 de Java (JDK 9), on peut utiliser des chaînes de caractères comme comparateur de la structure algorithmique `switch`.

7 Conversions automatiques entre types primitifs

Java permet parfois de convertir les types de données entre eux. Par exemple:

```

1 boolean test = true;
2 char c = '2';
3 int i = 3;
4 long l = 499784L;
5 float f = 4.0f;
6 double d = 8.5;

7
8 test = c; // Conversion impossible, provoque une erreur à la compilation
9 c = test; // Conversion impossible, provoque une erreur à la compilation

10
11 i = c; // Ok, c de type char a été converti implicitement en int
12 l = i; // Ok, i de type int a été converti implicitement in long
13 f = l; // Ok, l de type long a été converti implicitement en float
14 d = f; // Ok, f de type float a été converti implicitement en double

15
16 i = l; // Attention, cette ligne provoque une erreur à la compilation
17 // En effet, l de type long n'est pas converti implicitement en int (risque de
   perte d'information).
18 // Mais on peut utiliser la conversion explicite.
19 i = (int) l;

20
21 // De même entre les double et float
22 f = d; // Provoque une erreur à la compilation
23 f = (float) d; // utilisation de la conversion explicite

```

Convertir en ->	boolean	char	int	long	float	double
boolean	=	Impossible	Impossible	Impossible	Impossible	Impossible
char	Impossible	=	Implicite	Implicite	Implicite	Implicite
int	Impossible	Explicite	=	Implicite	Implicite	Implicite
long	Impossible	Explicite	Explicite	=	Implicite	Implicite
float	Impossible	Explicite	Explicite	Explicite	=	Implicite
double	Impossible	Explicite	Explicite	Explicite	Explicite	=

8 Autres opérateurs

8.a Comparaison

Pour comparer des **valeurs** de types primitifs (attention à ne pas confondre type primitif et type prédefini), on peut utiliser les opérateurs suivants: `<`, `>`, `<=`, `>=`, `==`, `!=`.

Attention à ne pas confondre `a = b` et `a == b` !



- `a = b`: **affectation** de la valeur qui est dans la case mémoire `b` à la case mémoire `a`, c'est-à-dire que la case `a` prend la valeur de `b`.
- `a == b`: **comparaison** de la valeur qui est dans la case mémoire `b` à la valeur qui est dans la case mémoire `a`.

Les opérateurs de comparaison renvoient toujours une valeur de type `boolean` (`true` et `false`).

8.b Affectations

Pour les affectations, on peut utiliser l'opérateur binaire `=` ou les opérateurs unaires `+=`, `*=`, `-=`, `/=`.

Les opérateurs d'affectation renvoient la valeur affectée, ils sont donc utilisables dans une expression, mais cela peut souvent nuire à la lisibilité du programme (de plus, ils font partie des rares opérateurs associatifs à droite).

```

1 int a = 1;
2 int b = 2;
3 int c;
4
5 c = b += a; // Avec l'associativité, cela s'interprète comme c = (b += a);
6 // ici : a == 1, b == 3 et c == 3. En effet, (b += a) affecte 3 à b
7 // et renvoie 3 qui est affecté à c.
```

Les Exceptions en Java

Imaginez que vous cuisinez en suivant une recette. Soudain, vous réalisez que vous n'avez plus d'œufs. C'est un problème inattendu, non ? En Java, on appelle les problèmes inattendus des *exceptions*.

Définition



Une **exception** est un évènement qui se produit pendant l'exécution d'un programme et perturbe le flux normal des instructions.
En Java, les exceptions sont des objets qui représentent ces situations imprévues. Elles sont comme de petits alarmes qui se déclenchent lorsque quelque chose ne va pas dans votre code.

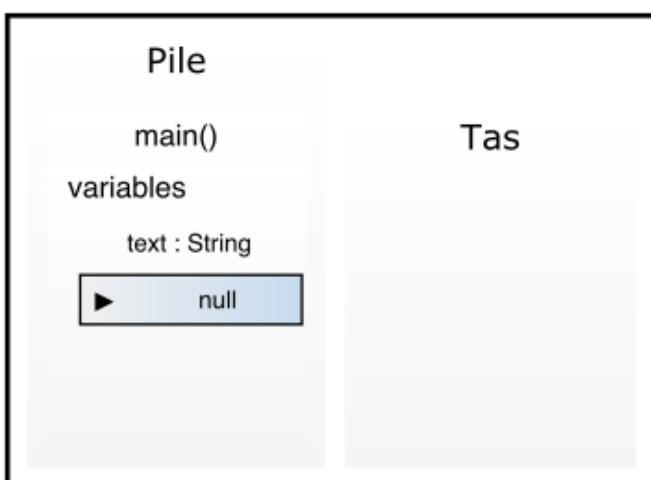
Note: *C'est la manière de Java de dire : "Oups ! Quelque chose ne s'est pas passé comme prévu !"¹*

Par exemple si l'on considère la classe suivante :

```

1 public class ExempleException {
2     public static void main(String[] args) {
3         String text = null;
4         System.out.println(text.length());
5     }
6 }
```

Si l'on dessine le diagramme APO de ce qui se passe dans le code ci-dessus (cf chapitre 6), dans la mémoire, on aura :



¹Ce cours est largement inspiré de ce très bon site https://w3schools.tech/fr/tutorial/java/java_exceptions

Et à l'exécution, on aura l'erreur suivante :

```

1 > javac ExempleException.java
2 > java ExempleException
3 Exception in thread "main" java.lang.NullPointerException: Cannot invoke "
   String.length()" because "<local1>" is null
4         at ExempleException.main(ExempleException.java:4)

```



Comment lire cette erreur ?

Tout d'abord, on remarque que l'erreur apparaît après la deuxième ligne de commande. La première ligne de commande, `javac ExempleException.java` est la ligne de compilation, et ne produit **pas** d'erreur. C'est la ligne `java ExempleException` qui produit une erreur, c'est-à-dire l'exécution par la JVM. Une exception est toujours une **erreur d'exécution**.

L'erreur est une `NullPointerException` c'est-à-dire que l'on essaie de lire un attribut ou d'appliquer une méthode à une *instance* qui n'existe pas ! L'exception indique la ligne de code qui provoque l'erreur ici `ExempleException.java:4` c'est-à-dire la ligne 4 du fichier `ExempleException.java`, soit l'instruction `System.out.println(text.length());`.

En effet, on remarque que dans le diagramme APO, que la variable `text` de type référence vers une `String` ne pointe vers aucune instance de chaîne de caractère. L'instruction `text.length()` ne peut donc pas être exécutée.

Note



Il sera dit et répété plusieurs fois dans cette UE qu'une bonne pratique de programmation est :

une seule instruction par ligne.

Il est facile ici de comprendre que cela facilite le débogage des exceptions.

Les exceptions peuvent se produire pour de nombreuses raisons. Voici quelques-unes des raisons courantes:

1. Entrée utilisateur invalide
2. Panne matérielle
3. Problèmes de réseau
4. Erreurs de programmation

1 Exceptions Intégrées

Java prédéfinit une variété d'exceptions, organisées en hiérarchie (cette hiérarchie est matérialisée par de l'héritage que vous verrez au chapitre 16). Au sommet de cet arbre de famille se trouve la classe `Throwable` (qui signifie *lancable*) qui a 2 enfants principaux: `Error` et `Exception`.

1.a Erros

Les erreurs sont des problèmes sérieux qui se produisent généralement à cause de problèmes en dehors du contrôle de votre programme. Par exemple, si votre ordinateur manque de mémoire, Java pourrait lancer une `OutOfMemoryError`. En tant que programmeur, vous ne tentez généralement pas d'attraper ou de gérer les erreurs.

1.b Exceptions

Les Exceptions, elles, sont des problèmes qui **doivent** être anticipés et gérés dans votre code.



En tant que programmeur, vous devez, certes, anticiper et gérer les futures erreurs / exceptions. **MAIS** il faut toujours penser / décrire / coder le cas normal (c'est-à-dire celui où tout se passe bien) **AVANT** de penser / décrire / coder la gestion des erreurs / exceptions.

Sur le plan technique, on considère 3 types principaux d'exceptions en Java:

1. les Exceptions vérifiées (*Checked Exceptions*)
2. les Exceptions non vérifiées (*Runtime Exceptions*)
3. les Erreurs

1.c Les Exceptions non vérifiées (*Runtime Exceptions*)

Les *Runtime Exceptions* se produisent à l'exécution des méthodes prédéfinies en Java et n'ont pas besoin d'être explicitement gérées ou déclarées. Elles sont généralement causées par des erreurs de programmation comme dans l'exemple du code de `ExempleException.java`. La gestion de ces exceptions se fait dans la phase de débogage et de tests.

Des exemples courant de *Runtime Exceptions* lancées suite à des erreurs de programmation courantes sont par exemple:

- *ArithmaticException* en cas de division par zéro
- *IndexOutOfBoundsException* lorsqu'un indice sort d'un tableau
- *NullPointerException* lorsque l'on essaie de lire un attribut ou appliquer une méthode à une instance qui n'existe pas...



Pour débuger ces erreurs de programmation, le diagramme APO est un outil redoutablement efficace !

1.d Les Exceptions vérifiées

Ce sont des exceptions que vous **devez** gérer dans votre programme. En effet, le compilateur vérifiera que vous gérez ces exceptions et vous obtiendrez une erreur à la compilation si ce n'est pas le cas. La section suivante explique les différentes façons de gérer ces exceptions. En dehors des exceptions que vous créerez (qui seront d'office des *Checked Exceptions*), les exceptions vérifiées les plus courantes sont

- *FileNotFoundException* dans le cas de la tentative d'ouverture d'un fichier qui n'existe pas
- *ClassNotFoundException* dans le cas où vous utilisez une classe qui n'a pas été définie ou qui n'a pas pu être compilée.

2 La gestion des Exceptions

Comme vu plus haut, les exceptions sont des objets qui représentent des situations inattendues. Lorsqu'une exception se produit, elle est *lancée* (*throw* en anglais) par une méthode qui a été appelée. Si elle n'est pas *attrapée* (*catch* en anglais), soit :

- le compilateur bloque la compilation, s'il s'agit d'une *Checked Exception*
- le programme *plante* à l'exécution (ce qui produit une expérience utilisateur fort désagréable !)

Note



Nous verrons dans la section 2.b que l'on peut *gérer* les exceptions sans forcément les *attraper* partout.

2.a Le bloc *try-catch*

Le bloc *try-catch* est constitué de la manière suivante :

```

1 try {
2     // Code qui pourrait provoquer une exception
3 } catch( Déclaration de l'exception attrapée ) {
4     // Code pour gérer l'exception
5 }
```

Considérons l'exemple suivant :

```

1 import java.util.InputMismatchException;
2 import java.util.NoSuchElementException;
3 import java.util.Scanner;
4
5 public class DivisionCalculator {
6     public static void main(String[] args) {
7         Scanner scanner = new Scanner(System.in);
8
9         try {
10             System.out.print("Entrez le numérateur : ");
11             int numérateur = scanner.nextInt();
12
13             System.out.print("Entrez le dénominateur : ");
14             int dénominateur = scanner.nextInt();
15
16             int résultat = numérateur / dénominateur;
17             System.out.println("Résultat : " + résultat);
18         } catch(InputMismatchException e) {
19             System.out.println("Un des éléments entré n'est pas un entier.");
20         } catch(NoSuchElementException | IllegalStateException e) {
21             System.out.println("Rien n'a été entr ou le scanner a été fermé et
22 ne peut rien lire.");
23             System.out.println(e.getMessage());
24         } catch(ArithmetricException e) {
25             System.out.println("Erreur : Impossible de diviser par zéro !");
26         } finally {
27             scanner.close();
28         }
29     }
}
```

le bloc *try*

Les instructions du bloc *try* sont susceptibles de lancer des exceptions. L'objet `scanner` initialisé ligne 7 permet de lire du texte depuis l'entrée standard (la ligne de commande).

Les instructions `scanner.nextInt()` lignes 11 et 14 sont en effet susceptibles de renvoyer les exceptions de types suivants : ² :

- `InputMismatchException` - si l'élément entré n'est pas un entier
- `NoSuchElementException` - si l'entrée est vide
- `IllegalStateException` - si le `scanner` est fermé.

Enfin, l'instruction `numérateur / dénominateur` ligne 16 est susceptible de lancer une exception de type `ArithmetricException` en cas de division par zéro.

²cf. javadoc: <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html#nextInt-->

Note

Dans cet exemple, toutes les exceptions (`InputMismatchException`, `NoSuchElementException`, `IllegalStateException` et `ArithmetricException`) sont de type *Runtime Exception*. Ceci fait que l'on peut compiler ce code *sans* le bloc *try-catch*. Par exemple, si l'on commente les parties *try-catch*, on n'obtient pas d'erreur à la compilation, mais on peut se retrouver avec un plantage comme ci-dessous.

```

1 > javac DivisionCalculator.java
2 > java DivisionCalculator
3 Entrer le numérateur : bonjour
4 Exception in thread "main" java.util.InputMismatchException
5     at java.base/java.util.Scanner.throwFor(Scanner.java:939)
6     at java.base/java.util.Scanner.next(Scanner.java:1594)
7     at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
8     at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
9     at DivisionCalculator.main(DivisionCalculator.java:11)
```

les blocs *catch*

Le premier bloc *catch*, à la ligne 18-19, permet de récupérer les potentielles exceptions de type `InputMismatchException`. L'exception est déclarée dans les parenthèses du *catch* et peut être utilisée dans le bloc. Ici, la gestion de cette exception consiste à écrire un message à l'utilisateur.

Le deuxième bloc *catch*, lignes 20 à 22 permet de récupérer 2 types d'exception : les `NoSuchElementException` et `IllegalStateException` (notez le `|` pour signifier le *ou bien* entre les 2 types). Ici, on imprime un message personnalisé, puis on imprime également le message de l'exception attrapée.

Le troisième bloc *catch* lignes 23-24 permet de gérer les divisions par zéro.

le bloc *finally*

Le bloc *finally* contient du code qui sera exécuté **quoi qu'il arrive**, c'est-à-dire indépendamment du fait qu'une exception ait été attrapée ou non.

Dans l'exemple, avant le bloc *try*, un `scanner` a été instancié et, par définition de son instantiation, ouvert. En sortant du programme, qu'il y ait eu une erreur ou non, il faut fermer ce scanner. C'est ce que fait l'instruction ligne 26.

2.b La propagation des exceptions

Si une exception se produit dans une méthode, elle crée un objet de type `Exception` et le lance. La méthode qui a appelé cette méthode cherche alors un gestionnaire d'exception approprié (*try-catch*). Si elle n'en trouve pas, elle lance alors à son tour cette exception à la méthode qui l'a appelée. Ce processus continue vers le haut de la pile d'exécution jusqu'à ce qu'un gestionnaire soit trouvé ou que l'exception atteigne la méthode `main`.

Par exemple :

```

1 public class ExceptionPropagationDemo {
2     public static void main(String[] args) {
3         try {
4             int res = method1();
5         } catch (Exception e) {
6             System.out.println("Exception attrapée dans main : " + e.getMessage());
7         }
8     }
9
10    static int method1() {
11        return method2();
12    }
13}
```

```

14     static int method2() {
15         return method3();
16     }
17
18     static int method3() {
19         int resultat = 12 / 0;
20         return resultat;
21     }
22 }
```

Dans cet exemple, la méthode `main` appelle la méthode `method1()` qui elle-même appelle la méthode `method2()`, qui elle-même appelle la méthode `method3()`. La méthode `method3()` lance une `ArithmException` (par la division par zéro). Cette exception n'est pas attrapée dans la méthode `method2()`, elle est donc propagée, c'est-à-dire renvoyée par la méthode `method2()`. De la même manière elle n'est pas attrapée par la méthode `method1()` qui la renvoie à son tour. L'exception est donc gérée par la méthode `main` dans le *try-catch*.



L'exception `ArithmException` est une *Runtime Exception*, donc les méthodes `method1()`, `method2()` et `method3()` qui la lancent tour à tour n'ont pas besoin de déclarer qu'elles la lancent. Lorsqu'il s'agira de *Checked Exceptions*, il faudra déclarer, dans la signature de chacune des méthodes `method1()`, `method2()` et `method3()` `throws TypeDeLException`.

3 Créer ses propres Exceptions

Lorsque vous programmez, vous devez gérer les exceptions natives de Java, mais vous devez également prévoir les erreurs qui pourraient se produire dans vos propres programmes. En effet, que se passe-t-il si les entrées / paramètres ne sont pas dans les plages de valeur attendues ? Si une opération est impossible ? etc.



Rappel : En tant que programmeur, vous devez, certes, anticiper et gérer les futures erreurs / exceptions. **MAIS** il faut toujours penser / décrire / coder le cas normal (c'est-à-dire celui où tout se passe bien) **AVANT** de penser / décrire / coder la gestion des erreurs / exceptions.

3.a Créer un type d'Exception personnalisé

Comme expliqué plus haut, une exception est une instance / un objet de type `Exception`. Pour une gestion des exceptions efficace, il faut que le type des exceptions soit différentié selon l'exception (et non pas seulement le message d'erreur). Cela permet de faire des traitements différents en fonction de chaque type, via des `catch` (`TypeDeLException e`), pour chaque type d'exception.

Lorsque vous programmez, vous souhaitez donc créer des types d'exception spécifique à votre code. Pour cela, on utilise le principe de l'héritage. Ce principe sera vu au chapitre 16, mais vous pouvez d'ors et déjà créer vos propres exceptions en vous basant sur l'exemple ci-dessous :

```

1 public class MonProblemeException extends Exception {
2     public MonProblemeException(String message) {
3         super(message);
4     }
5 }
```

On remarque :

- On crée une nouvelle classe (i.e. dans un fichier qui porte le même nom). Remarquons que son nom se termine par `Exception`. C'est une bonne pratique de nommer les exceptions par `NomSpécifiqueException`.

3. CRÉER SES PROPRES EXCEPTIONS

55

- Notre classe `MonProblemeException` hérite (mot clé `extends`) de la classe `Exception`
- Il faut créer un constructeur qui prend en paramètre une chaîne de caractères.
- Ce constructeur appelle le constructeur de la superclasse en utilisant `super(message)`.

3.b Lancer une exception

Définition



Le mot-clé `throw` est utilisé pour lancer une exception. Il est systématiquement suivi d'une instance d'exception (souvent `throw new TypeException("message d'erreur")`).

Définition



Le mot-clé `throws` est utilisé dans les déclarations de méthodes pour indiquer que cette méthode peut lever un ou plusieurs types d'exceptions.

Note : Il doit être mis dans les méthodes qui appellent des méthodes qui génèrent des exceptions mais qui ne les gèrent pas à l'aide du bloc *try-catch*.

Note : Il n'est pas obligatoire pour les méthodes qui lèvent des *Runtime Exceptions* (mais cela reste une bonne pratique de les déclarer).

La syntaxe est illustrée dans l'exemple ci-dessous.

```
1 class AgeException extends Exception {  
2     public AgeException(String message) {  
3         super(message);  
4     }  
5 }  
6  
7 public class SystemeDeVote {  
8     public static void vérifierÉligibilité(int age) throws AgeException {  
9         if (age < 18) {  
10             throw new AgeException("Vous devez avoir 18 ans ou plus pour voter.  
11                     ");  
12         } else {  
13             System.out.println("Vous êtes éligibles pour voter !");  
14         }  
15     }  
16     public static void choisirCandidat() {  
17         // Permet de choisir anonymement un(e) candidat(e)  
18         System.out.println("A voté !");  
19     }  
20  
21     public static void main(String[] args) {  
22         try{  
23             vérifierÉligibilité(20);  
24             choisirCandidat();  
25             vérifierÉligibilité(15);  
26             choisirCandidat();  
27         } catch (AgeException e){  
28             System.out.println("Le vote n'a pas pu avoir lieu: " + e.getMessage  
29                         ());  
30         }  
31     }  
}
```

qui donne le résultat suivant à l'exécution :

```
1 > javac SystemeDeVote
2 > java SystemeDeVote
3 Vous êtes éligibles pour voter !
4 A voté !
5 Le vote n'a pas pu avoir lieu: Vous devez avoir 18 ans ou plus pour voter.
```

Note



Le deuxième vote n'a pas eu lieu (étant donné que l'âge requis n'était pas vérifié).



Comme l'exception a été gérée, le programme n'a pas planté (bien que le deuxième vote n'ait pas eu lieu) et a terminé normalement.

Part B

Rappel des Structures Algorithmiques

Instructions conditionnelles

Fiche

<u>Prérequis</u>	Chapitre 6 : Types prédéfinis en Java
<u>Objectif pédagogique</u>	Acquérir les bonnes pratiques d'utilisation des instructions conditionnelles.
<u>Intérêt</u>	<ul style="list-style-type: none"> • Ecrire des instructions conditionnelles avec un risque minimum d'erreurs • Ecrire des instructions conditionnelles qui peuvent être relues facilement
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> • Ecrire des instructions conditionnelles avec un risque minimum d'erreurs • Ecrire des instructions conditionnelles qui peuvent être relues facilement

1 Introduction

On se situe au niveau du code d'une méthode. C'est dans les méthodes que l'on va effectuer les traitements avec les données disponibles (attributs, paramètres, variables...). Dans une méthode il est parfois nécessaire de réaliser un choix entre :

- deux calculs
- deux suites d'instructions à réaliser

Exemple : selon la valeur du discriminant, la racine d'une équation du second degré est possible ou non.

2 Principe algorithmique

2.a Instruction conditionnelle

```

1 si (condition) alors
2   début // nouveau bloc d'instruction
3     // bloc contenant les instructions à exécuter lorsque la condition est
4       vraie
5     ...
6   fin

```

(condition) doit être exprimée par une expression booléenne (composée potentiellement d'opérateur ou bien, et puis, négation, tests sur des valeurs...).

2.b En Java

```

1 if (condition) {
2   // bloc contenant les instructions à exécuter lorsque la condition est
3   vraie
4   ...
5 }

```

Par exemple

```

1 public class Test
2 {
3   public static void main(String[] args) {
4     int variable = 20;
5     // Traitement où l'on peut éventuellement changer la variable 'variable'
6     //
7     //
8     if(variable == 20) {
9       System.out.println("La variable 'variable' est bien égale à 20");
10    }
11  }

```

condition est une expression de type boolean



Note

le plus difficile est d'écrire la bonne expression boolean



Note

une instruction conditionnelle peut être utilisée dans n'importe quel bloc, y compris, un bloc d'instructions à l'intérieur d'un autre bloc d'instructions, lui-même à l'intérieur d'un autre bloc d'instructions, lui même à l'intérieur d'un autre bloc d'instructions,

Pour des raisons de lisibilité et de maintenabilité du code, à partir de deux blocs imbriqués, il vaut mieux créer des méthodes intermédiaires.



3 Instruction conditionnelle à deux branches

Il est parfois nécessaire de faire un traitement T1 lorsque l'expression booléenne est vérifiée et un traitement T2 dans le cas contraire (lorsque l'expression booléenne n'est pas vraie).

3.a En algorithmique

```

1 si (condition) alors
2   début
3     // T1
4   fin
5 sinon
6   début
7     // T2
8   fin

```

3.b En Java

```

1 if (condition) {
2   // T1
3 }
4 else {
5   // T2
6 }

```

Par exemple:

```

1 public class Test
2 {
3   public static void main(String[] args) {
4     int variable = 20;
5     // Traitement où l'on peut éventuellement changer la variable 'variable'
6     //
7     ...
8     if(variable == 20) {
9       System.out.println("La variable 'variable' est bien égale à 20");
10    } else {
11      System.out.println("La variable 'variable' n'est pas égale à 20");
12    }
13  }
14 }

```

Note



A l'intérieur d'un bloc `if` ou d'un bloc `else`, on peut avoir un autre bloc `if`

Par exemple:

```

1 public class Test
2 {
3   public static void main(String[] args) {
4     int variable = 20;
5     // Traitement où l'on peut éventuellement changer la variable 'variable'
6     //
7     ...
8     if(variable == 20) {
9       System.out.println("La variable 'variable' est bien égale à 20");
10    } else {

```

```

11     if (variable == 15) {
12         System.out.println("La variable 'variable' n'est pas égale à
13             20, mais est égale à 15");
14     } else {
15         if (variable == 10) {
16             System.out.println("La variable 'variable' n'est pas égale à
17                 20 ni à 15, mais est égale à 10");
18         } else {
19             System.out.println("La variable 'variable' n'est égale ni à
20                 20, ni à 15, ni à 10");
21         }
22     }
23 }
```

Note

Dans le cas où les conditions imbriquées dépendent des même variables que le **if** initial, il vaut mieux les mettre dans les blocs **else** pour ne pas retester les mêmes conditions plusieurs fois.

Par exemple, le code suivant est faux et ne fonctionne pas:

```

1 public class Test
2 {
3     public static void main(String[] args) {
4         int variable = 20;
5         // Traitement ou l'on peut éventuellement changer la variable 'variable'
6         //
7         ...
8
9         if(variable == 20) {
10             System.out.println("La variable 'variable' est bien égale à 20");
11         }
12         if (variable == 15) {
13             System.out.println("La variable 'variable' n'est pas égale à 20,
14                 mais est égale à 15");
15         }
16         if (variable == 10) {
17             System.out.println("La variable 'variable' n'est pas égale à 20 ni
18                 à 15, mais est égale à 10");
19         } else {
20             System.out.println("La variable 'variable' n'est égale ni à
21                 20, ni à 15, ni à 10");
22         }
23 }
```

On peut noter que non seulement que ce code est faux puisque l'on effectue des tests non nécessaires, mais également qu'il ne fonctionne pas !

En effet, mettons que la variable **variable** reste égale à 20. On effectue d'abord le premier test et l'on écrit sur la sortie standard: La variable 'variable' est bien égale à 20. Pour l'instant, c'est juste... Mais ensuite, on effectue le test inutile **if (variable == 15)**. En effet, ce test est inutile puisque l'on sait déjà que **variable** est égale à 20. Il est donc *faux* de ne pas mettre ce **if** dans le bloc **else**. Par ailleurs, ce code porte à confusion. En effet, on effectue ensuite le test inutile **if (variable == 10)** (même remarque que précédemment). Il faut alors bien se souvenir que le **else** qui suit ce bloc porte uniquement sur la condition **if (variable == 10)**. Avec **variable == 20**, on entre donc dans ce bloc **else** et l'on écrit sur la sortie standard La variable 'variable' n'est égale ni à 20, ni à 15, ni à 10... Ce qui est faux...

Au lieu des blocs `if` et `else` imbriqués, on peut utiliser l'instruction `else if`. Le bloc `else` est alors exécuté uniquement si les conditions de tous les `if` et `else if` précédents sont fausses.

4.a En algorithmique

```
1 si (condition) alors
2     début
3         // T1
4     fin
5 sinon si (condition2) alors
6     début
7         // T2
8     fin
9 sinon
10    début
11        // T3
12    fin
```

4.b En Java

```
1     if (condition) {  
2         // T1  
3     }  
4     else if (condition2) {  
5         // T2  
6     }  
7     else {  
8         // T3  
9     }
```

T3 n'est exécuté que si condition et condition2 sont fausses.

Notre exemple précédent peut être ré-écrit de manière juste comme suit:

```
1 public class Test
2 {
3     public static void main(String[] args) {
4         int variable = 20;
5         // Traitement où l'on peut éventuellement changer la variable 'variable'
6         //
7         ...
8
9         if(variable == 20) {
10             System.out.println("La variable 'variable' est bien égale à 20");
11         }
12         else if (variable == 15) {
13             System.out.println("La variable 'variable' n'est pas égale à 20,
14                             mais est égale à 15");
15         }
16         else if (variable == 10) {
17             System.out.println("La variable 'variable' n'est pas égale à 20 ni
18                             à 15, mais est égale à 10");
19         } else {
20             System.out.println("La variable 'variable' n'est égale ni à
21                             20, ni à 15, ni à 10");
22         }
23     }
24 }
```

5 Instruction à choix multiple

On a parfois (rarement) besoin d'écrire un algorithme où le traitement dépend de la *valeur* prise par une variable non booléenne. Selon cette valeur (qui doit être prise dans un ensemble prédéfini, par exemple pour un `enum`¹), le traitement diffère.

On utilisera alors l'instruction algorithmique à choix multiple, parfois appelée :

- branchement à choix multiple / plusieurs choix
- aiguillage (en anglais `switch`)
- instruction "selon"

5.a En algorithmique

```

1 Selon la valeur de (variable)
2   début
3     cas valeur1:
4       // Traitement pour valeur1
5       fin
6     cas valeur2:
7       // Traitement pour valeur2
8       fin
9     ...
10    pour tout autre valeur:
11      // Traitement "par défaut"
12      fin
13  fin

```

5.b En Java

```

1 switch (variable) {
2   case valeur1:
3     // Traitement pour valeur1
4     break;
5   case valeur2:
6     // Traitement pour valeur2
7     break;
8   ...
9   default:
10     // Traitement "par défaut"
11     break;
12 }

```



`break` = interruption brutale de l'algorithme / du bloc courant. Le **seul endroit** où on peut utiliser l'instruction `break` est pour terminer un `case`.

5.c Types utilisables pour la variable

`variable` peut être de type :

```

1 Selon la valeur (version de Java)
2   début
3     cas Java 1.4:
4       - int, short, byte, char
5       fin

```

¹ voir chapitre 9

```
6      cas Java 1.5: // Java 1.5 = Java 5
7      cas Java 5:
8      cas Java 6:
9          - int, short, byte, char, enum
10         fin
11     cas Java 7:
12     cas Java 8:
13     cas Java 9 et supérieures:
14         - int, short, byte, char, enum, String
15 fin
```

5.d Valeurs utilisables

Dans la partie `case` on peut utiliser **que** des valeurs **prédeterminées** au moment de la compilation. C'est-à-dire que l'on ne peut pas faire de calculs ou utiliser d'expressions arithmétiques.

```
1 ...
2     case i+1:
3         ... // interdit : calcul
4 ...
5     case getValue():
6         ... // interdit : appel de méthode
7 ...
8     case "TIS3":
9         ... // Ok pour Java 7, 8 et 9
10 ...
11    case INTERNE:
12        ... // Ok pour les enum pour Java 5 et suivant
```


Constantes et énumérations

Fiche

<u>Prérequis</u>	Chapitre 3 : Classe et Instance, Chapitre 6 : Types prédéfinis en Java
<u>Objectif pédagogique</u>	Acquérir les bonnes pratiques pour l'utilisation des constantes.
<u>Intérêt</u>	<ul style="list-style-type: none"> Maîtriser l'utilisation des constantes et énumérations en Java
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> Maîtriser l'utilisation des constantes et énumérations en Java Ne jamais écrire un nombre ou un lien <i>en dur</i> dans du code Java

1 Les constantes

Définition



De manière générale, une **constante** est une donnée dont la valeur est *fixée* une fois pour toute par le programmeur et **ne peut plus** être modifiée au cours de l'exécution du programme.

En POO, les constantes sont forcément des *attributs* d'une **classe**.

En Java, pour déclarer qu'un attribut est *constant* on utilise le modificateur d'accès **final**.

Par définition, une constante est **unique** dans tout le système d'information. Nous avons vu qu'un attribut permettait de modéliser l'état d'une instance : chaque instance a donc ses attributs pouvant prendre des valeurs différentes selon l'instance (les valeurs des attributs sont propre à chaque instance). La notion d'attribut dans une instance est donc *contraire* à la notion de constante.

Pour faire en sorte de n'avoir **qu'une seule copie** de la constante dans tout le SI¹, on va utiliser la notion d'attribut de classe. Un attribut de classe est associé directement à la classe dans laquelle il est déclaré (aucune instance ne possédera cet attribut).

Pour lier un attribut directement à la classe on utilise le modificateur d'accès **static**².

Une constante est donc un attribut qui a une valeur fixée une fois pour toutes et un attribut qui est unique quelque soit le nombre d'instances. Une constante est un attribut qui est à la fois **final** (une valeur fixée une fois pour toute) et **static** (un attribut unique quelque soit le nombre d'instances).

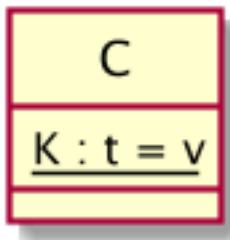
¹Système d'Information

²cette notion de **static** sera revue en partie ??

Ainsi une constante en Java se déclare comme un attribut avec en plus les modificateurs d'accès `static final`.

1.a En UML

Déclaration d'une constante K de type t et qui vaut v.



1.b En Java

```

1 class C {
2     public static final t K = v;
3 }
```

1.c Accès à une constante

Pour accéder à une constante, il n'est pas nécessaire d'avoir une instance. On utilise le nom de la classe, l'opérateur `.` et le nom de la constante.

```

1 // accès à la constante directement par la classe
2 System.out.println("Valeur de K = " + C.K); // C == nom de la classe
```

1.d Exemple de constantes dans le langage Java

Dans la classe `Math` (dans le package `java.lang`) il y a deux constantes :

- PI: la valeur de `double` qui est la plus proche de π , le rapport de la circonférence d'un cercle sur son diamètre.
- E: la valeur de `double` qui est la plus proche de e , la base du logarithme.

```

1 // accès à la constante directement par la classe
2 System.out.println("Valeur de PI = " + Math.PI);
```

1.e Convention de nommage

Le nom des constantes s'écrivent en MAJUSCULES. On utilise le caractère souligné `_` pour séparer les mots. Par exemple en partie `??`, on verra la constante `Double.MAX_VALUE` de la classe `Double`.

2 Énumérations

Lorsqu'on doit modéliser un `simpleType` représentant une énumération, c'est-à-dire un type qui limite l'ensemble des valeurs possibles à une liste prédéfinie de valeurs, on utilise une énumération (au sens de la POO).

En Java, les énumérations sont disponibles depuis la version 5. On utilise le mot clé `enum` à la place du mot clé `class`.

2.a Exemple (très simplifié)

On considère les catégories de personnels dans un hôpital : aide-soignant.e, infirmier.ère, interne en médecine et médecin. On utilise une énumération nommée **Personnel**

En UML



En Java

```

1 // dans le fichier Personnel.java
2 enum Personnel {
3     AIDE_SOIGNANT,      // noms des constantes séparées par une virgule
4     INFIRMIERE,
5     INTERNE,
6     MEDECIN;           // et terminé par point-virgule
7 }
```

Personnel est un nouveau type de données très spécial :

- il peut être utilisé comme tout type de données lors d'une déclaration d'une variable, d'un attribut ou d'un paramètre.
- pour manipuler une valeur de l'énumération, on utilise une **référence**.
- Une données de type **Personnel** ne peut prendre qu'une des quatre valeurs définis dans l'énum.
- Les valeurs définies se comportent comme des constantes, ce sont en quelque sorte des attributs de classe, et de ce fait elles sont uniques dans tout le SI.

Exemple d'utilisation :

```

1 Personnel p;
2 p = Personnel.INTERNE; // pas d'instanciation, on utilise une des valeur prédéfinies
3 Personnel p2;
4 p2 = Personnel.MEDECIN;
5 p = p2;
6 // Simplification pour l'affichage :
7 // la valeur de l'enum se transforme automatiquement en String
8 System.out.println(p); // écrit MEDECIN
  
```

2.b Associer des attributs à une valeur d'enum

On va pouvoir ajouter des informations liées à chacune des valeurs de l'`enum`. En Java on va ajouter des attributs directement à la structure `enum` et un constructeur qui permet d'associer différentes valeurs d'attributs aux différentes valeurs de l'`enum`.

```

1 // dans le fichier Personnel.java
2 enum Personnel {
3     AIDE_SOIGNANT(260),    // noms des constantes séparées par une virgule
4     INFIRMIERE(420),
5     INTERNE(342),
6     MEDECIN(620);          // et terminé par point-virgule
7
8     // constructeur pour associer une valeur d'enum à une valeur d'indice
9     Personnel (int indice) {
10         this.indice = indice;
11     }
12
13     // indice dans la grille salariale
14     int indice;
15 }
```

Exemple d'utilisation :

```

1 Personnel p3;
2 p3 = Personnel.INFIRMIERE;
3 System.out.println("L'indice d'un " + p3 + " est " + p3.indice);
4 // écrit sur la sortie standard :
5 // L'indice d'un INFIRMIERE est 420
```

tableaux de données

Fiche

<u>Prérequis</u>	Chapitre 3 : Classe et Instance Chapitre 5 Le diagramme APO
<u>Objectif pédagogique</u>	Représenter une séquence de données du même type. Les collections Java seront vues au deuxième semestre. Les exercices d'algorithmique de ce semestre seront tous effectués sur cette structure de donnée statique.
<u>Intérêt</u>	<ul style="list-style-type: none"> • Représenter une séquence de données du même type. • Comprendre les mécanismes de manipulation des collections simples • Maîtriser la différence entre l'instanciation d'un tableau et l'instanciation de ce qu'il contient.
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> • Etre capable de créer et d'utiliser un tableau • Manipuler simplement les éléments d'un tableau, qu'ils soient de type primitif ou de type référence. • Etre capable d'expliquer et de prévoir le comportement d'un algorithme contenant un tableau.

1 Introduction

En programmation, on a besoin la plupart du temps de traitements sur un ensemble, une collection d'objets **de même type**. Le tableau est la structure de données la plus simple pour représenter une séquence de données du même type. Une séquence signifie que l'on introduit un ordre des éléments dans la collection. Chaque élément du tableau aura donc un indice qui correspondra à son classement dans le tableau.



Le rôle d'un tableau est de mémoriser n éléments de type A . n doit être connu au moment de la création du tableau, c'est-à-dire qu'il faut connaître la taille du tableau au moment où ce dernier est instancié en mémoire. Le nombre d'éléments d'un tableau ne peut plus être modifié une fois que le tableau a été instancié.

Note



Un tableau permet un *accès aléatoire* aux éléments qu'il contient. Cela signifie que le coût algorithmique pour accéder à n'importe quelle case du tableau est constant. L'accès à un élément se fait directement par son indice.



La structure de données tableau est *très* peu utilisée dans les applications réelles (de la "vraie" vie).

L'objectif de la manipulation des tableaux est essentiellement **pédagogique** : les tableaux vont nous servir pour faire des exercices algorithmiques et des exercices sur la manipulation de données primitives ou de références.

Soyons rassurés: nous verrons en partie C les classes Java qui permettent de manipuler des collections dans la "vraie" vie.

2 Déclaration et instantiation

2.a Déclaration

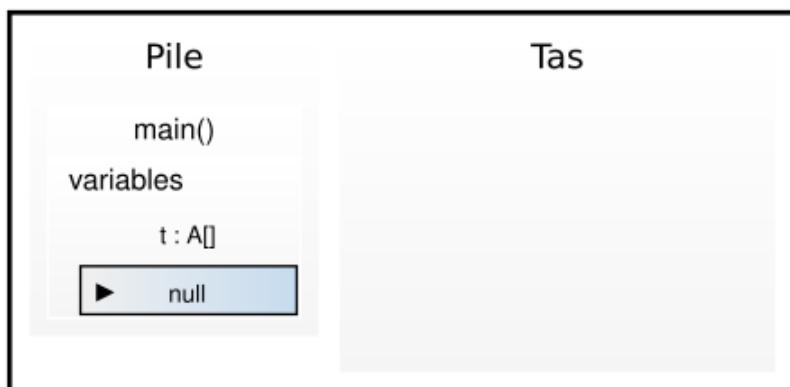
```
1 A[] t;
```

On peut lire de droite à gauche: *déclaration de t , un tableau d'éléments de type A* .

Note



On préférera la notation `A[] t;` à la notation `A t[]`; moins lisible puisque les `[]` indiquant un tableau devraient porter sur le type des éléments que l'on met dans le tableau plutôt que sur la référence du tableau.





Quel que soit le type A, c'est-à-dire qu'il soit de type primitif ou de type référence, t est une **référence**. C'est-à-dire que A[] est de type référence.

2.b Instanciation

Comme t est une référence vers un tableau, il faut **instancier** le *tableau*. Au moment de son instantiation, le tableau contiendra exactement le nombre de cases du tableau et chaque case aura la taille du type des éléments à stocker (c'est-à-dire soit la taille d'un type primitif, soit la taille d'une référence). Il faut donc indiquer, lors de l'instanciation du tableau, le nombre de cases et le type des éléments contenus dans les cases.

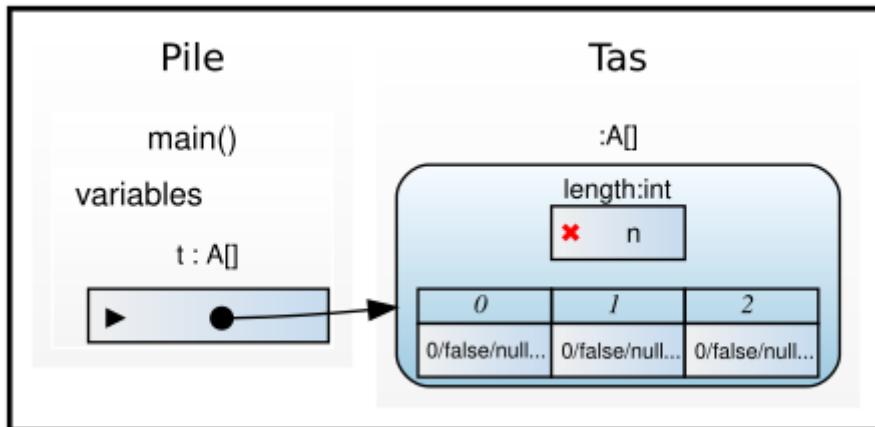
```
1 t = new A[n]; // n est de type int et est positif
```

Note



L'instanciation d'un tableau se fait avec l'opérateur **new**, mais au lieu des parenthèses (), on utilise les crochets [] pour faire appel au constructeur implicite des tableaux. Le constructeur de tableau:

- réserve l'emplacement mémoire nécessaire pour stocker n éléments de type A
- initialise toutes les cases à leurs valeurs par défaut



Lors de l'instanciation d'un tableau, on fait appel au constructeur de tableau. **On ne fait pas appel** au constructeur du type des éléments du tableau. Le constructeur du type A n'est jamais appelé ici. Attention à ne pas confondre `new A[3]`; et `new A(3)`;



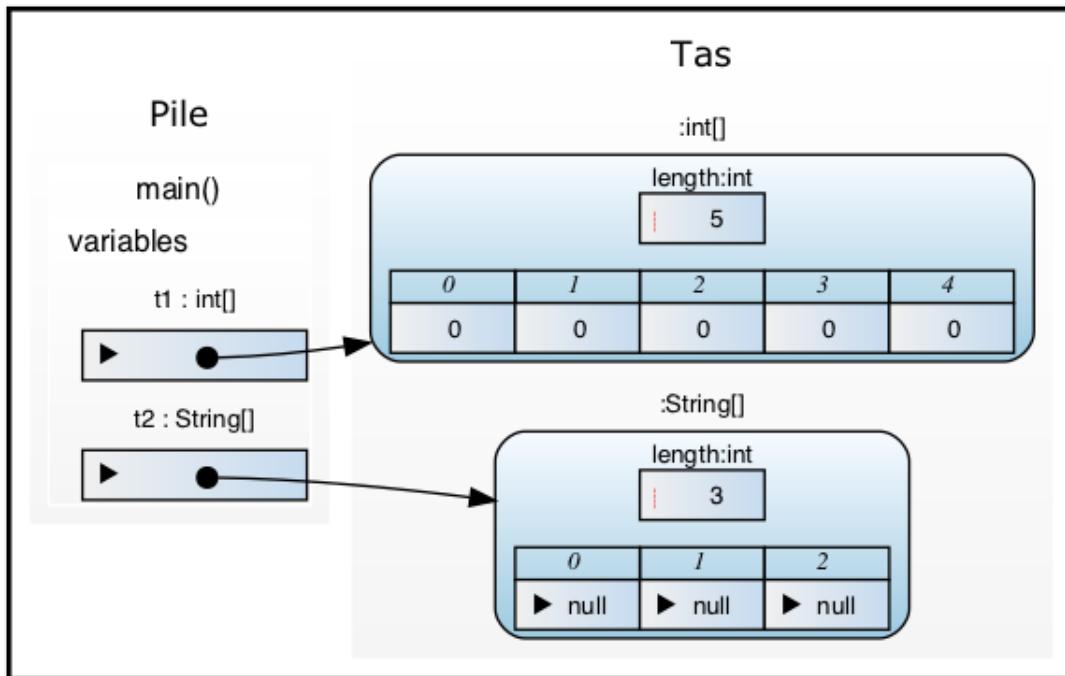
Si A est de type référence, toutes les cases sont initialisées à **null** ! Il faudra prendre soin de bien remplir / initialiser ces cases avant l'utilisation du tableau, sous peine de `NullPointerException`.

2.c Exemple

```

1 int[] t1;
2 String[] t2;
3 t1 = new int[5];
4 t2 = new String[3];

```



La croix rouge sur l'attribut length du tableau signifie que l'attribut est accessible directement (sans passer par une méthode accesseur) mais il n'est pas modifiable.

Par exemple, le code suivant:

```
1 System.out.println("La taille du tableau t1 est " + t1.length);
```

écrit:

```
1 La taille du tableau t1 est 5
```



Dans l'exemple précédent, les tableaux t1 et t2 ont été déclarés et instanciés. Mais ils n'ont pas été remplis/initialisés : chacune des cases du tableau a été initialisée avec sa valeur par défaut. Dans les cas des types référence, comme pour String, il s'agit de la valeur null. Si l'on écrit par la suite la ligne suivante:
t2[1].length();

pour obtenir la longueur de la chaîne de caractères dans la deuxième case du tableau, on obtiendra un NullPointerException !!

3 Accès à une case du tableau

Pour accéder à une case du tableau, on utilise l'opération [i] où i est le numéro de la case.

Par exemple, sur l'exemple précédent, la ligne suivante:

```
1 System.out.println("Le contenu de la case no 3 du tableau t1 est: " + t1[3]);
```

1 Le contenu de la case no 3 du tableau t1 est: 0



La première case d'un tableau a l'indice 0. L'indice de la dernière case d'un tableau t est donc `t.length -1`.

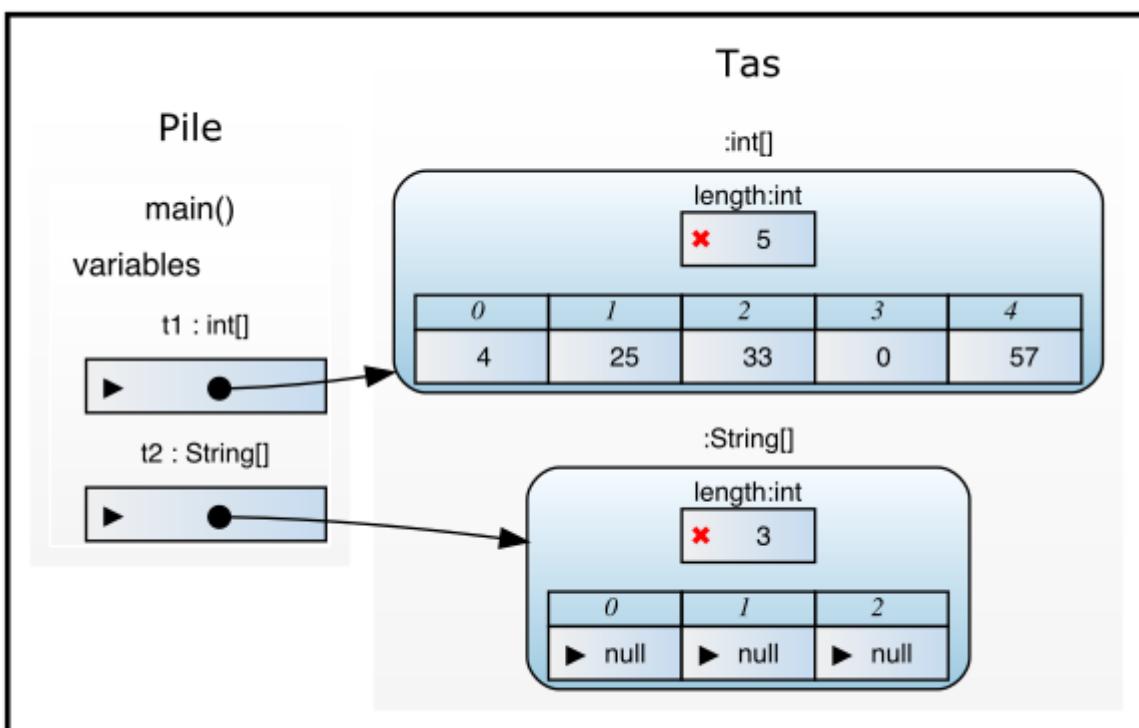
L'opérateur `[]` donne accès à la fois en *lecture* et en *écriture*. C'est-à-dire que l'on peut lire le contenu de la $i^{\text{ème}}$ case d'un tableau t grâce à `t[i]`, mais on peut aussi remplir cette même case si l'on met `t[i]` à gauche de l'affectation (`=`).

Par exemple, après le code suivant:

```

1 // Accès en écriture à la première case (no 0) de t1
2 t1[0] = 4;
3 // Accès en lecture à la première case (no 0) de t1
4 System.out.println("Le contenu de la case no 0 de t1 est maintenant:" + t1[0]);
5
6 t1[1] = 25;
7 t1[2] = 33;
8 // Accès en écriture à la dernière case (no 4) de t1
9 t1[4] = 57;
10 // Accès en lecture à la dernière case (no 4) de t1
11 System.out.println("Le contenu de la case no 4 de t1 est maintenant:" + t1[4]);
```

On a en mémoire :



Note

On notera que la case no 3 du tableau référencé par `t1` n'a pas été modifiée...



Dans l'exemple ci-dessus, `t1` a 5 cases, donc `t1.length = 5`. Le numéro de la dernière case de `t1` est donc `4 = t1.length - 1`.

Ainsi, le code suivant pourra générer des erreurs:

```

1 t1[5] = 42;
2 // -> lève une exception de type 'ArrayIndexOutOfBoundsException'
3 System.out.println("Le contenu de la case -1 est:" + t1[-1]);
4 // -> lève une exception de type 'ArrayIndexOutOfBoundsException'
5 t1[t1.length - 1] = 77;
6 // -> ne génère pas d'erreur (remplace 57 par 77)
7 t1[t1.length] = 0;
8 // -> lève une exception de type 'ArrayIndexOutOfBoundsException'
```



`ArrayIndexOutOfBoundsException` se traduit de l'anglais littéralement par *exception d'indice de tableau hors des limites*, ce qui veut bien dire que l'index utilisé n'est pas dans les indices du tableau...

4 Remarques sur les tableaux

4.a Troatation et initialisation

Pour les tableaux, le principe de troation (cf chapitre 5, section 4.b) devra être respecté. C'est-à-dire que pour chaque tableau, il faudra bien penser à

1. le déclarer
2. l'instancier
3. l'initialiser

L'initialiser va signifier remplir les cases du tableau. Cela est d'autant plus important si le tableau contient des éléments de type références. En effet, par défaut, un tableau de référence ne contient, à sa création, que des valeurs `null` dans chacune de ses cases.

Note



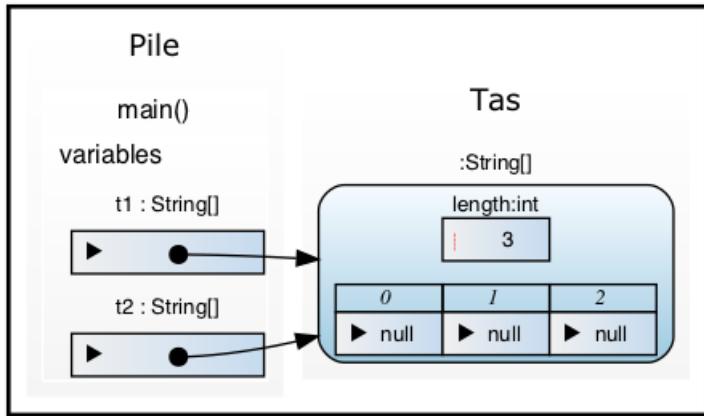
Si une classe a un attribut de type tableau, pour initialiser cet attribut, il va falloir instancier un tableau. Il faudra également penser à initialiser *chacune* des cases du tableau.

4.b Un tableau est de type référence

Si l'on perd la référence à l'instance du tableau, on ne peut plus jamais y accéder. Si l'on passe un tableau en paramètre, c'est la référence vers l'instance qui est passée en paramètre (la méthode qui reçoit ce paramètre peut modifier les éléments du tableau). On peut avoir plusieurs références pour la même instance de tableau.

```

1 String[] t1; // déclaration d'une référence t1
2 t1 = new String[3]; // instantiation d'un tableau de 3 cases
3
4 String[] t2; // déclaration d'une référence t2
5 t2 = t1; // il y a maintenant 2 références qui pointent sur la même instance
       de tableau
```

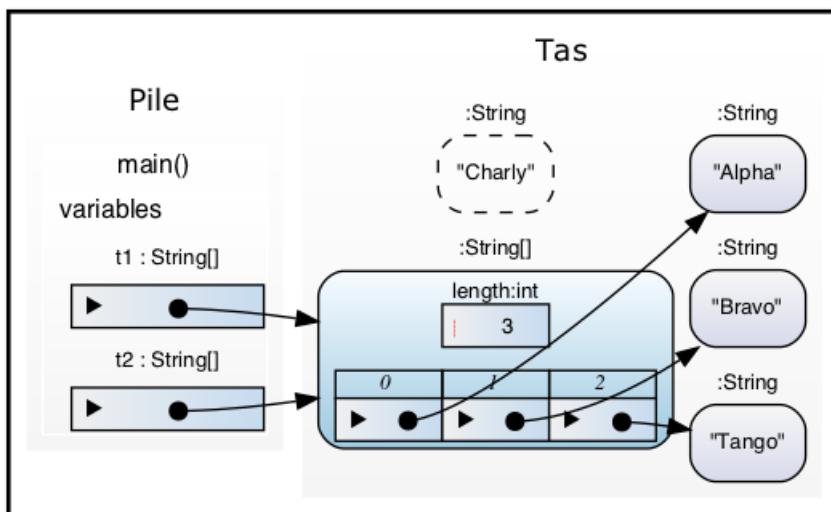
**Note**

Les cases de l'instance du tableau contiennent toujours `null`, il serait peut être temps de l'initialiser...

```

1 t1[0] = "Alpha";
2 t2[1] = "Bravo";
3 t1[2] = "Charly";
4 t2[2] = "Tango";

```



4.c Initialisation avec l'opérateur { }

L'initialisation d'un tableau peut également se faire grâce à l'opérateur { }.

```

1 int[] t = {5, 24, 33, 56};
// est équivalent à
3 int[] t = new int[4];
4 t[0] = 5;
5 t[1] = 24;
6 t[2] = 33;
7 t[3] = 56;

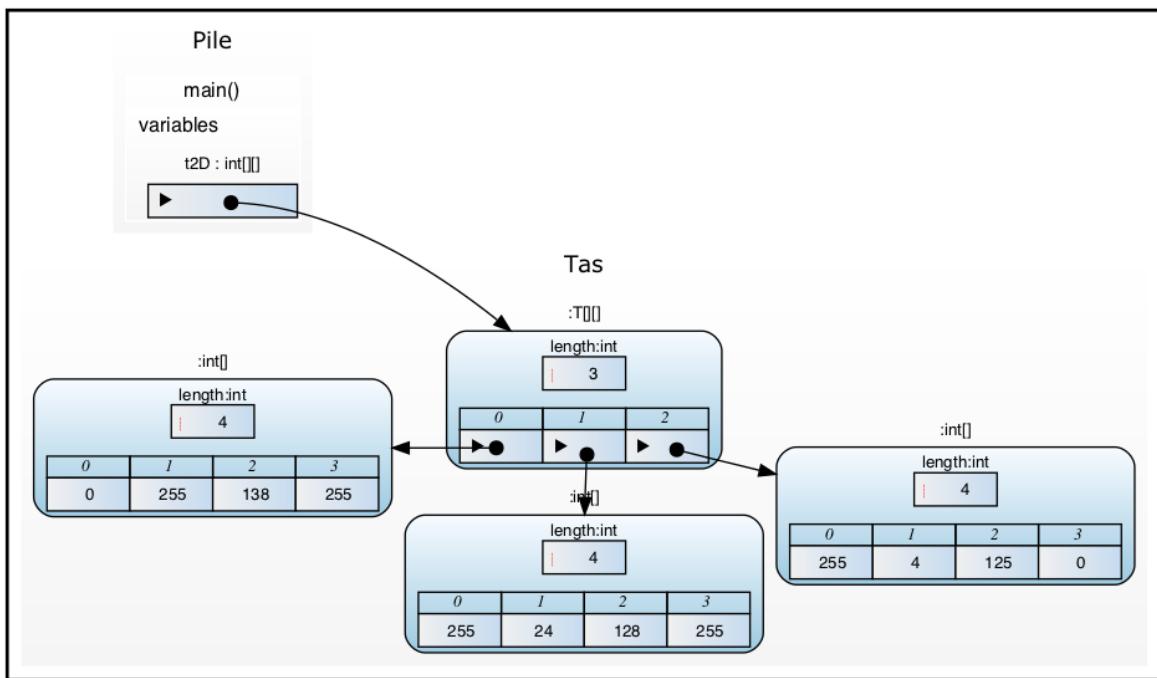
```

5 Tableau à n dimensions

Il peut arriver que l'on se serve de tableaux à plusieurs dimensions pour représenter des objets mathématiques, des matrices/tenseurs, etc. On utilise également beaucoup de tableaux à 2 dimensions pour représenter et traiter des images 2D ou 3D.

Malheureusement, physiquement, la mémoire a 1 seule dimension. En programmation, les tableaux à 2 ou n dimensions sont donc représentés par un assemblage de tableaux 1D.

Par exemple, pour un tableau 2D de 3×4 éléments de type entier, on a en mémoire:



Un tableau qui représente une image de 512×256 va donc être stocké en mémoire par 512 tableaux de 256 cases + 1 tableau qui permet de référencer les 512 tableaux (i.e. un tableau de 256 tableaux).

5.a Déclaration

Un tableau à n dimensions va être déclaré avec n crochets à la suite du type:

```

1 // Déclaration d'un tableau d'entiers à 1 dimension
2 int[] tableau1D;
3
4 // Déclaration d'un tableau d'entiers à 2 dimensions
5 int[][] tableau2D;
6
7 // Déclaration d'un tableau d'entiers à 3 dimensions
8 int[][][] tableau3D;
9
10 // etc.
11
12 // Déclaration d'un tableau de chaînes de caractères à 2 dimensions:
13 String[][] tableau2DChaines;
```

5.b Instanciation

L'instanciation d'un tableau à plusieurs dimensions peut se faire en une ou plusieurs fois.

```

1 // Instanciation indépendante de chaque sous-tableau
2 int [][] tableau2D;
3 tableau2D = new int [512];
4 for (int j = 0; j < 512; j++){
5     tableau2D[j] = new int [256];
6 }
7
8 // Instanciation en une seule fois
9 int [][] [] tableau3D;
10 tableau3D = new int [512][256][34];

```

5.c Initialisation

L'initialisation d'un tableau à n dimensions peut également se faire en ligne ou en plusieurs fois.

```

1 String [] [] exemple2D = {{"on bouffera vite fait dans la bagnole", "nous
    mangerons rapidement dans la voiture", "nous prendrons une rapide collation
    dans le véhicule"}, {"familier", "courant", "soutenu"}, {"mon pote", "un
    enseignant", "un courrier papier"}};
2
3 // ou
4 for (int j = 0; j < tabEntier2D.length; j++) {
5     for (int i = 0; i < tabEntier2D[j].length; i++) {
6         tabEntier2D[j][i] = 0;
7     }
8 }

```


Structure algorithmique de parcours

Fiche

<u>Prérequis</u>	Chapitre 10 : Tableaux de données
<u>Objectif pédagogique</u>	Répéter un même traitement plusieurs fois sans réécrire du code.
<u>Intérêt</u>	La structure algorithmique de parcours est l'une des structures les plus utilisées en algorithmique. Encore faut-il savoir comment l'utiliser à bon escient et proprement.
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> • Savoir quand utiliser une structure algorithmique de parcours • Savoir utiliser les 2 formes de structure algorithmique de parcours • Savoir <i>dérouler</i> une structure algorithmique de parcours écrite par quelqu'un d'autre

1 Introduction

La structure algorithmique de parcours sert à :

- répéter un même traitement sur un *ensemble d'éléments* d'une structure de données (par exemple un tableau). Par exemple
 - afficher toutes les valeurs d'un tableau
 - saisir ou initialiser toutes les valeurs d'un tableau
- permet de calculer une nouvelle donnée qui nécessite *toutes* les données (ou un sous-ensemble précis des données) d'une structure de données (tableau ou collection). Par exemple:
 - calcul d'une moyenne
 - nombre de noms qui commencent par la lettre "P"

Note



On n'est pas obligé d'avoir un tableau pour écrire une structure algorithmique de parcours.

La structure algorithmique de parcours est aussi appelée *boucle de répétition*. Elle ne doit être utilisée que pour les cas énoncés ci-dessus.



Dans une structure algorithmique de parcours, on doit forcément connaître précisément l'ensemble (ou le sous-ensemble) des éléments à parcourir.

- on connaît à l'avance le début et la fin
- on sait exactement à l'avance combien de fois le traitement est répété.

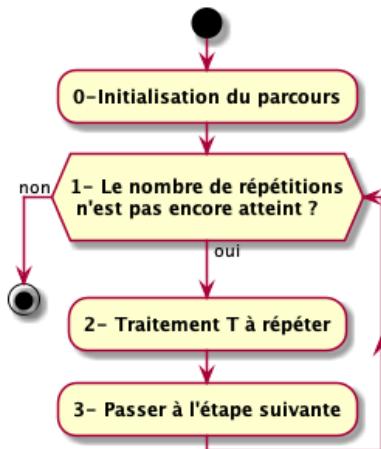


Il est **interdit** d'interrompre un parcours.

2 Déroulement

Une structure algorithmique de parcours se déroule toujours de la même façon:

- 0- initialisation du parcours
- 1- question: a-t-on atteint le nombre d'itérations prédefini ?
- Si **oui**
 - 2- on effectue le traitement T
 - 3- on passe à l'étape suivante
- Si **non** on sort du parcours



2.a Le compteur

Dans une structure algorithmique de parcours, on va toujours utiliser un compteur. Ce compteur va être initialisé avant le parcours et va être modifié à chaque étape du parcours pour tenir compte du fait qu'une étape a été réalisée.

Ainsi, durant la phase d'initialisation du parcours, on va déclarer et initialiser un compteur, généralement de type `int`. Par exemple:

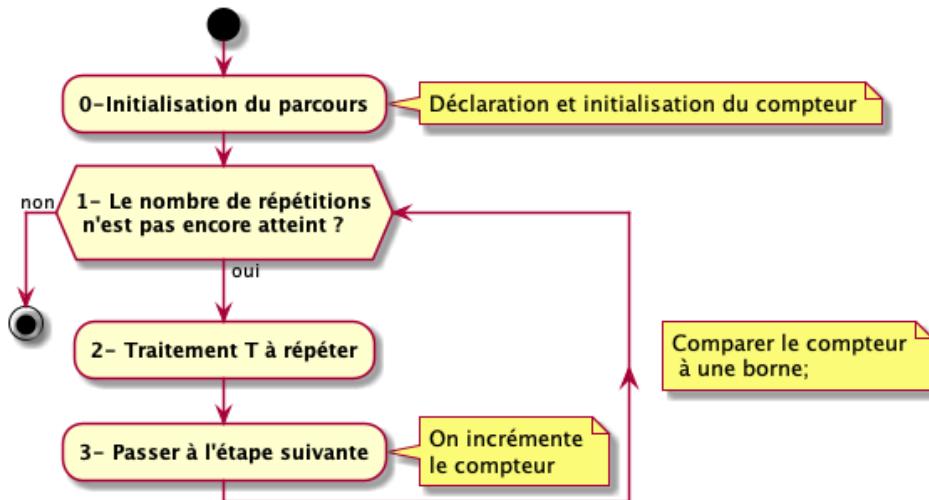
```
1 int i = 0;
```

Lorsque l'on se pose la question si le nombre de répétitions est atteint, on va comparer ce compteur à une borne. Par exemple

```
1 i < t.length;
```

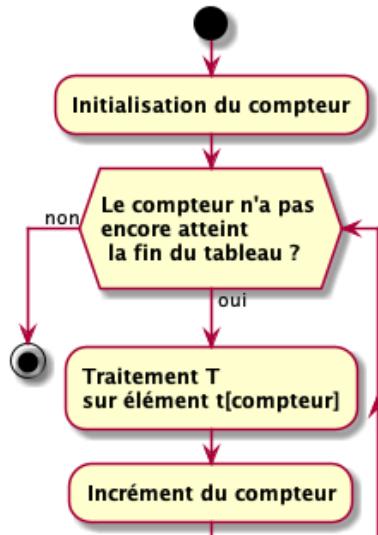
Enfin, à chaque fois que l'on terminé le traitement T, on incrémente (c'est-à-dire que l'on augmente la valeur) du compteur (généralement de 1).

```
1 i++; // équivalent à i = i + 1;
```



2.b Pour les tableaux

Le compteur va permettre la plupart du temps d'indiquer la case courante dans laquelle ou avec laquelle on va effectuer le traitement. Pour les parcours de tableaux, l'initialisation du compteur se fera à 0 (première case du tableau) et la répétition du traitement se fera `t.length` fois (de 0 à `t.length - 1`).



3 En Java

3.a Forme classique

En java, la structure algorithmique de parcours se traduit par `for`. Dans sa forme classique, le `for` comprend 3 sections entre parenthèse séparées par des points virgule (;). La première section correspond à l'initialisation du compteur (dans 95% des cas, l'initialisation consistera à `int i = 0;`), la deuxième section correspond à la question *Le nombre de répétitions n'est pas encore atteint ?*, il s'agit de la comparaison du compteur `i` avec une borne prédéfinie. La troisième section correspond à l'incrémentation (ou le décrément dans certains cas) du compteur.

```

1   for (int i = 0; i < 5; i++) {
2     // (1)      (2)    (3)  (4)
3       System.out.println(i);
4     //      (5)
5   }
6 // (6)
7 // (1): déclaration et initialisation du compteur i
8 // (2): a-t-on déjà écrit 5 chiffres ?
9 // (3): incrémentation du compteur
10 // (4): début du bloc d'instructions T
11 // (5): Traitement T à effectuer (ici on écrit le chiffre courant sur l'écran)
12 // (6): fin du bloc de traitement, ici on effectue (3) puis (2)
```

Une illustration pas à pas du déroulement pas à pas de cette structure algorithmique de parcours est disponible sur CaseInE.



Il est préconisé de déclarer le compteur **dans l'instruction for** et jamais en dehors de la boucle, comme illustré dans le code ci-dessous.

```

1 // NE JAMAIS FAIRE CA (sauf cas exceptionnel très très très très très très rare)
2 int i;
3 for (i = 0; i < t.length; i++) {
4 }
5 // ici i est utilisable ET i vaut t.length
6 // le compilateur autorise la ligne suivante :
7 System.out.println("dernière valeur = " + t[i]); // ->
8           ArrayIndexOutOfBoundsException
9
10 // SOLUTION RECOMMANDEE = bonne pratique
11 for (int j = 0; j < t.length; j++) {
12   // ici le bloc de la boucle est à la portée de j
13 }
14 // après la boucle, j n'est pas utilisable
15 // le compilateur n'autorise pas la ligne suivante :
16 System.out.println("dernière valeur = " + t[j]);
```

3.b Forme *foreach*

En java, lorsque l'on utiliser une structure algorithmique de parcours sur un tableau, il est préférable (plus lisible) d'utiliser la forme *foreach* de cette structure. Les étapes précédentes sont toutes présentes, mais le compteur devient implicite. C'est-à-dire que l'on n'écrit pas la déclaration, l'initialisation ni l'incrémentation du compteur `i` qui correspond à l'indice des éléments du tableau, cette opération est faite de manière automatique et silencieuse.

```

1 A[] t;
2 // Création et remplissage de t....
3 //...
4 // Forme "foreach" (pour chaque)
```

```
5  for (A e : t) { // Pour chaque élément e de type A contenu dans le tableau t
6      // Traitement T à répéter t.length fois
7      // e prend successivement les valeurs de toutes les cases de t
8      // e est donc bien de type A
9 }
```

Par exemple, le code suivant est illustré pas à pas sur CaseInE.

```
1 int[] t;
2 t = new int[4];
3 t[0] = 3;
4 t[1] = 5;
5 t[2] = 1;
6 t[3] = 2;
7
8
9
10 for (int e: t) {
11     System.out.println(e);
12 }
```



La forme *foreach* de la structure algorithmique de parcours ne donne accès qu'en lecture à des éléments d'un tableau. On ne peut donc pas l'utiliser pour écrire dans les cases du tableau.

Structure algorithmique de recherche

Fiche

<u>Prérequis</u>	Chapitre 10 : Tableaux de données
<u>Objectif pédagogique</u>	Effectuer une recherche sur des données, effectuer un traitement un nombre inconnu de fois.
<u>Intérêt</u>	La structure algorithmique de recherche est l'une des structures les plus utilisées en algorithmique. Encore faut-il savoir comment l'utiliser à bon escient et proprement.
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> • Savoir quand utiliser une structure algorithmique de recherche • Savoir utiliser la structure algorithmique de recherche avec les bonnes conditions • Savoir <i>dérouler</i> une structure algorithmique de recherche écrite par quelqu'un d'autre

1 Introduction

Il existe des cas où on ne connaît pas le nombre de répétitions à faire pour effectuer un algorithme. On ne peut donc pas effectuer un parcours. On souhaite alors réaliser un traitement *tant qu'un certain évènement n'est pas survenu*.

Par exemple, si l'on demande de rentrer un mot de passe. Le traitement est *saisir le mot de passe*. L'événement attendu est: (*le mot saisi correspond au mot de passe stocké en mémoire*) ou bien (*trois essais ont été fait*). La structure algorithmique de parcours ne permet pas de traiter ce problème (on sait qu'il y aura au plus 3 itérations, mais on ne sait pas combien il y en aura en tout...). Ici, on utilisera une structure algorithmique de recherche, c'est-à-dire que l'on va *rechercher* la survenue d'un évènement.

Définition



La structure algorithmique de recherche est utilisée lorsque l'on recherche la survue d'une certaine propriété P. P peut être liée à une valeur ou à un calcul.



Lorsque l'on recherche la propriété P dans une structure de donnée (comme un tableau par exemple), il faudra *toujours* veiller à ne pas dépasser les bornes de la structure (sinon, on obtient une `ArrayIndexOutOfBoundsException`). On ajoutera à la propriété P recherchée, une clause L pour éviter de sortir des limites.

Par exemple, si l'on recherche le premier nom dans une liste qui commence par la lettre "X",

- la propriété P recherchée est : "*on a trouvé un nom qui commence par la lettre X*"
- la clause L est : "*on est arrivé au bout de la liste*"



Attention à l'utilisation abusive du mot *rechercher* en français ou en mathématiques. Par exemple, lorsque l'on dit que l'on *cherche* le nom le plus court de la liste, il faut **parcourir** toute la liste pour connaître le nom le plus court (indice: lorsque l'on a "le plus" dans l'expression, il y a de fortes chances qu'il faille regarder tous les éléments).

A la fin de la structure algorithmique de recherche, il y a 2 situations possibles:

- La clause L est vérifiée (on a atteint la limite)
- La propriété P est vérifiée (on a trouvé la propriété que l'on cherchait !)

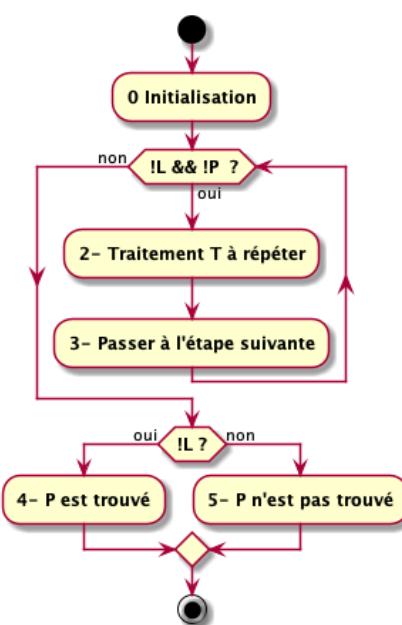


Dans le cas où l'on sort de la boucle parce que l'on a atteint la limite L, il se peut que l'on **ne puisse pas tester P**.

2 Déroulement

Une structure algorithmique de recherche se déroule la plupart du temps de la manière suivante:

- 0- initialisation
- 1- tant que **!L et puis !P** faire
 - 2- on effectue le traitement T
 - 3- on passe à l'étape suivante
- 4- Si (**! L**)
 - On est dans le cas P trouvée
- 5- Sinon
 - On est dans le cas P non trouvée



Note



Il arrive parfois qu'il n'y ait pas de clause L (si l'on ne recherche pas dans une structure de donnée). La condition finale portera donc sur P.



L'instruction algorithmique de recherche est le contraire de "Arrêter dès que", c'est-à-dire que les tests sont inversés. En effet, "*on effectue ce qui est dans la boucle tant que !A*" signifie la même chose que "*on sort de la boucle dès que A*"

- La clause L correspond à "*on est arrivé à la fin de la structure*", dans la condition de "*tant que*", on retrouve !L (*on continue tant que !L* ou bien *on sort dès que L*)
- L'évènement recherché pour sortir du la structure algorithmique de recherche est P (par exemple " on a trouvé un nom commençant par 'X'"), alors la condition que l'on retrouve dans le "*tant que*" est !P. (*on continue tant que !P* ou bien *on sort dès que P*)

3 En Java

```

1 // initialisation
2 int i = 0;
3 while (!L && !P) {
4     T; // traitement à répéter (peut être absent)
5     i++;
6 }
7 // Tester la situation à la fin de la recherche
8 if (!L) {
9     // Traitement pour P trouvée
10 } else {
11     // Traitement pour P non trouvée
12 }
```

Par exemple:

```

1 int[] t = {3, 5, 1, 2};
2 int nb = 1;
3
4 // Initialisation: ici, on utilise l'indice i pour les cases du tableau
5 int i = 0;
6
7 // On recherche si le chiffre nb est dans le tableau
8 // La clause de limite L est la sortie de tableau
9 // L = i >= t.length (attention si i == t.length, on n'est déjà plus dans le
10 // tableau)
11 // donc !L = i < t.length
12 // P = t[i] == nb; La propriété que l'on cherche est "la valeur d'une case du
13 // tableau est nb" donc !P = t[i] != nb;
14 // On a donc:
15 while ((i < t.length) && (t[i] != nb)) {
16     //      (!L)          && (!P)
17     // Ici, il n'y a pas de traitement T à effectuer
18     i++; // passage à la case suivante
19 }
20 if (i < t.length) {
21     //      (!L)
22     System.out.println("Le tableau contient le nombre 1");
23 } else {
24     System.out.println("Le tableau ne contient pas le nombre 1");
25 }
```

Note

On remarque ici que le `i++;` n'est pas contenu dans la structure (contrairement à la forme du `for`). Il faudra donc bien veiller à ne pas oublier d'écrire explicitement l'instruction de passage à l'étape suivante !



Dans l'exemple ci-dessus, l'ordre des conditions est **essentiel** ! En effet, si l'on avait écrit `while ((!t[i] == nb) && (i < t.length))`, alors l'algorithme serait tout simplement faux ! Une illustration de cet exemple est donné sur CaseInE.



Dans le `if` qui suit le `while`, on ne peut que tester la condition L, pas la condition P. En effet, si l'on écrit `if (t[i] == nb)` (qui peut paraître plus lisible pour savoir si oui ou non on a trouvé nb...), alors dès que le nombre recherché ne sera pas dans le tableau (par exemple `nb=42`), alors la sortie du `while` ne se fera pas parce que P est vraie, mais parce que L est vraie. ce qui voudra dire que l'on sera resté dans la boucle `while` pour chacune des cases du tableau, que l'on aura fait `i++` avec `i = t.length - 1`, ce qui voudra dire qu'en sortie de la boucle, i vaudra `t.length`. Du coup, si à l'étape d'après on écrit `if (t[i] == nb)`, i vaut `t.length` et `t[i]` n'existe pas. On récupère donc une `ArrayIndexOutOfBoundsException` !!

4 Boucle Faire

Dans de très (très très très très) rare cas, on doit effectuer la recherche en ayant fait *au moins* une fois le traitement.

Dans ce cas, on utilisera la structure `do ... while` de la même façon que la `while`.

```

1 // Initialisation
2
3 do{
4     T    // Traitement à répéter (peut être absent)
5     ... // Passage à l'étape suivante
6 } while (!L && !P);
7 if (!L) {
8     // Traitement pour P trouvée
9 } else {
10    // Traitement pour P non trouvée
11 }
```

5 Parcours ou recherche ?

Préconisations pour déterminer la nature de la structure algorithmique à utiliser (parcours ou recherche ?):

- dans le cas d'un parcours, on a besoin de tous les éléments d'une structure de donnée (tableau, liste, etc.) ou bien on connaît à l'avance le nombre de répétition à faire (ex: pour tous les entiers de 0 à 5)
- dans le cas d'une recherche, on a une propriété P ou !P qui permet d'arrêter le traitement (par exemple, lorsque l'on cherche un élément, la propriété P est que l'on a trouvé cet élément)

Ensuite:

- Si c'est un parcours :

1. on détermine l'objectif du parcours (la/les instruction(s) T à répéter) (5)
 2. on détermine les limites : cas de départ (1) et borne supérieure (2)
 3. on détermine comment passer d'une étape à l'autre (3)
- Si c'est une recherche :
 1. on détermine la propriété P et on *calcule !P*
 2. on détermine si la recherche se fait sur une structure de données avec des limites. Si c'est le cas, on détermine la clause de sortie L et on *calcule !L*
 3. on détermine comment passer à l'étape suivante (attention à ne pas oublier de l'écrire explicitement dans la boucle)
 4. on détermine s'il y a besoin d'effectuer un traitement à chaque étape T
 5. on détermine le traitement à effectuer lorsque la limite est atteinte et la propriété non trouvée
 6. on détermine le traitement à effectuer lorsque la propriété a été trouvée



Si dans une structure algorithmique de recherche, on n'a que la clause L (c'est-à-dire que l'on n'a pas de propriété recherchée P), c'est qu'il s'agit en fait d'un parcours ! Il **faut** alors réécrire l'algorithme avec un **for**.



Si dans une structure algorithmique de parcours, on aimeraient bien écrire **si** (...), **alors c'est terminé**, c'est qu'il s'agit en fait d'une structure algorithmique de recherche. On **doit** utiliser un **while**.



Vous trouverez très fréquemment sur internet et a fortiori sur les IAs génératives qui vous *aident* à coder des recherches effectuées avec des **return** dans les structures de parcours (**for**). Voici les raisons pour lesquelles dans ce cours (et de manière générale pour un code lisible et maintenable) vous **devez** utiliser les bonnes structures algorithmiques :

Les instructions de type **return** lorsqu'ils ne sont pas la dernière instruction d'une méthode, ou **break** qui sont nécessaire pour faire des recherches avec un **for** sont des héritages des **GOTO** de la programmation impérative non structurée. Depuis la critique d'Edsger Dijkstra qui dénonce en 1968 les méfaits de l'instruction **GOTO** très prompte à erreurs et *indébuggable*, la programmation impérative est devenue **structurée** avec des structures algorithmiques spécifiques. Le **while** ayant été créé en plus du **for** pour spécifier une recherche, l'utilisation de la seule structure de parcours pour faire une recherche est une faute algorithmique qui nuit à la lisibilité du code.

La *simplicité* supposée d'utiliser un **break** dans un **for**, ne vient que d'une situation d'*(e mauvaise)* habitude de programmation. Mettre 2 conditions L et P l'une sous l'autre au lieu de dans un **&&** n'est pas plus simple, c'est une question d'habitude. Ces deux conditions doivent de toutes façons être réfléchies et explicitées avant d'écrire la structure algorithmique.

Part C

Programmation Orientée Objet en Java

Les paquetages en Java

Fiche

<u>Prérequis</u>	<ul style="list-style-type: none"> • Avoir déjà compilé et exécuter un programme Java • Chapitre 2; Présentation de Java
<u>Objectif pédagogique</u>	<ul style="list-style-type: none"> • Comprendre le fonctionnement des <i>packages</i> en Java. Encapsuler et modulariser son code.
<u>Intérêt</u>	<ul style="list-style-type: none"> • Savoir comment structurer un programme à l'aide des <i>package</i> • Pouvoir développer du code modulaire et réutilisable
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> • Savoir utiliser des classes qui viennent des <i>packages</i> Java • Savoir créer des <i>packages</i> • Savoir compiler et exécuter un programme Java avec des <i>packages</i> • Connaître l'arborescence créée lors de la création et la compilation de packages.

Note



Même si la plupart d'entre vous utilisent ou utiliseront à terme des environnements de développement intégrés (Eclipse, Netbeans, IntelliJ...), où la plupart des opérations de compilation et d'exécution sont automatisées, il est néanmoins indispensable de comprendre comment ça fonctionne, afin d'être capable de comprendre d'où viennent les problèmes lorsqu'il y en a....

1 Les paquetages (*packages*)

1.a Généralités et déclaration

En Java, on peut regrouper des classes en *packages* afin de mieux structurer l'application (par exemple, les classes métiers, utilitaires, IHM, etc.). Cette structuration a 3 intérêts:

- permettre la modularité, séparation plus claire des différentes parties de l'application

- éviter les conflits de noms (en fournissant un espace de nommage)
- limiter la visibilité de certaines classes

Pour déclarer qu'une classe fait partie d'un paquetage, on utilise l'instruction `package` en toute première instruction d'une classe (avant même la déclaration de la classe). Exemple:

```

1 package fr.uga.animaux;
2 class Pangolin {
3 // La classe Pangolin fait donc partie du paquetage fr.uga.animaux
4 [...]
5 }
```



Si l'on ne spécifie pas d'instruction `package`, la classe appartient par défaut au paquetage anonyme. Selon les conventions de codage Java (*Java coding style*), les noms de paquetage sont en minuscule (et souvent normalisés).

Note



Le `.` induit une hiérarchie de paquetage: le paquetage `fr.uga.animaux` est un sous-paquetage du paquetage `fr.uga`, lui-même un sous-paquetage de `fr`.

La hiérarchie des paquetages est directement liée aux répertoires contenant les fichiers source. Le paquetage par défaut est dans le répertoire racine `.`, le paquetage `fr.uga` est quant à lui dans le sous-répertoire `./fr/uga`, tout comme le paquetage `fr.uga.animaux` est dans le sous-répertoire `./fr/uga/animaux/`.

1.b Visibilité

L'utilisation des paquetages induit deux niveaux de visibilité pour les classes : `public` et par défaut. Si la classe est déclarée `public`, elle est visible et accessible depuis n'importe quel paquetage. Si on ne met rien, elle ne sera accessible que depuis son paquetage.

1.c Utilisation

Dès qu'une classe fait partie d'un paquetage, elle change de nom et porte en préfixe le nom du paquetage. Ainsi par exemple, notre classe `Pangolin` ci-dessus s'appelle désormais `fr.uga.animaux.Pangolin`. Pour utiliser cette classe, il faut donc :

- soit utiliser son nom complet (on dit aussi nom *qualifié*) `fr.uga.animaux.Pangolin`, comme par exemple:
`fr.uga.animaux.Pangolin animal = new fr.uga.animaux.Pangolin();`
- soit ajouter en début de fichier (après une éventuelle instruction `package`) une instruction d'importation de la classe :
`import fr.uga.animaux.Pangolin;.` Si cette ligne est présente au début du fichier, alors on pourra utiliser le nom court (`Pangolin`) dans tout le fichier.

Note



L'instruction d'importation n'importe pas le code de la classe au sens propre. Cette commande signifie simplement littéralement que l'on a le droit d'utiliser le nom court de la classe importée en lieu et place du nom qualifié.



S'il y a un conflit de nom (une classe `Pangolin` dans deux paquetages différents), alors il faut impérativement utiliser le nom long.

On peut également importer toutes les classes d'un paquetage en utilisant le symbole *. Ainsi, par exemple, `import fr.uga.animaux.*;` signifie simplement que toutes les classes du paquetage `fr.uga.animaux` pourront être utilisées avec leur nom court. Attention, cela ne s'applique pas récursivement aux sous-paquetages (s'il existe par exemple des classes dans un sous-paquetage `fr.uga.animaux.gui`, ces classes ne seront pas importées).

2 Paquetages et compilation / exécution

Lorsque l'on compile une classe d'un certain paquetage, le compilateur place automatiquement le code binaire (le fichier compilé `.class`) dans un sous répertoire dont le nom correspond à celui du paquetage. Ainsi, une classe `Pangolin` appartenant au paquetage `fr.uga.animaux` sera compilée dans le fichier `fr/uga/animaux/Pangolin.class`.

On voit donc qu'il y a une correspondance entre la hiérarchie de paquetages et la hiérarchie du système de fichiers. Cela a une incidence sur la compilation et l'exécution.

2.a Classpath et sourcepath

Pour bien comprendre comment le compilateur et la JVM trouvent les ressources nécessaires lors de la compilation et de l'exécution, il faut parler du `classpath` et du `sourcepath`.

Définition



Le `classpath` est une variable contenant la liste des répertoires du système de fichiers dans lesquels le compilateur et la JVM peuvent aller chercher les *bytecodes* des classes dont ils peuvent avoir besoin.

Définition



Le `sourcepath` est une variable contenant la liste des répertoires dans lesquels le compilateur peut aller chercher les codes sources des classes dont il peut avoir besoin.

On peut spécifier ces deux variables :

- soit en positionnant les variables d'environnement `CLASSPATH` et `SOURCEPATH`. Exemple en bash :

```
1 export CLASSPATH=$CLASSPATH:/home/canard/java:/bin
```

- soit en passant directement le `classpath` et le `sourcepath` en argument du compilateur ou de la JVM (avec les options `-classpath` et `-sourcepath`).

Lorsque le compilateur trouve dans le `classpath` et dans le `sourcepath` deux versions de la même classe qui peuvent convenir, il prend la plus récente. S'il s'agit d'un fichier source, il recompile d'abord ce fichier pour créer le *bytecode* nécessaire.



Le compilateur comme la JVM s'attendent à trouver chaque classe nécessaire à la compilation et à l'exécution **dans le sous-répertoire correspondant à son paquetage** d'au moins un des répertoires du `classpath` ou du `sourcepath` (pour le compilateur).

Par exemple, si l'exécution de la classe `TestPangolin` requiert le *bytecode* de la classe `fr.uga.animaux.Pangolin` situé dans le répertoire `/home/canard/java/fr/uga/animaux`, le `classpath` devra être positionné sur `/home/canard/java`, et non sur `/home/canard/java/fr/uga/animaux`.

2.b Un exemple

Considérons l'exemple suivant où nous voulons compiler puis exécuter notre programme de pangolins. La structure du système de fichiers (à partir de l'endroit où nous nous trouvons) est la donnée dans la figure 13.1 (a).

Afin de compiler correctement, nous devons fixer le `sourcepath` de manière à ce que le compilateur trouve les bonnes classes. Voici ce que l'on peut taper :

```
1 javac -d ./bin -sourcepath ./src -classpath ./bin src/TestPangolin.java
```

Ici, l'option `-d` permet de préciser dans quel répertoire doivent être placées les classes compilées. L'option `-classpath ./bin` n'a aucun effet, mais permettrait de prendre en compte des classes compilées plus récentes que le source si jamais il y en avait dans le répertoire `bin`.

Après cette ligne de commande, notre système de fichier se trouve dans la situation de la figure 13.1 (b).

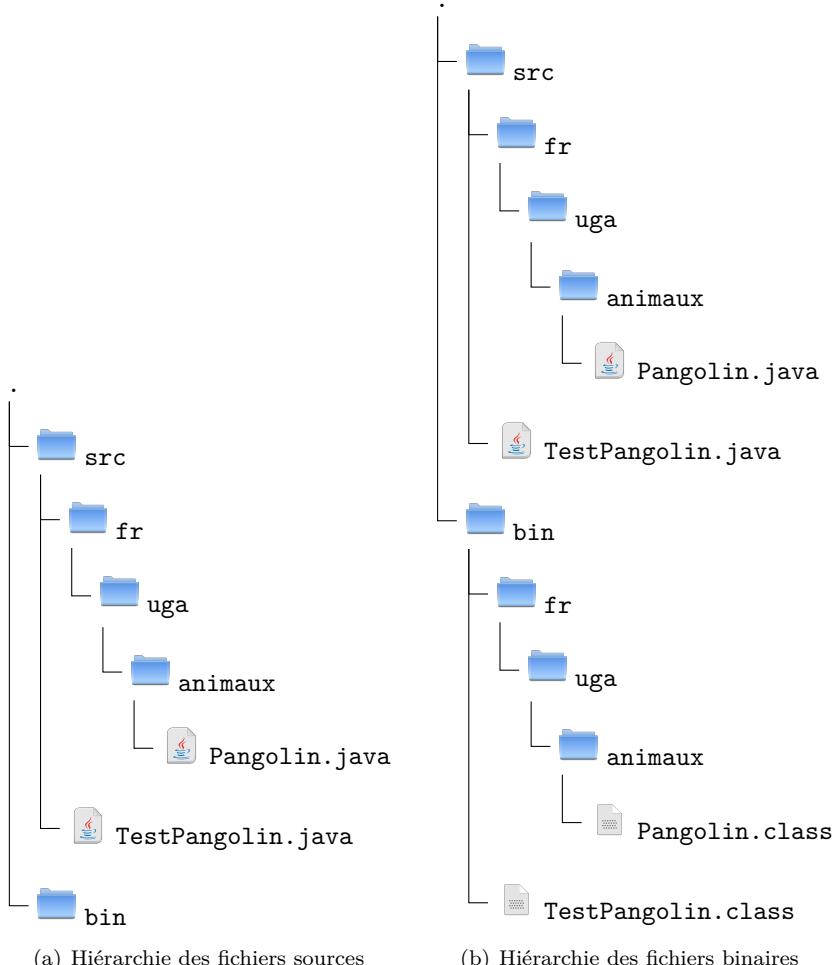


Figure 13.1: Hiérarchie des sources et objets de Pangolin

Le compilateur a créé lui-même la hiérarchie de répertoires dans le répertoire `bin`. Dans cette situation, pour lancer le programme, il suffit de procéder comme suit :

```
1 java -classpath ./bin TestPangolin
```

Encore une fois, ce que l'on passe comme argument obligatoire à la JVM est le nom d'une classe, pas le nom d'un fichier. D'où le fait qu'il n'y ait pas d'extension à `TestPangolin1`, et qu'il ne soit pas nécessaire de préciser que cette classe est dans le sous-répertoire `bin` (ce répertoire est dans le `classpath`).

¹Attention, il faut passer le nom **long** de la classe, ici, la classe `TestPangolin` se trouve dans le paquetage par défaut

3 Archives JAR

Pour distribuer du *bytecode* Java, il peut être utile de fournir une simple archive plutôt qu'un ensemble de fichiers avec une arborescence compliquée. L'écosystème Java possède un outil pour ça, les fichiers JAR (pour Java ARchive – *jar* signifie aussi en Anglais « bocal, pot », ce qui permettait aux créateurs de Java, qui sont des petits rigolos, de faire un bon jeu de mot au passage).

Un fichier JAR ressemble à s'y méprendre à une archive TAR. D'ailleurs, la syntaxe pour créer une archive JAR est quasiment la même. Jugez plutôt :

```
1 jar cvf jean-michel.jar
```

Une spécificité est que l'on peut créer des archives JAR exécutables. Pour cela, il faut :

1. Créer un fichier `manifest.txt` avec le contenu suivant : `Main_Class: TestPangolin` (à supposer que votre classe contenant la méthode `main` soit la classe `TestPangolin`).
2. Créer le JAR en ajoutant le fichier `manifest`, de la manière suivante:

```
1 jar cvfm manifest.txt jean-michel.jar bin/TestPangolin.class bin/fr/ensimag  
/animaux/Pangolin.class
```

Une fois une telle archive créée, vous pourrez lancer la JVM sur `jean-michel.jar` avec l'option `-jar` :

```
1 java -jar jean-michel.jar
```

Pour les adeptes des cliquodromes, la plupart des environnements de bureaux savent lancer une archive JAR exécutable lorsque l'on clique dessus.

Note



Une autre spécificité des fichiers JAR est que l'on peut les utiliser dans un `classpath`. Lorsqu'il y a des fichiers JAR dans un `classpath`, le compilateur ou la JVM va regarder ce qu'ils contiennent en les considérant de la même manière que s'ils étaient des répertoires. Ainsi, lorsque vous récupérez une bibliothèque sous forme d'archive JAR, vous pouvez en utiliser les classes sans même avoir à la décompresser, en incluant directement la bibliothèque dans le `classpath`.

Java: La classe !

Fiche

Prérequis	<ul style="list-style-type: none"> Partie A, Chapitres I: Classe et instance Partie A, Chapitres II: Le diagramme APO Chapitre 2; Présentation de Java
Objectif pédagogique	<ul style="list-style-type: none"> Maîtriser le principe d'encapsulation Connaître les modificateurs d'accès des classes, méthodes et attributs
Intérêt	<ul style="list-style-type: none"> Responsabiliser chaque instance par rapport à ses données
Compétences à acquérir	<ul style="list-style-type: none"> Utiliser à bon escient les modificateurs d'accès Savoir quand écrire des accesseurs et/ou modificateurs Savoir mettre en œuvre le principe d'encapsulation Savoir créer et utiliser des objets immuables.

1 Le principe d'encapsulation

Une classe Java décrit

- la composition et le type des sous-éléments, c'est-à-dire des attributs
- la liste des actions possibles sur les données représentées par la classe, c'est-à-dire les méthodes
- le code, l'implémentation des algorithmes qui permet d'effectuer des actions.

Les actions, on parle en programmation orientée objet de **méthodes**, décrivent les traitements à faire sur les données.

Dans une classe Java, on va donc pouvoir non seulement modéliser un ensemble de données, mais également ajouter des opérations qui vont permettre la manipulation, la transformation et la production de nouvelles valeurs à partir de ces données.

En Programmation Orientée Objet, les classes Java vont suivre le principe d'*encapsulation*. Il s'agit de l'un des 6 concepts fondamentaux de la programmation orientée objet, comme illustré figure 14.1¹.

¹Origine: <https://www.javatpoint.com/java-oops-concepts>

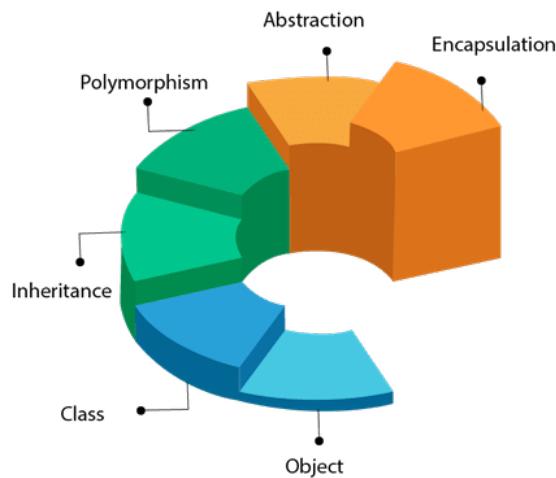


Figure 14.1: Les concepts clés de la Programmation Orientée Objet.

Définition



L'encapsulation est un mécanisme qui consiste à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés.

L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

Dans la suite, nous allons voir, pour la classe elle-même et chacun de ses membres (attributs et méthodes), les outils permettant de garantir le principe d'encapsulation, notamment, les modificateurs d'accès.

Les modificateurs d'accès vont pouvoir définir la **portée** de la classe, des attributs et des méthodes. C'est-à-dire, qu'au moment de la programmation d'une classe, on va devoir décider si d'autres classes et d'autres méthodes (comme par exemple la méthode `main` de la classe `Test`) vont pouvoir accéder, c'est-à-dire connaître ou modifier la valeur de la classe ou de ses attributs ou bien appliquer une méthode sur une instance particulière.

Le tableau suivant énumère les modificateurs d'accès possibles en Java.

Modificateur d'accès UML	Modificateur d'accès Java	Description
+	<code>public</code>	L'accès au membre de la classe est possible dans toute classe programmée en Java, que ce soit à l'intérieur du <i>package</i> considéré ou à l'extérieur. ! Ne pas utiliser ce modificateur d'accès pour les attributs. Ce modificateur d'accès ne fournit en effet aucune protection au membre. Le membre est donc accessible depuis n'importe où par n'importe qui. On réserve ce modificateur d'accès pour les classes que l'on souhaite exporter ainsi que pour les opérations que l'on veut pouvoir exécuter depuis l'extérieur de la classe.
-	<code>private</code>	L'accès au membre n'est possible que depuis les membres de la même classe. L'utilisation de ce modificateur d'accès permet d'encapsuler les attributs. Ce modificateur d'accès permet de créer des actions intermédiaires ou des traitements internes aux données des classes. ! L'accès est possible depuis les autres membres de la même <i>classe</i> , quel qu'en soit l'instance.
#	<code>protected</code>	le membre est accessible depuis les classes de la famille étendue, c'est-à-dire les sous classes et les classes du même <i>package</i> .
	<code>Ø (rien)</code>	le membre est accessible depuis les classes du même <i>package</i> (même répertoire). A réservé à des cas très particuliers.

3 La classe

La déclaration d'une classe en Java se fait

- dans un fichier qui porte le même nom que la classe (attention à la casse) `.java`. En conséquence, dans ce cours, on considérera systématiquement une seule classe par fichier et un seul fichier par classe, le cas des classes internes ne sera pas vu dans ce cours et ne sera pas utilisé dans 99% des cas de ce que vous aurez à programmer à l'avenir)
- à l'aide du mot clé java `class`

Par convention, en Java, le nom des classes commence par une majuscule et comme il ne peut pas y avoir d'espace dans un nom de classe, pour que celui-ci soit suffisamment descriptif, on peut mettre plusieurs mots en mettant la première lettre de chaque mot en majuscule.

Note



Comme décrit dans la convention Java d'Oracle.com (qui définit le JDK)^a :
Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

^a<https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

Vous trouverez ici² une rapide explication de pourquoi les conventions de nommage sont importantes³.

²<https://medium.com/modernnerd-code/java-for-humans-naming-conventions-6353a1cd21a1>

³Rappel: dans ce cours, on ne va pas seulement apprendre à programmer *pour que ça marche*, mais pour que le programme soit facilement maintenable et puisse évolué, c'est-à-dire pour qu'il soit écrit de manière **juste**

Après la définition de la classe, il y a la définition d'un bloc défini par { }.

Exemple:

```
1 // Dans le fichier NomDeLaClasse.java
2 class NomDeLaClasse {
3 }
4 }
```



On remarque dans ce cas que le modificateur d'accès de la classe est `package`... C'est-à-dire que la classe ne sera accessible que par les classes du même package. C'est ok si l'on souhaite faire des tests locaux, mais si l'on souhaite que la classe soit utilisée par d'autres *packages* (c'est-à-dire par plusieurs programmes ou par des collègues qui développent indépendamment une autre partie du programme), on préférera la déclarer avec le modificateur d'accès `public`. De manière générale, dans ce cours, on déclarera toutes les classe `public`, et dans le cas où une classe ne sera pas déclarée `public`, il faudra en justifier la raison dans les commentaires.

```
1 // Dans le fichier NomDeLaClasse.java
2 public class NomDeLaClasse {
3 }
4 }
```

4 Les attributs

Un **attribut**, aussi appelé **variable d'instance**, sert à stocker les données permettant de caractériser l'état d'une instance de la classe.

La déclaration d'un attribut se fait directement à l'intérieur du bloc de la classe, c'est-à-dire **en dehors de toute méthode**.

Lors de la déclaration de l'attribut, on ajoute souvent un modificateur d'accès qui permet d'étendre l'accessibilité de l'attribut à l'extérieur de la classe. La déclaration d'un attribut en Java se fait de la manière suivante:

```
1 modificateurDAcces Type nom;
```

Exemple:

```
1 // Dans le fichier NomDeLaClasse.java
2 public class NomDeLaClasse {
3     private int unAttribut;
4     protected String unaAutreAttribut;
5
6 }
```

Note



Pour des raisons de lisibilité du code et pour respecter les *Java Code Convention*, on ne déclarera qu'un seul attribut par ligne.



Afin de respecter le principe d'encapsulation, il est **FORTEMENT recommandé** d'utiliser le modificateur d'accès `private` pour tous les attributs. Ici, on va énoncer que ce principe doit être **OBLIGATOIUREMENT** respecté (sauf dans le cas où le modificateur d'accès `protected` s'impose^a...).

^acf chapitre 16

Les attributs étant déclarés avec le modificateur d'accès **private** ne sont donc pas accessibles depuis l'extérieur de la classe.

Dans certaines situations, on a quand même besoin :

- de récupérer ou
- de modifier

la valeur de l'attribut.

Pour accéder/modifier un attribut *encapsulé* on va créer deux méthodes dans la classe :

- un accesseur (en anglais *getter*, du mot *get*)
- un modificateur (en anglais *setter*, du mot *set*)



Attention à ne pas confondre **modificateur setter** et **modificateur d'accès public, private, protected, package...**



Attention, le principe d'encapsulation implique que les attributs d'une classes sont déclarés **private** (ou exceptionnellement **protected**). En revanche, les accesseurs et modificateurs ne doivent être créés que

- s'ils ont une utilité **et**
- s'ils ne mettent pas en péril la cohérence de la classe

et non systématiquement !

5 Les méthodes

Une méthode décrit une action possible sur les données définies dans la classe.

La déclaration d'une méthode contient obligatoirement :

- un modificateur d'accès (qui peut être **package**, donc rien)
- le type de la donnée créée et renvoyée par la méthode (qui peut être **void**)
- un nom (souvent un verbe)
- zéro, un ou plusieurs paramètres entre parenthèses

Ces quatre éléments constituent ce qu'on appelle la **signature** d'une méthode.

5.a Modificateur d'accès

Cf. tableau précédent.

5.b Le type

Le type d'une méthode est le résultat de l'exécution de la méthode. Il peut être de type primitif ou de type référence vers une instance. Dans le cas où aucune nouvelle donnée n'est créée, le type de la méthode est **void**.

Si le type de la méthode est **void**, cela indique que la méthode **ne produit aucune donnée** : elle n'a pas pour but de *calculer* une valeur, mais elle *modifie* l'état interne de l'instance.

Il existe un cas particulier pour le type des méthodes : **les constructeurs** n'ont pas de type (même pas **void**).

Si l'on excepte le cas des constructeurs et dans le cas où le type de retour n'est pas **void**, la **dernière instruction** de la méthode doit être une ligne **return**.

```

1  modificateur TYPE nom(...) {
2  // si TYPE n'est pas \texttt{void}, on doit avoir
3  return ....;
4  //      ^----- ici quelque chose de type TYPE
5 }
```



Le fait que l'instruction **return** soit la dernière instruction de la méthode signifie que l'on sort de la méthode après cette instruction, même s'il y a du code après cette instruction (ce code ne sera jamais exécuté). S'il y a un besoin impérieux d'interrompre une méthode, les exceptions (que nous verrons au chapitre 7) sont le seul moyen lisible de procéder. De manière général, une méthode ne **doit** pas être interrompue par un **return** au milieu de la méthode. **return** doit donc également être la dernière instruction écrite dans la méthode.

5.c Le nom

Le nom d'une méthode commence toujours par une **minuscule sauf** dans le cas des constructeurs. D'après la convention de codage Java, le nom d'une méthode doit être un verbe et indiquer l'action de la méthode. Les accesseurs commencent généralement par **get** et les modificateurs commencent généralement par **set**.

5.d Les paramètres

Les paramètres permettent de donner l'ordre, le nom et le types des données **supplémentaires** nécessaires à l'exécution de l'action.

5.e Invocation d'une méthode

Sauf cas particulier⁴, une méthode est invoquée sur une instance (c'est-à-dire sur la référence permettant d'accéder à l'instance).

Une méthode étant définie dans le corps de la classe, l'ensemble des attributs de la classe est accessible à la méthode, même les attributs déclarés en **private**, et ce, même si ces attributs appartiennent à une autre instance de la même classe.

5.f Le constructeur

Un constructeur est une méthode spécifique qui est invoquée lors de l'instanciation (et uniquement à ce moment là), grâce à l'opérateur **new**. Un constructeur **ne peut pas** avoir de type (pas de **return**), pas même le type **void**. Il ne crée pas de nouvelles données mais sert à initialiser l'état de l'instance, c'est-à-dire, les valeurs des attributs.

Note



Le compilateur Java n'a pas besoin que l'on précise le type d'un constructeur, puisque qu'il renvoie forcément la référence vers l'instance créée par le **new** qui appelle le constructeur en question. Cette référence est généralement affectée à une référence déclarée dans la méthode qui appelle le constructeur.

⁴cf chapitre 22



Le rôle d'un constructeur est d'initialiser les attributs de la classe.

- Une classe peut avoir plusieurs constructeurs (comme on peut voir plusieurs méthodes du même nom), mais ils ne peuvent pas avoir la même signature. On peut donc avoir plusieurs constructeurs ayant des paramètres différents.
- En Java il existe un constructeur **par défaut** (sans paramètre) *implicite* (qui n'est pas visible). Ce constructeur par défaut implicite est déclaré par Java *si aucun autre constructeur n'est déclaré dans la classe*. Ce constructeur implicite par défaut initialise les attributs à leur valeur par défaut.
- On peut redéfinir le comportement du constructeur par défaut en explicitant son code.
- Le nombre de constructeurs n'est pas limité (mais attention de ne pas en faire trop !)
- Dans un constructeur on peut faire appel à un autre constructeur de la classe, **mais** on ne peut le faire **que** sur la première ligne de code du constructeur. Pour cela on utilise *l'autoréférence this(...)*.



La valeur par défaut des références est **null**.

Si l'on ne fait pas attention, et que les attributs de type référence ne sont pas initialisés correctement (donc instanciés **avant** l'utilisation), on va avoir à l'exécution des **Null Pointer Exception**.

5.g L'autoréférence

En Java, le mot clé pour l'autoréférence est **this**. Pour appeler un constructeur de la même classe, on utilise **this(...)**. Pour accéder à un attribut ou une méthode, on peut également utiliser **this**, par exemple :

```

1 // Dans la classe Complexe
2 public void ajouter(Complexe c) {
3     this.ajouter(c.r,c.i);
4 }
5 public void ajouter(double r, double i) {
6     this.r += r; // l'attribut r est incrémenté de la valeur du paramètre r
7     this.i += i; // l'attribut i est incrémenté de la valeur du paramètre i
8 }
```

L'utilisation de **this** permet de simplifier/de clarifier les noms des paramètres en levant l'ambiguïté pour désigner les attributs et les distinguer des paramètres. Il vaut mieux utiliser **this** que dans le cas où il y a une ambiguïté.

6 Le modificateur final

Le mot clé **final** permet de transformer des variables en constantes. Par défaut, un attribut est variable, c'est-à-dire qu'il est possible de l'affecter (le modifier) dynamiquement (lors de l'exécution). Il peut être rendu constant en faisant précéder sa déclaration du mot clé **final**. Par exemple:

```

1 public class Product {
2     private final String nomEntreprise = "World Compagny";
3     public final double tauxTVA;
4
5     public Product() {
6         tauxTVA = 19.6;
7     }
8 }

```

Les attributs constants doivent être initialisés une seule fois, soit lors de la déclaration (c'est le cas de l'attribut **nomEntreprise** dans l'exemple ci-dessus), soit dans le constructeur (c'est le cas de **tauxTVA** dans l'exemple ci-dessus).



Un attribut constant peut être déclaré public. En effet, puisqu'il est impossible à modifier, il n'y a pas de risque sur la cohérence de la classe à le rendre public. C'est d'ailleurs le cas de l'attribut **length** des tableaux.

Le mot clé **final** peut aussi être utilisé dans les méthodes pour transformer en constante une variable ou un paramètre.

Note



Comme vu dans le chapitre A.V: Constantes et énumérations, les constantes sont généralement déclarées en majuscule.

Exemple:

```

1 public class ExModFinal {
2     public final int ATT_01 = 1; // attribut constant
3     public final int ATT_02;      // un autre attribut constant
4     public ExModFinal() {
5         ATT_02 = 2; // Ici, ac2 est initialisé dans le constructeur
                     // Il ne peut donc pas être initialisé à la déclaration
6     }
7     public void methode(final int pc) { // paramètre constant
8         final int VAR = 3; // variable constante...
9     }
10 }

```

Note



En java, le mot clé **final** est rarement utilisé pour les attributs contrairement à d'autres langages. En effet, les paramètres étant passés par copie en Java, une modification de ces paramètres à l'intérieur



Dans le cas où le mot clé **final** porte sur un paramètre de type référence, c'est la référence vers l'objet qui ne peut pas être modifiée, pas l'objet lui-même.

exemple: On considère la classe A suivante:

```

1 public class A {
2     private int attr;
3     public A(int attr) {
4         this.attr = attr;
5     }
6     public void setAttr(int val) {
7         this.attr = val;
8     }
9     public String toString() {
10        return "A: " + attr;
11    }
12 }
```

Ainsi que la classe ExempleFinal

```

1 public class ExempleFinal {
2     public final int ATT_01;
3
4     public ExempleFinal(int param) {
5         ATT_01 = param;
6     }
7
8     public void methode(final A b) {
9         a.setAttr(ATT_01);
10    }
11 }
```

et la méthode main:

```

1 public static void main(String[] args) {
2     ExempleFinal ex = new ExempleFinal(5);
3     A a = new A(2);
4     System.out.println(a);
5     ex.methode(a);
6     System.out.println(a);
7 }
```

Ici, la référence du paramètre **b** de la méthode **methode** est passé de manière constante (i.e. avec le mot clé **final**). Cependant, la méthode **methode** modifie l'instance référencée par **b**, comme illustré figure 14.2.

En effet, à la ligne 4 du **main**, le programme affiche: **A: 2** comme illustré figure 14.2 (a). A l'appel de la méthode **methode** sur l'instance référencée par **ex**, le paramètre de type **A** est passé de manière constant (figure 14.3 (a)). Or cette méthode fait appel à la méthode **setAttr** sur l'instance référencée par ce paramètre (figure 14.3 (b)) qui modifie l'attribut de cette instance (figure 14.3 (c)). Ainsi, l'instance référencée par le paramètre constant **a** changé (figure 14.2 (b)) et à la ligne 6 du **main**, le programme affiche **A: 5**.

7 Les objets immuables

Note



Cette section est largement inspirée de la page <https://gfx.developpez.com/tutoriel/java/immutables/>

7.a Définition et intérêt

Définition



Un objet dit immuable est une instance de classe dont les membres exportés (visibles, que cela soit par un modificateur d'accès direct ou indirect **protected**, **public** ou **package**) ne peuvent pas être modifiés après création

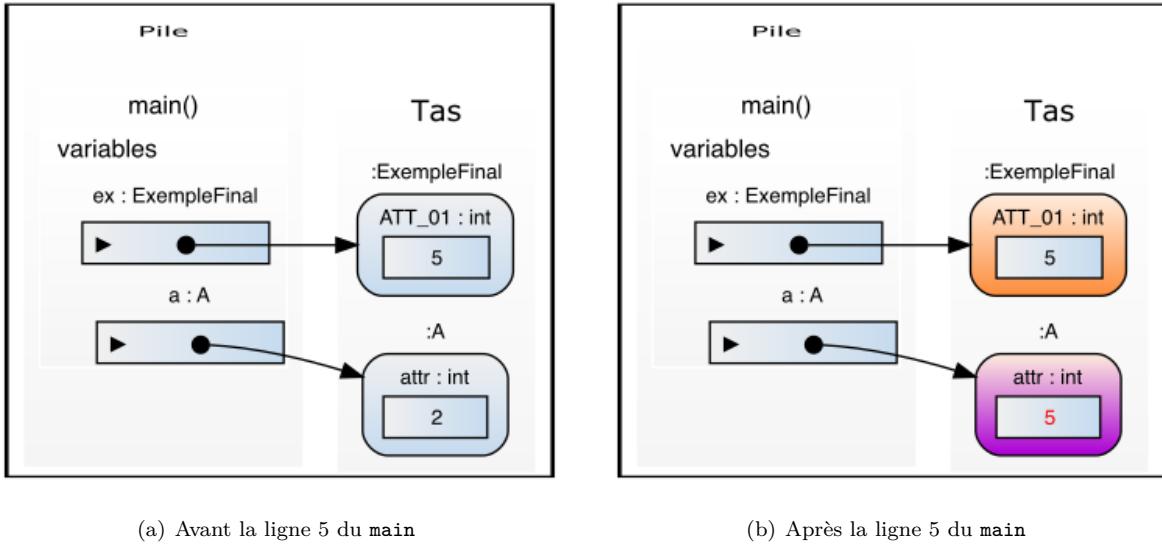


Figure 14.2: Déroulement du programme ci-dessus. l'instance référencé par un paramètre `final` a été modifiée

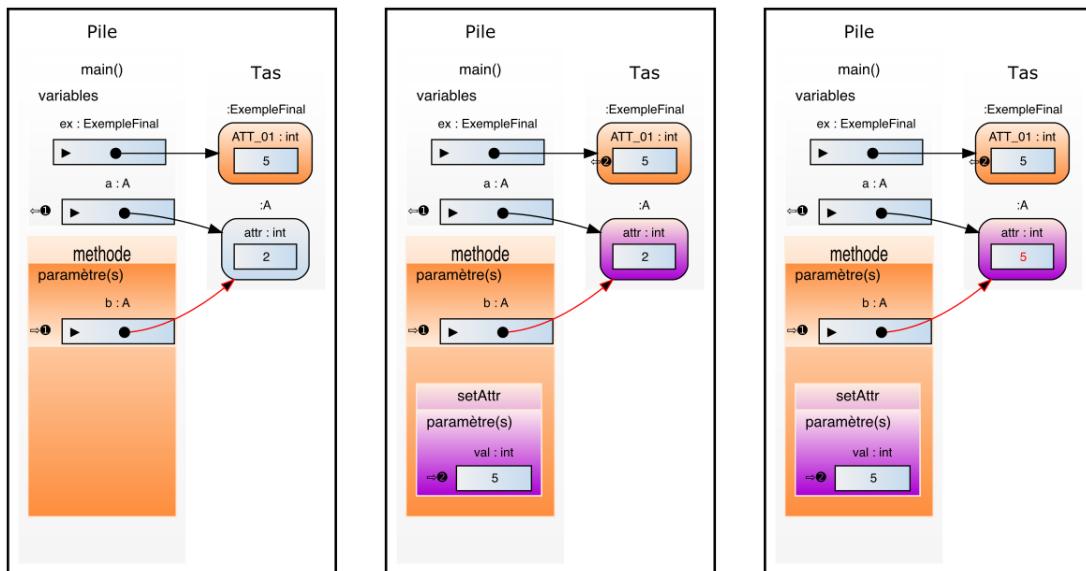


Figure 14.3: Déroulement du programme ci-dessus. Détail

Les classes immuables⁵ ont de nombreux avantages en leur faveur et leur utilisation simplifie souvent le développement. De part leur propriété intrinsèque, ces objets ont un seul état. Parmi les avantages, on peut compter:

- ils sont *thread-safe*
- ils peuvent être mis en cache

⁵la notion de *classe immuable* est un abus de langage souvent utilisé puisque ce sont les instances qui ne changent pas d'état.

- ils n'ont besoin ni de constructeur par copie, ni d'implémentation de l'interface `Cloneable`
- ils constituent d'excellentes clés pour les `Map` et `Set`

Les classes immuables sont particulièrement adaptées à la représentation de types de données abstraits. L'API de Java en contient d'ailleurs plusieurs: `String`, `Integer`, `Color`, etc. La définition de type abstrait dépend toutefois de votre application. Ainsi, un logiciel affichant le contenu d'un magasin en ligne (livres, musique, DVD, etc.) pourra utiliser des classes immuables pour représenter les articles.

7.b Comment créer une classe immuable

Pour créer une classe immuable:

1. la classe doit être déclarée `final`, sans quoi il pourrait être possible de modifier une instance par héritage (cf chapitre suivant)
2. tous les attributs doivent être déclarés `final`
3. la référence à `this` ne doit jamais être exportée
4. tous les champs faisant référence à un objet non immuable doivent être privés, ne jamais être exportés, représenter l'unique référence à cet objet et ne jamais modifier l'état de l'objet.

Note



Dans le cas des attributs de classe immuable, étant donné qu'il y a un attribut par instance, il ne s'agit pas de constante au sens du chapitre A.V, et il n'est donc pas besoin de déclarer cet attribut en majuscule.



Le fait que chaque attribut doive être déclaré `final` implique que la classe ne doit implémenter aucun **modificateur** d'attribut.

Exemple simple

On considère la classe suivante:

```

1 public final class MyImmutable {
2     private final int val;
3
4     public MyImmutable(int val) {
5         this.val = val;
6     }
7
8     public int getVal() {
9         return val;
10    }
11
12    public String toString() {
13        return "Classe MyImmutable, " + val;
14    }
15 }
```

On ne peut alors modifier aucune instance de `MyImmutable`, comme illustré dans le `main` ci-dessous:

```

1 public class TestImmutable {
2     public static void main(String[] args) {
3         MyImmutable immutable = new MyImmutable(5);
4         System.out.println(immutable);
5
6         // Le seul moyen de changer la valeur 5 est de créer une nouvelle
7         // instance...
```

```

7         immutable = new MyImmutable(2);
8         System.out.println(immutable);
9     }
10 }
```

Le diagramme APO du `main` est donné figure 14.4.

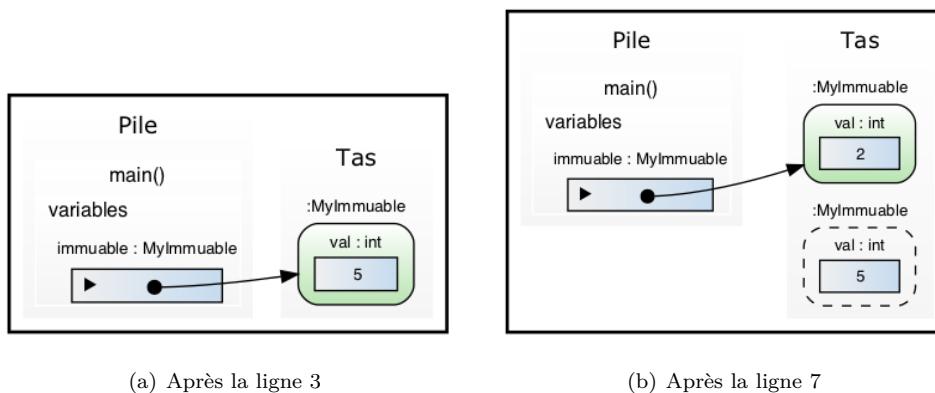


Figure 14.4: Exemple d'objet immuable simple

Exemple illustratif du point 4

On considère à présent les classes suivantes:

```

1 public class A {
2     private int val;
3     public A(int val) {
4         this.val = val;
5     }
6     public int getVal() {
7         return val;
8     }
9     public void setVal(int val) {
10        this.val = val;
11    }
12     public String toString() {
13         return "A: " + val;
14     }
15 }
```

```

1 public final class MyImmutable {
2     private final A a;
3     public MyImmutable(A a) {
4         this.a = a;
5     }
6     public String toString() {
7         return "Classe MyImmutable, " + a.toString();
8     }
9 }
```

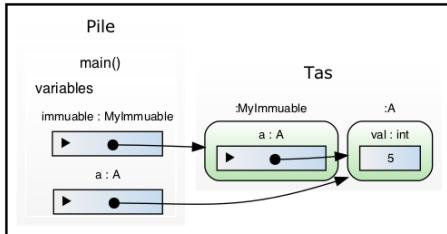
A première vue, cette classe n'exporte jamais l'attribut `a` qui est `final` et `private`. En réalité, dans cet exemple, la classe `MyImmutable` n'est pas immuable. En effet, si l'on considère le `main` ci-dessous et le diagramme APO de la figure 14.5, on constate que l'on peut changer la valeur de l'instance référencée par l'instance immuable.

```

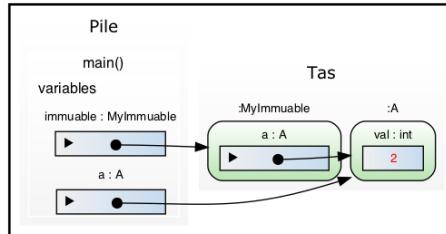
1 public class TestImmutable {
2     public static void main(String[] args) {
3         A a = new A(5);
```

```

4     MyImmutable immutable = new MyImmutable(a);
5     System.out.println(immutable);
6     a.setVal(2);
7     System.out.println(immutable);
8
9 }
10 }
```



(a) Après la ligne 4



(b) Après la ligne 6

Figure 14.5: Exemple d'objet immuable avec un attribut de type référence

Pour résoudre ce problème, la solution est de réaliser une copie défensive des objets non immuables passés en paramètre, comme illustré dans le code suivant et le diagramme APO de la figure 14.6. Cela est possible si la classe non immuable possède un constructeur par copie ou a redéfini la méthode `clone()`.

```

1 public class A {
2     private int val;
3     public A(int val) {
4         this.val = val;
5     }
6     public A(A a) {
7         this.val = a.val;
8     }
9     public int getVal() {
10        return val;
11    }
12    public void setVal(int val) {
13        this.val = val;
14    }
15    public String toString() {
16        return "A: " + val;
17    }
18 }
```

```

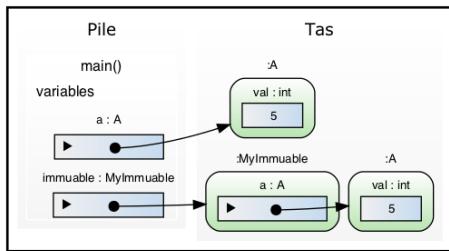
1 public final class MyImmutable {
2     private final A a;
3
4     public MyImmutable(A a) {
5         this.a = new A(a);
6     }
7     public String toString() {
8         return "Classe MyImmutable, " + a.toString();
9     }
10 }
```

```

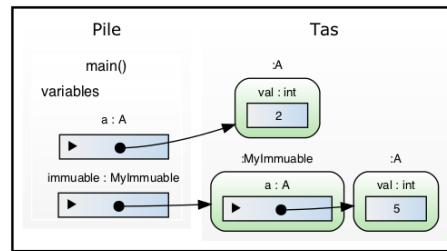
1 public class TestImmutable {
2     public static void main(String[] args) {
3         A a = new A(5);
4         MyImmutable immutable = new MyImmutable(a);
5         System.out.println(immutable);
6         a.setVal(2);
```

```

7     System.out.println(immutable);
8
9 }
10 }
```



(a) Après la ligne 4



(b) Après la ligne 6

Figure 14.6: Exemple d'objet immuable avec un attribut de type référence

Les classes Numériques

Fiche

Prérequis	<ul style="list-style-type: none"> Chapitre 14 Java: La classe !
Objectif pédagogique	Connaître et savoir utiliser les classes numériques de Java à bon escient.
Intérêt	Ces classes sont très utiles pour les collections et la généricité que nous verrons plus tard.
Compétences à acquérir	<ul style="list-style-type: none"> Connaître les classes numériques existantes Comprendre la différence entre un type primitif et une classe numérique Connaître les conversions possibles entre les classes numériques et le types primitifs.

1 Introduction

En Java, il existe des classes qui permettent de représenter des valeurs numériques correspondant aux types primitifs par des instances. Exemples : pour le type primitif `int`, on pourra avoir une instance de type `Integer`, pour le type primitif `double`, on pourra avoir une instance de type `Double`.

Toutes ces classes sont regroupées dans une même famille appelée `Number`

Ces classes servent à :

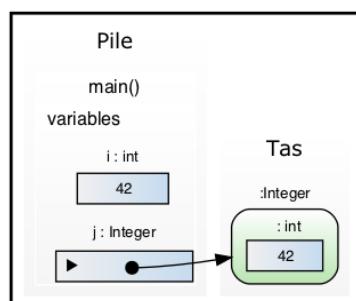
- encapsuler des valeurs primitifs dans des instances
- traduire entre valeurs numériques et chaînes de caractères.

Exemple:

```

1 int i;
2 i = 42;
3
4 Integer j;
5 j = 42;

```



2 La famille Number

Les classes de la famille **Number** permettent :

- l'encapsulation d'un type primitif
- l'utilisation d'une instance pour manipuler un type primitif (exemple: manipulation des collections de données dynamiques contenant des valeurs de type primitif)
- l'accès à des méthodes de transformations/traductions entre les types
- la définition des constantes pour les valeurs limites ou remarquables des types primitifs.

type primitif	classe correspondante dans la famille Number
<code>int</code>	<code>Integer</code>
<code>double</code>	<code>Double</code>
<code>long</code>	<code>Long</code>
<code>byte</code>	<code>Byte</code>



Pour ne pas confondre le type primitif et les classes de la famille **Number** correspondantes, il est important de bien respecter la casse, c'est-à-dire la différence entre les minuscules et les majuscules. Pour rappel, le nom d'un type primitif commence toujours par une minuscule alors que le nom d'un type référence commence toujours par une majuscule...

Pour chaque classe de la famille **Number**, on dispose de toutes les opérations suivantes :

- `+doubleValue(): double`
- `+intValue(): int`
- `+shortValue(): short`
- `+longValue(): long`
- `+byteValue(): byte`

Pour chaque classe de la famille **Number**, on dispose également de constantes :

- `MAX_VALUE`: valeur numérique maximale représentable par le type primitif associé
- `MIN_VALUE`: valeur numérique minimale représentable par le type primitif associé
- `SIZE`: taille mémoire nécessaire pour stocker une valeur du type primitif associé

Dans certaines classes (pour les flottants), on a également les constantes suivantes: `NEGATIVE_INFINITY` et `POSITIVE_INFINITY`.

Les classes `Integer`, `Double`, `Long` et `Byte` ont également un constructeur qui prend une chaîne de caractère `String` en paramètre, ce qui permet de convertir facilement la valeur d'un nombre écrit dans une chaîne de caractères en nombre.

3 Conversion automatique

On parle également d'encapsulation automatique ou boxing (mise en boîte automatique)

Cela n'est disponible que depuis Java 5.

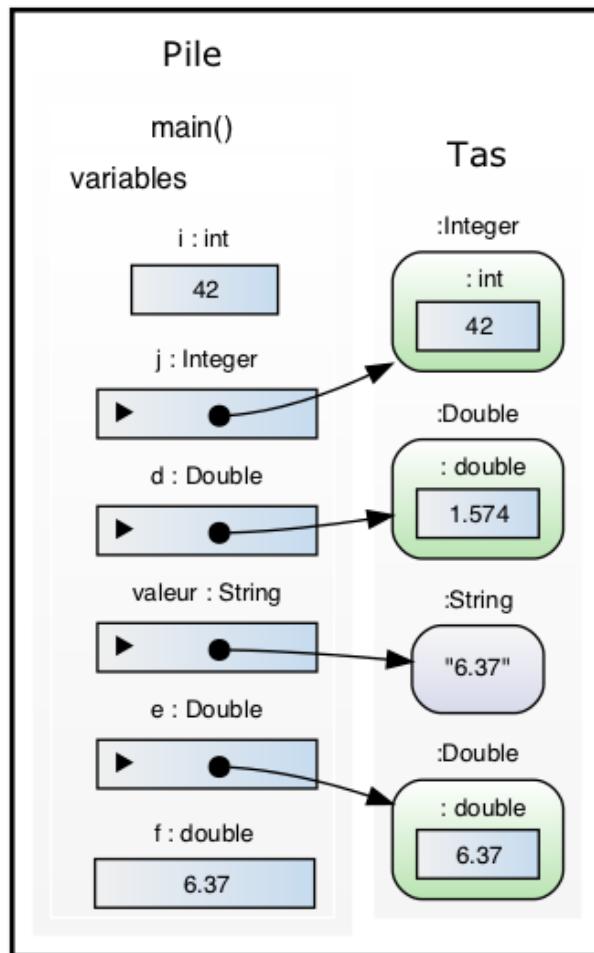
Exemple:

```

1 int i;
2 i = 42;
3
4 Integer j;
5 j = 42; // autoboxing : la valeur 42 est encapsulée dans une instance de la
       classe Integer
6
7 Double d;
8 d = 1.574;
9
10 // conversion depuis String
11 String valeur = "6.37";
12 Double e;
13 e = new Double(valeur); // utilise le constructeur Double(String) : conversion
                           automatique de String vers valeur numérique
14 double f;
15 f = e.doubleValue(); // conversion automatique Double vers double
16 f = Math.sqrt(f) + 1.5;
17 Double g = f;
18
19 if (g.isInfinite()) {
20     ...
21 }

```

Le diagramme APO suivant représente l'état de la mémoire à la fin de la ligne 16.



Vide de famille: Héritage et Polymorphisme

Fiche

<u>Prérequis</u>	<ul style="list-style-type: none">Chapitre 14 Java: La classe !
<u>Objectif pédagogique</u>	Connaître un principe fondamental de la Programmation Orientée Objet: l'héritage.
<u>Intérêt</u>	<ul style="list-style-type: none">connaître toutes les conséquences de l'héritageÊtre capable de décider quand il doit y avoir héritage
<u>Compétences à acquérir</u>	<ul style="list-style-type: none">Connaître le vocabulaire relatif à l'héritageÊtre capable de représenter l'héritage dans un diagramme de classes UMLÊtre capable de reconnaître l'héritage dans un diagramme de classes UMLConnaître les 11 points clés de l'héritage

1 les principes de l'héritage

1.a Introduction

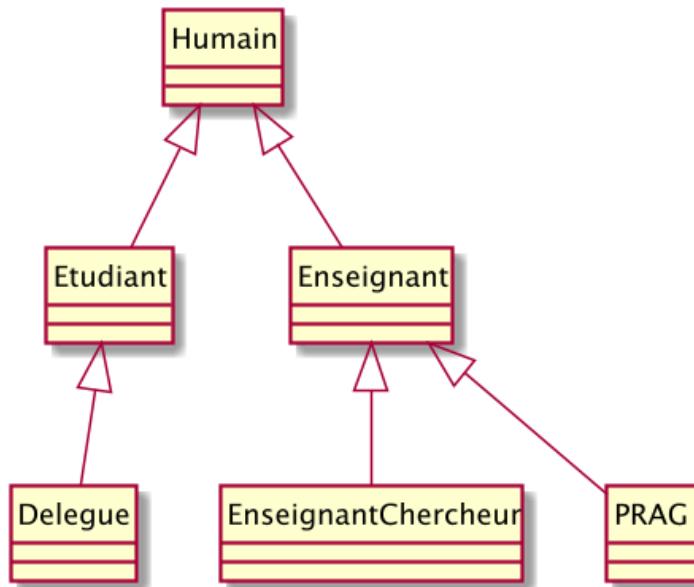
Lorsqu'on modélise un concept (on fabrique une classe), on veut ou doit regrouper plusieurs concepts entre eux : pour des raisons de conception, d'organisation ou de modularité ainsi que de maintenabilité.

Pour regrouper plusieurs concepts :

- on peut utiliser des packages : ensemble de classe qui sont au même endroit.
- on peut créer des familles de classes / concepts.

Exemple : le concept `List` regroupe plusieurs façons de manipuler des listes ordonnées: plusieurs classes.

Une famille de classes représente un concept développé sur plusieurs niveaux d'abstractions.



Le niveau de concept le plus haut est **Humain**. Le niveau de concept le plus précis est celui qui contient **Delegue**, **EnseignantChercheur** et **PRAG**.

Pour indiquer les relations entre les classes, on utilise la flèche triangle vide en UML = relation d'héritage.

Autrement dit :

- les classe **Etudiant** et **Enseignant** héritent de la classe **Humain**
- **Humain** contient tous les concepts communs à **Etudiant** et **Enseignant**

En POO, on utilise *principalement* l'héritage par **extension** : une classe va *ajouter ou modifier* des informations / des comportements de la classe dont elle hérite. (en XML Schema on utilise surtout l'héritage par **restriction**).

Autrement dit :

- les classes **Etudiant** et **Enseignant** étendent la classe **Humain**

1. b Principes généraux

- Il n'existe pas de limite préfixée au nombre de niveaux d'héritage

Note

A partir de 7 ou 8 niveaux d'héritage, ce n'est pas forcément judicieux ou efficaces.



- En Java, l'héritage ne peut se faire que par extension:
 - ajouter des données
 - préciser / spécialiser des concepts
- En Java une classe *ne peut pas hériter de plusieurs classes*. C'est possible dans d'autres langages (comme le C++), mais ce n'est pas une bonne pratique. En effet, nous verrons les interfaces pour répondre aux contrats que peuvent remplir des objets.
- Un membre d'une classe A sera disponible dans les classes qui héritent de A, sauf si le modificateur d'accès est **private** (réutilisation)

1.c Principes de l'héritage par extension

En UML:



En POO on dit :

- B hérite de A
- B étend A
- A est la superclasse de B
- B est une classe dérivée de A
- B est la sous-classe de A
- A est la classe mère de B

B hérite de tous les membres de A (attributs et méthodes) et peut accéder à tous les membres de A non déclarés avec le modificateur d'accès **private**.

Dans la classe B, pour une méthode `+m(..)` ou `m(..)` ou `#m(..)` (mais pas pour `-m(..)`), déclarée dans la classe A, on peut choisir :

- soit de ne rien faire dans la classe B, et le comportement d'une instance de B sera identique au comportement d'une instance de A: B hérite de la méthode `m(..)` et de son algorithme / traitement
- soit modifier le comportement de `m(..)` dans la classe B, c'est-à-dire réécrire `m(..)` (*override* en anglais)
- soit enrichir le comportement de `m(..)` en ajoutant d'autres méthodes `m(.., param)` avec une signature différente, c'est-à-dire en surchargeant la méthode `m(..)` (*overload* en anglais)

Dans la classe B, on peut aussi ajouter des nouvelles méthodes et des nouveaux attributs (enrichir le concept décrit dans A).

La classe B hérite de la classe A, c'est-à-dire qu'une instance de la classe B contient (au moins) les mêmes données et les mêmes comportements (actions/méthodes) qu'une instance A. Une instance de B peut ainsi être vue comme une instance de A. Donc une instance de B peut être vue soit comme une instance de A, (sous une forme générale), soit comme une instance de B (sous une forme spécifique). Une instance de B a donc plusieurs formes. C'est en cela que l'héritage en Java implique le *polymorphisme*.

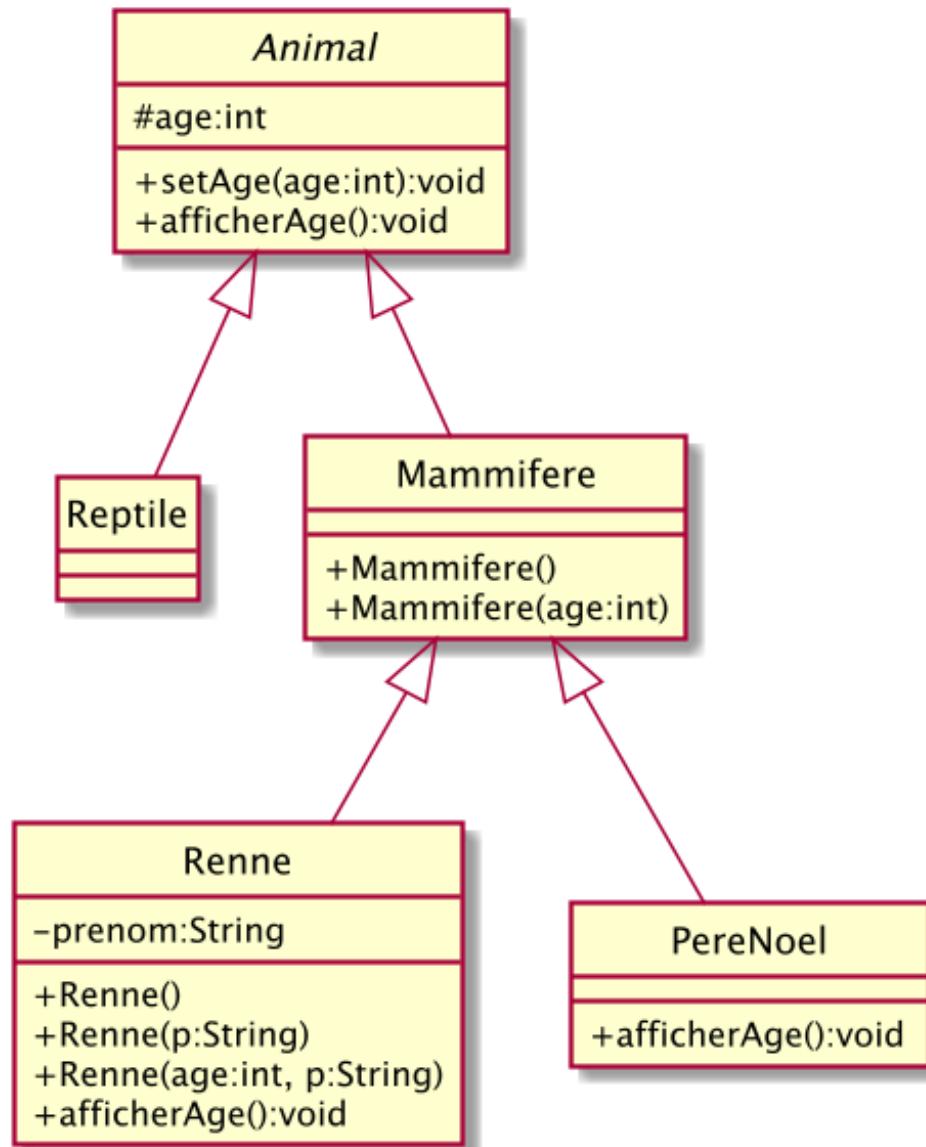
2 Héritage en Java

En Java, pour déclarer qu'une classe dérive d'une autre classe, on utilise le mot clé **extends**.

```

1 class B extends A { // la classe B hérite de la classe A
2
3 }
  
```

2.a Exemple



En Java :

```
1 class Animal {
2     protected int age;
3     public void setAge(int age) {
4         if (age>0) {
5             this.age = age;
6         }
7     }
8     public void afficherAge() {
9         System.out.println("age = " + age + " ans");
10    }m
11 }
12
13 class Reptile extends Animal {
14 }
15
16 class Mammifere extends Animal {
17     public Mammifere() {
18         age = 1;
19     }
20     public Mammifere(int age) {
21         setAge(age);
22     }
23 }
24
25 class Renne extends Mammifere {
26     private String prenom;
27     public Renne() { // implicitement : super()
28     }
29     public Renne(int age, String p) {
30         // setAge(age);
31         super(age);
32         prenom = p;
33     }
34     public Renne(String p) {
35         this(5,p);
36     }
37     public void afficherAge() {
38         super.afficherAge(); // ou directement afficherAge()
39         System.out.println("prenom = " + prenom);
40     }
41 }
42
43 class PereNoel extends Mammifere {
44     public void afficherAge() {
45         System.out.println("confidentiel");
46     }
47 }
48
49 public class Test {
50     public static void main(String[] args) {
51         Reptile s = new Reptile();
52         s.afficherAge();
53         s.setAge(-4);
54         s.afficherAge();
55         s.setAge(3);
56         s.afficherAge();
57 }
```

```

58     Mammifere m = new Mammifere();
59     m.afficherAge();
60     m = new Mammifere(4);
61     m.afficherAge();

62
63     Renne r1 = new Renne();
64     r1.afficherAge();
65     r1 = new Renne("Rodolf");
66     r1.afficherAge();
67     r1 = new Renne(12, "Rodolf père");
68     r1.afficherAge();
69     r1.setAge(42);
70     r1.afficherAge();

71
72     PereNoel p = new PereNoel();
73     p.afficherAge();

74
75     PereNoel p2 = new PereNoel();
76     p2.afficherAge();

77
78     List<Animal> l;
79     l = new ArrayList<>();
80     l.add(p);
81     l.add(r1);
82     l.add(m);
83     for (int i=0; i<l.size();i++) {
84         System.out.println("---- " + i);
85         l.get(i).afficherAge();
86     }

87
88 }
89 }
```

Affiche :

```

1 age = 0 ans
2 age = 0 ans
3 age = 3 ans
4 age = 1 ans
5 age = 4 ans
6 age = 1 ans
7 prenom = null
8 age = 5 ans
9 prenom = Rodolf
10 age = 12 ans
11 prenom = Rodolf père
12 age = 42 ans
13 prenom = Rodolf père
14 confidentiel
15 confidentiel
16 ---- 0
17 confidentiel
18 ---- 1
19 age = 42 ans
20 prenom = Rodolf père
21 ---- 2
22 age = 4 ans
```

3 Les 11 points clés de l'héritage

3.a Point clé n°1 : héritage des membres

Toutes les instances des classes `Reptile`, `Mammifere`, `Renne` et `PereNoel` héritent des méthodes et des attributs de la classe `Animal` :

- l'attribut `age`: `int`
- les deux méthodes `setAge(...)` et `afficherAge()`

3.b Point clé n°2 : héritage à tous les niveaux

Les membres d'une classe sont hérités par toutes les classes dérivées quelque soit leur niveau dans la hiérarchie d'héritage.

Exemple : `setAge(...)` déclaré dans la classe `Animal` est hérité par les classes `Renne` et `PereNoel`.

3.c Point clé n°3 : Redéfinition du comportement

Lorsque le comportement défini par la super classe (ou une des super classe) ne convient pas, on peut réécrire (redéfinir / override) la méthode définissant le comportement. Pour cela il suffit de déclarer une méthode ayant la même signature et de coder le nouveau comportement dans le bloc de la méthode.

Exemples :

- le constructeur par défaut de la classe `Mammifere` redéfini le comportement défini dans `Object`
- la méthode `afficherAge()` de la classe `Renne` et `PereNoel` redéfinissent le comportement défini dans `Animal`

3.d Point clé n°4 : Priorité dans l'héritage

Dans le cas où une méthode est redéfinie, c'est l'implémentation la plus spécifique (la plus précise) qui est utilisée.

Exemple : pour une instance de la classe `PereNoel`, c'est le code de la méthode `afficherAge()` de la classe `PereNoel` qui est utilisée.

3.e Point clé n°5 : non redéfinition

On peut choisir de conserver le même comportement que la superclasse si celui-ci convient. Dans ce cas, il suffit de ne rien faire (ne pas redéfinir la méthode).

Exemple : pour une instance de la classe `Mammifere`, c'est le code de la méthode `afficherAge()` de la classe `Animal` qui est utilisé (en effet, la classe `Mammifere` ne redéfini pas le comportement de la méthode `afficherAge()`).

3.f Point clé n°6 : modificateur d'accès

Un attribut ou une méthode avec le modificateur d'accès `public`, `protected` ou `Ø` est accessible dans toutes les classes dérivées. `private` permet d'interdire l'accès à toutes les autres classes. En conséquence, les méthodes `private` ne peuvent pas être redéfinies dans les classes dérivées.

3.g Point clé n°7 : enrichissement du comportement

On peut enrichir le comportement d'une classe en lui ajoutant des membres qui n'existent pas dans les super classes. Cela peut être de nouveaux attributs (exemple : `prenom` dans la classe `Renne`) ou des méthodes ayant des signatures différentes (exemple : le constructeur `Renne(age:int, p:String)`).

3.h Point clé n°8 : super

Les membres de la super classe (déclaré en `public`, `protected` ou `Ø`) sont accessibles à l'aide du mot clé `super` :

- `super.a` permet d'accéder à l'attribut `a` de la super classe (ou d'une des super classes)
- `super.m(...)` permet d'accéder à la méthode `m(...)` de la super classe
- `super(...)` permet d'accéder à un constructeur de la super classe.

Exemple : l'utilisation de `super(int)` (constructeur de la super classe prenant un `int` en paramètre) à la ligne 31.

A tout moment, on peut faire appel à `super.a` ou `super.m(...)`.



A l'instar de `this(...)`, on ne peut utiliser `super(..)` que à la **première ligne** d'un constructeur.

A l'instar de `this`, `super` est un attribut `private`. En conséquence, `super.super.m(...)` n'est pas possible.

3.i Point clé n°9 : constructeurs

Les constructeurs sont des méthodes comme les autres et respectent les points clés. Tout constructeur qui ne fait pas appel **explicitement** à `super(..)` ou `this(..)` à la **première ligne** de son bloc, appelle implicitement `super()`.

Exemple : le constructeur `Renne()`

3.j Point clé n°10 : le polymorphisme

Une instance d'une classe peut être vue comme une instance de l'une de ses super classes.

Exemple : l'instance référencée par `r1` peut être vue comme une instance de `Renne`, une instance de `Mammifere`, une instance de `Animal` et une instance de `Object`. `r1` est bien polymorphe.



C'est dans l'utilisation (par exemple dans une `ArrayList`) que le polymorphisme est intéressant.

3.k Point clé n°11 : modularité

Dans l'exemple, ligne 78 à 86 (utilisation d'une `ArrayList` pour stocker des instances polymorphes de `Animal`), on utilise la méthode `afficherAge()`. C'est possible car `l` est déclarée comme une liste d'instance de `Animal`, et que `afficherAge()` est déclarée dans `Animal`. Le compilateur est satisfait : il arrive à faire la liaison entre `l.get(i)` qui renvoie une instance de `Animal` et le fait que la classe `Animal` déclare bien une méthode `afficherAge()`.

Lors de l'exécution du programme, la liaison tardive (ou *liaison dynamique*) permet de faire le lien entre chaque instance et la méthode `afficherAge()` spécifique qui lui correspond. Pour `l.get(0).afficherAge()` on obtient `confidentiel`.

L'existence de différents niveaux d'abstraction dans une hiérarchie d'héritage, associé au polymorphisme, permet d'écrire des algorithmes à plusieurs niveaux d'abstraction.

4 Transtypage

Lors de la compilation, Java vérifie la *compatibilité des types* (vocabulaire/grammaire), c'est ce qu'on appelle la *liaison statique* entre une donnée (variable, paramètre et attribut) et un type et ses opérations.

Lors de l'exécution, Java va charger les classes utilisées dans le programme dans la machine virtuelle. Ensuite, lors de l'exécution de chaque instruction, le polymorphisme va pouvoir être utilisé. Cela permet d'écrire des algorithmes qui manipulent/utilisent des méthodes définies dans les classes en haut de la

hiérarchie d'abstraction (exemple : algorithme d'affichage qui utilise la méthode `afficherAge()` définie dans la classe `Animal`).

Cependant dans certains cas, le programmeur.e sait quelles sont les instances qui sont manipulées dans les algorithmes. Il sait que ces instances polymorphes sont issues d'une classe plus spécialisée (de plus bas niveau dans la hiérarchie d'abstraction). Dans ces situations, on veut pouvoir *forcer* le type d'une référence à être considérée directement comme une instance de la classe de bas niveau. Le polymorphisme autorise alors une *liaison tardive* (appelée *liaison dynamique*).

Deux possibilités

- on laisse la liaison dynamique effectuer la liaison entre la donnée et le type bas niveau dont elle est issue (dans exemple, c'est la méthode `afficherAge` des classes `Renne` et `PereNoel` qui sont exécutées)
- on force la liaison entre la donnée et le type bas niveau (qu'on espère être correcte).

```

1 // modification de la classe Renne
2 class Renne extends Mammifere {
3 ...
4 public String getPrenom() {
5     return prenom;
6 }
7 }
8 // modification du programme principal
9 List<Animal> l;
10 l = new ArrayList<>();
11 l.add(p);
12 l.add(r1);
13 l.add(m);
14 for (int i=0; i<l.size();i++) {
15 System.out.println("---- " + i);
16 l.get(i).afficherAge();
17 String p = l.get(i).getPrenom(); // cette ligne génère une erreur de
18     compilation
19 System.out.println(p);
}

```

Dans les cas où on veut **forcer** l'utilisation d'une méthode, il faut prendre des précautions et vérifier au préalable que la donnée est bien une instance de la classe de bas niveau.

Pour **forcer** le type d'une référence, on utilise l'opérateur *cast* : (`TypeForcé`). Exemple :

```

1 double d = 3.14159;
2 int i = (int) d; // force d à être vue comme un int -> i vaut 3
3 double e = 3.999999;
4 int j = (int) e; // j vaut 3 -> transtypage ou forçage (int) ne garde que la
    partie entière,
5 // la programmeur.e sait et comprend ce mécanisme.

```

```

1 // modification du programme principal
2 List<Animal> l;
3 l = new ArrayList<>();
4 l.add(p);
5 l.add(r1);
6 l.add(m);
7 for (int i=0; i<l.size();i++) {
8     System.out.println("---- " + i);
9     l.get(i).afficherAge();
10    Renne rTranstypé = (Renne) l.get(i); // OK pour le compilateur
11    String p = rTranstypé.getPrenom(); // OK pour le compilateur
12    System.out.println(p);
13 }
14 // Lors de l'exécution, l.get(0) est une instance de la classe PereNoel
15 // le transtypage ne va pas fonctionner -> Exception

```

Pour vérifier que le transtypage peut fonctionner et est valide, on utilise le mot clé `instanceof`. `r instanceof T` renvoie `true` si et seulement l'instance référencée par `r` peut être vue comme une instance de `T`.

```

1 // modification du programme principal
2 List<Animal> l;
3 l = new ArrayList<>();
4 l.add(p);
5 l.add(r1);
6 l.add(m);
7 ...
8 System.out.println(l.get(0) instanceof PereNoel); // true
9 System.out.println(l.get(0) instanceof Object); // true
10 System.out.println(l.get(0) instanceof Mammifere); // true
11 System.out.println(l.get(0) instanceof Renne); // false

```

Une autre de façon de vérifier la classe d'une référence est de faire appel à la méthode `getClass()`. `getClass()` renvoie une instance de la classe `Class`, qui permet de connaître le nom de la classe (entre autre).

```

1 // modification du programme principal
2 List<Animal> l;
3 l = new ArrayList<>();
4 l.add(p);
5 l.add(r1);
6 l.add(m);
7 for (int i=0; i<l.size();i++) {
8     System.out.println(l.get(i).getClass().getName());
9 }
10
11 // affichage :
12 PereNoel
13 Renne
14 Mammifere

```

`instanceof` et `getClass()` sont des mécanismes d'introspection fournis par Java. En anglais introspection se dit "reflection".

La classe `Object`

Fiche

<u>Prérequis</u>	<ul style="list-style-type: none"> Chapitre 14 Java: La classe ! Chapitre 16 Héritage et Polymorphisme
<u>Objectif pédagogique</u>	Comprendre qu'en Java, <i>tout est objet !...</i> ...Et ce que cela implique sur les classes que vous écrivez.
<u>Intérêt</u>	Pouvoir adapter le comportement de ses propres classes en ce qui concerne l'affichage et la comparaison.
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> Comprendre ce qui se passe dans un programme lorsque l'on essaie d'afficher une référence Être capable de redéfinir la méthode <code>toString()</code> correctement Comprendre ce qui se passe dans un programme lorsque l'on essaie de comparer 2 références Être capable de redéfinir la méthode <code>equals()</code> correctement.

1 Introduction

En Java, tout est classe !

Dans le langage Java, toutes les références sont des instances de classe. En Java, il n'existe qu'une seule hiérarchie d'héritage. La classe `Object` est en haut de cette hiérarchie.

La classe `Object` est automatiquement (implicitement) superclasse de toute classe ne définissant pas explicitement une relation d'héritage.

Exemple : Considérons la classe `Complexe`:

```

1 public class Complexe { // <- pas de notion d'héritage dans la déclaration
2                         // de la classe Complexe
3                         // -> implicitement la classe Complexe
4                         // hérite de la classe Object
5
6 }
```

le code précédent est équivalent à :

```

1 public class Complexe extends Object {
2
3 }
```

La classe `Object` est fournie avec le langage Java.

2 Quelques méthodes de la classe `Object`

Toutes les classes qu'on a écrite jusqu'à présent héritent donc de la classe `Object`. La classe `Object` définit un certain nombre de comportements. La classe `Object` définit donc un certain nombre de méthodes. Dans plusieurs situations, le comportement par défaut défini dans la classe `Object` n'est pas suffisant pour nos propres classes. Dans ce cas, il suffit de *redéfinir* ce comportement, c'est-à-dire réécrire la méthode correspondante.

On va voir trois méthodes de la classe `Object` qu'il est souvent nécessaire de redéfinir :

- `+toString():String`
- `+equals(r:Object):boolean`
- `+hashCode():int` (utilisé par la collection `HashSet` par exemple)
- `#clone(): Object`

(rappel : # signifie `protected` = accessible dans la classe et dans les classes dérivées de la classe).

2.a Exemple

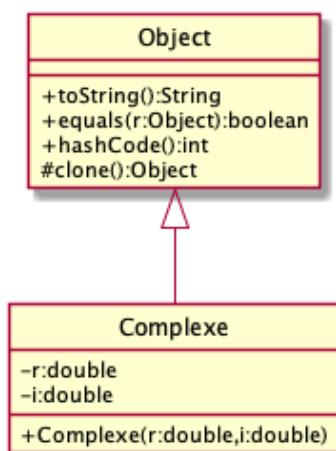
Prenons l'exemple de la classe `Complexe` qui contient `r` et `i` deux attributs de type `double` et un constructeur qui permet d'initialiser ces deux attributs.

```

1 public class Complexe {
2
3     double r;
4     double i;
5
6     public Complexe(double r, double i) {
7         this.r = r;
8         this.i = i;
9     }
10 }
```

Puisqu'aucun héritage n'est défini explicitement dans la déclaration de la classe `Complexe`, Java ajoute implicitement (automatiquement et en silence) la super classe `Object`.

On a en fait :



2.b +**toString():String**

La méthode **toString():String** de la classe **Object** renvoie une représentation d'une instance sous forme de chaîne de caractères. En général, on s'attend à ce que la méthode **toString()** renvoie une chaîne qui doit permettre d'afficher une information succinct mais représentant bien l'état de l'instance soit pour un affichage sur la sortie texte standard (en utilisant **System.out.println**), soit pour un affichage dans une interface graphique. Cet représentation est destinée aux humains (et non à la machine) et doit être représentative et lisible.

La classe **Object** implémente la méthode **toString()** mais ne fournit pas une représentation très lisible :

```

1 Complexe a = new Complexe(1.7, 2.4);
2
3 // Utilisation implicite lors de la transformation d'une instance en chaîne de
   caractères
4 // par exemple par l'opérateur de concaténation :
5 System.out.println("La méthode toString() de la classe Object affiche par défaut : " + a);
6 // Ce qui est équivalent à faire :
7 System.out.println("La méthode toString() de la classe Object affiche par défaut : " + a.toString());

```

Affiche :

```

1 La méthode toString() de la classe Object affiche par défaut :
2 nompackage.Complexe@4aa298b7
3 La méthode toString() de la classe Object affiche par défaut :
4 nompackage.Complexe@4aa298b7

```

La chaîne par défaut est le nom de la classe (incluant le nom de package), puis le caractère @, suivi enfin du hash code produit par la méthode **hashCode** sous forme hexadécimale (base 16 donc utilisant les chiffres de 0 à 9 et les lettres de a à f, comme **hashCode** renvoie un **int**, il y a toujours 8 caractères hexadécimaux).

Ceci n'est pas très lisible et ne permet généralement pas de comprendre l'état de l'instance. Il est donc recommandé de réécrire le comportement dans les classes que l'on veut afficher.

Par exemple :

```

1 // Réécriture de la méthode toString dans la classe Complexe
2 @Override
3 public String toString() {
4     return (r + " + " + i + "i");
5 }

```

et le code :

```

1 System.out.println("Le complexe est : " + a.toString());
2 // Ce qui est équivalent à faire :
3 System.out.println("Le complexe est : " + a);

```

on obtient :

```

1 Le complexe est : 1.7 + 2.4i
2 Le complexe est : 1.7 + 2.4i

```

2.c +equals(r:Object):boolean

Elle vérifie "l'égalité" entre deux instances.



Il ne faut pas confondre égalité entre deux instances et égalité des références.

Par défaut, dans la classe `Object`, `a.equals(b)` renvoie `true` si et seulement si :

- les instances référencées par `a` et par `b` sont issues de la même classe
- les comparaisons par `equals` des attributs un à un renvoient `true`

... Eh non ! en fait, Par défaut, dans la classe `Object`, `a.equals(b)` renvoie `true` si et seulement si `a == b`. Elle a donc le même comportement que l'utilisation de `==`, ce qui peut être problématique lorsqu'on utilise l'API Collection (notamment pour tester la présence d'un élément, récupérer sa position, l'effacer, l'insérer dans un `Set`...)

Ce comportement de la méthode `equals(..)` n'est pas satisfaisant pour :

- les chaînes de caractères. C'est pourquoi la classe `String` redéfinit la méthode `equals(...)` de façon à renvoyer `true` lorsque les 2 chaînes de caractères sont les mêmes
- de manière générale, dans les classes que l'on crée nous-même, il est souvent important/nécessaire de redéfinir la méthode `equals(...)`.

Exemple: on considère à nouveau la classe `Complexe`:

Complexe
-r:double
-i:double
+Complexe(r:double,i:double)
+toString():String

```

1 Complexe a = new Complexe(1.7, 2.4);
2 Complexe b = new Complexe(-11.2, 0.0);
3 Complexe c = new Complexe(1.7, 2.4);
4 String s = "toto";

5
6 System.out.println("a.equals(b) = " + a.equals(b)); // false
7
8 System.out.println("a.equals(s) = " + a.equals(s)); // false
9
10 System.out.println("(a == c) = " + (a == c)); // false
11
12 System.out.println("a.equals(c) = " + a.equals(c)); // false
13
14 List<Complexe> l = new ArrayList<>();
15 l.add(a);
16 if (l.contains(a)) {
17     System.out.println("l contient le complexe " + a);
18 }
19
20 if (l.contains(c)) {
21     System.out.println("l contient bien le complexe " + c );
22 }
23 else {
24     System.out.println("mais l ne contient pas le complexe " + c );
25 }

// Affiche :
// l contient le complexe 1.7 + 2.4i
// mais l ne contient pas le complexe 1.7 + 2.4i

```

Note

Lorsqu'on réécrit la méthode `equals` il vaut mieux réécrire également la méthode `hashCode():int`. Ce n'est pas obligatoire, mais c'est une bonne pratique. En effet, l'API de la classe 'Object' indique que "If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result."

```

1 public class Complexe {
2
3     double r;
4     double i;
5
6     public Complexe(double r, double i) {
7         this.r = r;
8         this.i = i;
9     }
10
11    @Override
12    public String toString() {
13        return (r + " + " + i + "i");
14    }
15
16    @Override
17    public boolean equals(Object obj) {
18        if (obj == this) {
19            return true;
20        } else if (!(obj instanceof Complexe)) {
21            return false;
22        } else {
23            Complexe c = (Complexe) obj;
24            return (this.r == c.r && this.i == c.i);
25        }
26    }
27
28    @Override
29    public int hashCode() {
30        ...
31    }
32 }
```

```

1 // Avec la nouvelle classe Complexe qui réécrit equals...
2 Complexe d = new Complexe(1.7, 2.4);
3 Complexe e = new Complexe(1.7, 2.4);
4
5 System.out.println("(d == e) = " + (d == e)); // false
6
7 System.out.println("d.equals(e) = " + d.equals(e)); // true
8
9 List<Complexe> l = new ArrayList<>();
10 l.add(d);
11 if (l.contains(d)) {
12     System.out.println("l contient le complexe " + d);
13 }
14
15 if (l.contains(e)) {
16     System.out.println("l contient bien le complexe " + e );
17 }
18 else {
19     System.out.println("mais l ne contient pas le complexe " + e );
20 }
```

Ce nouveau code affiche bien :

```
1 contient le complexe 1.7 + 2.4i
2 contient bien le complexe 1.7 + 2.4i
```

2.d +hashCode(): int

Définition



La méthode `hashCode()` *digère* les données stockées dans une instance de la classe dans une valeur de hachage (en un entier signé 32-bit). Cette valeur de hachage est utilisée par d'autres codes lors du stockage ou de la manipulation de l'instance - les valeurs visent à être réparties de manière homogène pour différentes entrées de manière à être utilisées en aggrégation.

La méthode `+hashCode(): int` est utile lorsque l'on souhaite retrouver rapidement une instance particulière. C'est le cas par exemple des collections qui, pour rechercher une instance particulière va utiliser la méthode `+hashCode(): int` plutôt que `equals` pour aller plus vite. En effet, un premier algorithme va récupérer le `hashCode` sur chaque instance et classer les instances dans une structure de donnée particulière (certains tris sur les entiers sont particulièrement efficace). La recherche d'une instance donnée va alors se faire selon son `hashCode` au lieu d'une comparaison via la méthode `equals()`. Ainsi, il est vraiment très important que les méthodes `equals` et `hashCode` renvoient des résultats équivalents c'est-à-dire que 2 instances qui seront égales selon la méthode `equals` devront avoir le même `hashCode` et réciproquement.

On peut ainsi voir la méthode `+hashCode(): int` comme une méthode renvoyant un résumé de l'état de l'instance sous forme numérique. Pour éviter un maximum une collision entre les instances, nous allons utiliser la technique suivante pour rédiger la méthode `+hashCode(): int` :

- initialiser le résultat avec un nombre premier
- ajouter le `hashCode` de chaque attribut s'il est non-null en le multipliant par un nombre premier

Exemple:

```
1 @Override
2 public int hashCode() {
3     // Good practice from from effective Java : Item 9
4     // usint two prime numbers
5     int result = 17;
6     result = 31 * result + new Double(r).hashCode();
7     result = 31 * result + new Double(i).hashCode();
8     return result;
9 }
```

Classes Abstraites

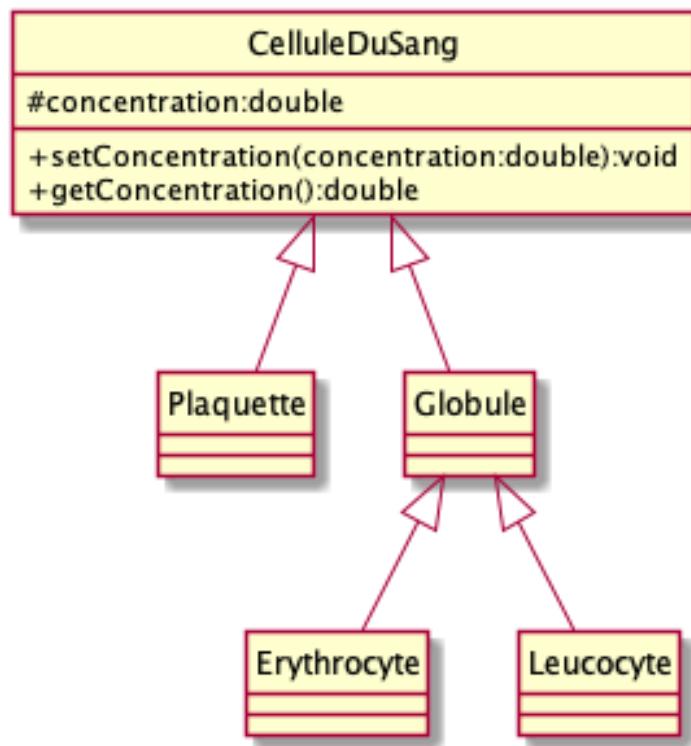
Fiche

<u>Prérequis</u>	<ul style="list-style-type: none">Chapitre 14 Java: La classe !Chapitre 16 Héritage et Polymorphisme
<u>Objectif pédagogique</u>	Connaître le fonctionnement et les intérêts des classes abstraites.
<u>Intérêt</u>	Utiliser et implémenter des classes abstraites
<u>Compétences à acquérir</u>	<ul style="list-style-type: none">Être capable de définir une classe abstraite dans un diagramme UML de classeÊtre capable d'écrire une classe abstraite en JavaÊtre capable d'utiliser une classe abstraite dans un programme Java.

1 Introduction

Parfois, lorsque l'on construit une architecture d'héritage de classes, on sait que les sous-classes vont toutes avoir un type de comportement commun, sans être capable de définir ce comportement dans la super-classe.

Par exemple, si l'on se place dans le cadre d'analyses sanguines, que l'on souhaite modéliser des plaquettes, des globules blancs (lymphocytes) et de globules rouges (leucocytes), on pourra avoir ce type de hiérarchie de classes:



Lors d'un contrôle sanguin, il existe, pour chaque type de cellule, une concentration considérée comme normale, et un seuil au delà duquel, un courrier doit être envoyé au médecin.

Pour cela, on souhaite ajouter une méthode `+seuilOK(): boolean` dans chacune des classes `Plaquette`, `Erythrocyte` et `Leucocyte`. De plus, même si l'on ne connaît pas le seuil en général pour `Globule` ni pour `CelluleDuSang`, c'est-à-dire que l'on ne peut pas implémenter la méthode `+seuilOK(): boolean` dans ces classes, on sait que toutes les classes spécifiques qui héritent de `CelluleDuSang` et de `Globule` sauront implémenter ces classes. C'est-à-dire que toute cellule du sang doit avoir une méthode `+seuilOK(): boolean`, et que l'implémentation de cette méthode est spécifique aux sous-classes.

Pour cela, on va définir des méthodes **abstraites**.

Définition



Une méthode abstraite est une méthode qui ne contient pas de code.

En pratique, on pourra indiquer dans la classe `CelluleDuSang` une méthode **abstraite** `seuilOK`:

```
1 public abstract boolean seuilOK();
```

et définir / implémenter cette méthode dans les sous-classes, c'est-à-dire que l'on aura dans la classe `Plaquette`,

```
1 public boolean seuilOK() {
2     // Dosage moyen de plaquettes pour un adulte: 160.000 - 350.000 /mm3
3     return (concentration > 160000) && (concentration < 35000);
4 }
```

dans la classe `Leucocyte`

```
1 public boolean seuilOK() {
2     // Dosage moyen de leucocytes pour un adulte: 4000 à 10500/mm3
3     return (concentration > 4000) && (concentration < 10500);
4 }
```

et dans la classe `Lymphocyte`

```
1 public boolean seuilOK() {
```

```

1 // Dosage moyen de lymphocytes pour un adulte: entre 4 et 5,6 millions par
2   mm3
3   return = ((concentration > 4000000) && (concentration < 5600000));
4 }
```

On remarque alors que l'on risque d'avoir des incohérences sur la classe `CelluleDuSang`. En effet, si l'on crée une instance de type `CelluleDuSang`, on ne pourra pas appeler toutes les méthodes déclarées dans la classe `CelluleDuSang` dans cette instance. En effet, le code suivant:

```

1 public static void main(String[] args) {
2     CelluleDuSang cells = new CelluleDuSang();
3     if(! cells.seuilOK()) {
4         prevenirMedecin();
5     }
6 }
```

Ici, la méthode `seuilOK()` ne peut pas renvoyer `true` ou `false` puisqu'elle n'est pas définie !

On ne peut donc pas appeler des méthodes abstraites sur l'instance d'une classe qui contient des méthodes abstraites. C'est pourquoi, lorsqu'une classe contiendra au moins une méthode abstraite, on parlera de **classe abstraite**.

Définition



Une classe est dite abstraite dès qu'elle contient une (ou plusieurs) méthode abstraite.

Une classe abstraite n'est pas instanciable :

- on ne peut pas créer d'instance manipulable
- on ne peut pas utiliser l'opérateur `new` sur un de ses constructeurs

Grâce au polymorphisme, on peut utiliser dans un programme des références à cette classe. On utilise les classes abstraites pour modéliser des concepts abstraits dont certains comportements sont déjà définis et certains autres manquent ou sont incomplets. Une classe abstraite va permettre de spécifier des membres communs à toute une hiérarchie de classe.



Avantages:

1. une classe abstraite va permettre d'écrire des algorithmes de haut niveau d'abstraction
2. une classe abstraite va pouvoir garantir l'existence de certains comportements *sans les définir*. A charge aux classes dérivées de le faire. Pour cela on va pouvoir définir dans les classes abstraites des méthodes abstraites. Une méthode abstraite est une méthode dont on déclare la *signature* sans fournir le code. Une classe dérivée qui se veut instanciable doit **obligatoirement** avoir un code pour toutes les méthodes abstraites.
3. Grâce au polymorphisme, on peut gérer des collections ou des références du type de la classe abstraite.

2 Classe abstraite en UML

En UML, le nom d'une classe abstraite est écrit en *italique*.

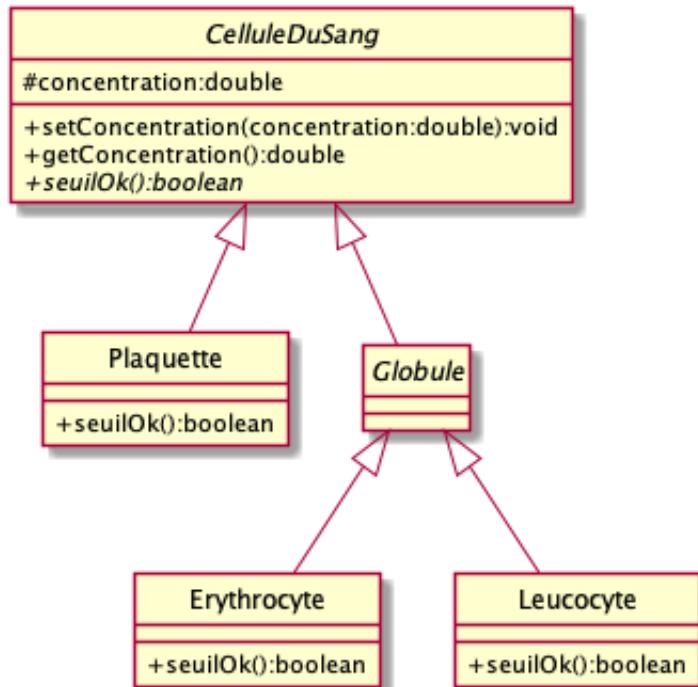
3 Classe et méthode abstraite en Java

En Java, on utilise le mot clé **abstract**. C'est un modificateur utilisable sur les classes (pour déclarer une classe abstraite) ou sur les méthodes (pour forcer une classe dérivée instanciable à fournir une implémentation).

La déclaration d'une méthode abstraite dans une classe a plusieurs conséquences :

- une classe dans laquelle une méthode abstraite est déclarée **doit** être abstraite
- une classe dérivée de la classe dans laquelle cette méthode est définie doit soit:
 - implémenter cette méthode (fournir le code pour le comportement)
 - être elle-même une classe abstraite

4 Exemple



```

1  public abstract class CelluleDuSang {
2      ...
3      public abstract boolean seuilOk();
4
5  }
6  public class Plaquette extends CelluleDuSang {
7      ...
8      public boolean seuilOk() {
9          ...
10     }
11 }
```



Dans cet exemple, la classe *Globule* est également abstraite (en italique dans le diagramme de classes ci-dessus) car même si elle ne déclare pas de méthode `+seuilOK(): boolean`, elle contient cette méthode car elle hérite de *CelluleDuSang*. Comme elle ne la définit/ ne l'implémente pas, elle contient une méthode abstraite. Il s'agit donc d'une classe abstraite.

Interfaces

Fiche

<u>Prérequis</u>	<ul style="list-style-type: none"> Chapitre 14 Java: La classe ! Chapitre 16 Héritage et Polymorphisme Chapitre 18 Classes Abstraites
<u>Objectif pédagogique</u>	Connaître le fonctionnement et les intérêts des interfaces.
<u>Intérêt</u>	Utiliser et implémenter des interfaces
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> Être capable de définir une interface dans un diagramme UML de classe Être capable d'écrire une interface en Java Être capable d'utiliser une interface dans un programme Java.



Ici le mot interface est utilisé au sens de concept dans la programmation orientée objet. On ne parle pas d'interface utilisateur (cours IHM) / GUI (Graphical User Interface), ni d'interface de programmation (API).

1 Définition

Les interfaces permettent de *définir* une liste de comportements *abstraits*. Une interface est :

- une classe abstraite
- sans attribut
- où toutes les méthodes sont abstraites
- qui n'a pas de constructeur

Une interface permet de définir une liste d'actions possibles. Une interface doit être vue comme un contrat.

Une interface ne peut pas hériter d'une classe. Une interface est **implémentée** par une classe. On dit d'une classe qui **implémente** une interface qu'elle remplit le contrat défini par l'interface.

Une classe peut implémenter plusieurs interfaces (Rappel : en Java l'héritage multiple entre classe est interdit, c'est-à-dire qu'une classe ne peut *dériver* que d'une seule autre classe).



Avec la hiérarchie d'héritage on définit le comportement "je suis une sorte de...".
Avec les interfaces (et les hiérarchies d'interface), on définit les capacités d'action "je suis capable de...".

Un des avantages des interfaces est la séparation complète avec les hiérarchies d'héritage. Une interface a souvent un nom finissant en "able" (comparable, dessinable...) (notez que le mot "able" en anglais est tout à fait approprié).



Dans les versions de Java ≥ 8 , on peut fournir des implémentations par défaut d'une méthode dans les interfaces. Attention cependant à réserver cela pour garantir la rétro compatibilité d'une API et de ne pas l'utiliser lors de la création de nouvelles interfaces^a.

^aCette possibilité a été définie en Java pour ajouter l'API des Streams aux classes de l'API Collection, tout en garantissant la compatibilité avec les programmes déjà écrits. Cette possibilité introduit, dans les faits, la possibilité d'héritage multiple jusqu'à là impossible en Java. L'héritage multiple existe dans d'autres langages, mais implique souvent une complexité importante génératrice de bugs complexes.

2 Exemple : l'interface Comparable

L'interface `Comparable` est définie dans le langage Java. Elle définit un contrat qui indique la présence de la méthode `+compareTo(r:Object):int`. Cette méthode permet de comparer une instance à n'importe quelle autre instance.

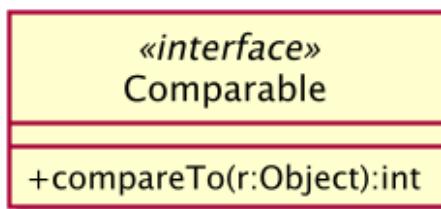
Elle renvoie un `int` :

- <0 si l'instance est "plus petite" que l'instance passée en paramètre
- ==0 si l'instance est identique à l'instance passée en paramètre
- >0 si l'instance est "plus grande" que l'instance passée en paramètre

Une classe qui implémente l'interface `Comparable` doit remplir le contrat et implémenter la méthode `compareTo` avec sa signature exacte. Pour éviter les comparaisons entre objets de classes incompatible, l'interface `Comparable` utilise la généricité (omis ici pour des raisons de simplification).

2.a En UML

En UML, on utilise le stéréotype «interface»:



2.b En Java

En Java, on utilise le mot clé `interface` qui remplace le mot clé `class` :

```

1 public interface Comparable {
2     public int compareTo(Object r);
3 }
```

Note

Le mot clé `abstract` n'est pas utilisé dans les interfaces : une méthode déclarée dans une `interface` est forcément abstraite.

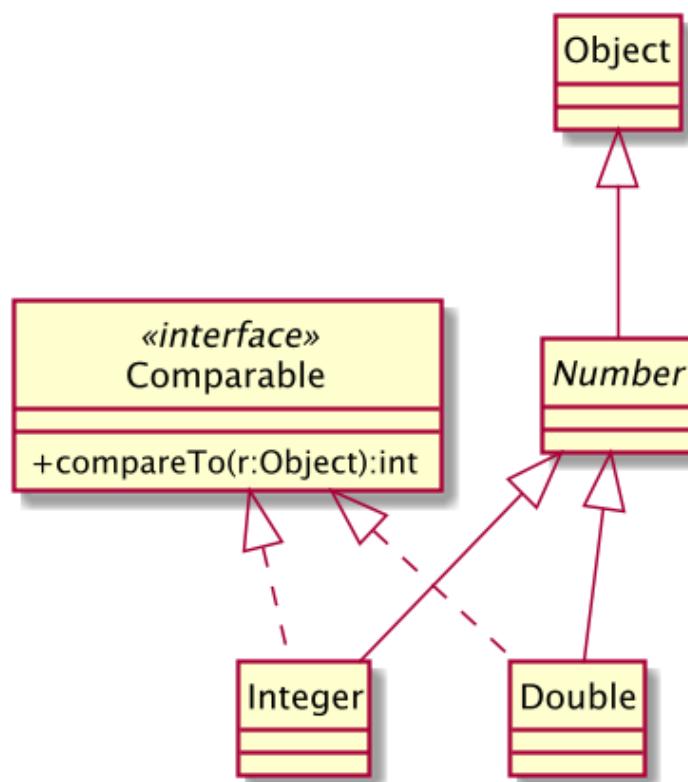
3 Exemple d'utilisation de l'interface Comparable

On peut maintenant mieux comprendre la hiérarchie d'héritage de la famille `Number` de Java. Il faut savoir également que :

- la classe `Number` est une classe abstraite.
- les classes `Integer` et `Double` implémentent l'interface `Comparable`

3.a En UML

Tout ceci apparait clairement dans le diagramme UML ci-dessous :



Notez la relation spécifique avec `Comparable` (flèche d'héritage mais avec des traits en pointillés).

3.b En Java

En Java, on utilise le mot clé `implements`:

```

1 public class Integer implements Comparable {
2     ...
3     public int compareTo(..) {
4         ...
5     }
6 }
  
```

4 Interface et polymorphisme

Une instance d'un classe C implémentant une interface I, peut être vue/manipulée comme si elle était une instance de I. On va donc pouvoir écrire des algorithmes qui utilisent des données de type I ou stocker des collection de références de I.

Le polymorphisme inclut donc les relations d'implémentation et d'héritage d'interface. Cela permet d'écrire du code où les données manipulées sont considérées à un haut niveau d'abstraction. Par exemple, le code suivant est valide :

```

1 Integer i = 10;
2 Double d = 11.0;
3 List<Comparable> l = new ArrayList<>();
4 // (a) i est une référence sur une instance de Integer
5 // (b) Integer implémente Comparable
6 // (a) et (b) => i est vu comme un Comparable
7 l.add(i);
8 // d est polymorphe également et peut être vu comme un Comparable
9 l.add(d);
10
11 // Notez au passage qu'en fait List est elle-même une interface !

```

5 Hiérarchie d'interfaces et générericité

Note



Ce paragraphe est trop avancé pour le cours d'aujourd'hui. Il sera à revoir dès que l'on aura vu la générericité.

Il est possible de créer des interfaces qui héritent d'autres interfaces (on utilise alors le mot clé `extends` entre ces interfaces) afin de créer des hiérarchies d'interfaces. Cela permet de créer des relations de type "contrat" / "sous-contrat".

Il est possible de déclarer des interfaces Java utilisant la générericité. Cela permet de simplifier les interactions et de faire vérifier les compatibilités entre les différentes classes implémentant la même interface directement par le compilateur .

Par exemple, en Java, l'interface `Comparable` est générérique :

```

1 public interface Comparable<T> {
2     public int compareTo(T r);
3 }

```

La classe `Integer` va implémenter seulement la comparaison entre `Integer`, elle sera déclarée ainsi :

```

1 public class Integer implements Comparable<Integer> {
2     ...
3     public int compareTo(Integer r) {
4         ...
5     }
6 }

```



ici T est remplacé par `Integer`, mais on aurait pu utiliser le nom d'une classe plus générale (par exemple `Number`). Dans ce cas on aurait du prendre en compte tous les cas possibles pour la comparaison entre n'importe quel `Number` et un `Integer`

Le principe de la générnicité

Fiche

<u>Prérequis</u>	<ul style="list-style-type: none"> • Chapitre 14 Java: La classe ! • Chapitre 15 Les classes numériques • Chapitre 16 Héritage et Polymorphisme • Chapitre 19 Interfaces
<u>Objectif pédagogique</u>	Connaître un principe fondamental de la Programmation Orientée Objet: la générnicité
<u>Intérêt</u>	<ul style="list-style-type: none"> • Écrire des méthodes réutilisables indépendamment du type de données • Utiliser des méthodes et objets génériques de l'API Java
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> • Être capable d'utiliser à bon escient des méthodes et classes génériques de l'API Java • Être capable de créer des classes et des méthodes génériques

1 Principe

Comme nous l'avons vu dans la partie algorithmique, il existe des algorithmes qui se prêtent à plusieurs types de données. Par exemple, on peut souhaiter qu'un algorithme de tri soit générique, indépendant du fait que l'on trie des entiers, des doubles ou des Cartes... à partir du moment où l'on sait comparer 2 valeurs.

La **générnicité** permet de faire abstraction du type de données manipulées par un algorithmique. Ainsi, les algorithmes pourront fonctionner pour un ensemble de types de données sans avoir à les décliner pour chacun de ces types.

C'est un des principes de la programmation orientée objet, disponible à partir de Java 5.

On appelle une classe "classe générique" si elle contient des méthodes (donc des algorithmes) génériques, c'est-à-dire qui fonctionnent avec des données génériques (non spécifiées directement).

Les données génériques sont forcément de la famille des références.



2 Intérêt

Pour mieux comprendre l'intérêt de la générnicité, prenons un exemple qui ne l'utilise pas.

2.a Exemple avec une classe par type

Prenons l'exemple d'une paire de valeurs (les calculs sur cette classe seront vues plus loin).

```

1 public class PaireEntiers {
2     private int premier;
3     private int deuxieme;
4
5     public PaireEntiers(int premier, int deuxieme) {
6         this.premier = premier;
7         this.deuxieme = deuxieme;
8     }
9
10    public int getPremier() {
11        return premier;
12    }
13
14    public int getDeuxieme() {
15        return deuxieme;
16    }
17
18    // Note: mettre le test directement dans la classe ne respecte pas les Java
19    // coding conventions, mais permet de gagner de la place ici...
20    public static void main(String[] args) {
21        PaireEntiers paire = new PaireEntiers(3, 4);
22        System.out.println("Premier: " + paire.getPremier() + ", Deuxième: " +
23                           paire.getDeuxieme());
24    }
25 }
```

Ensuite, pour une paire de double, on aura la classe suivante:

```

1 public class PaireDouble {
2     private double premier;
3     private double deuxieme;
4
5     public PaireDouble(double premier, double deuxieme) {
6         this.premier = premier;
7         this.deuxieme = deuxieme;
8     }
9
10    public double getPremier() {
11        return premier;
12    }
13
14    public double getDeuxieme() {
15        return deuxieme;
16    }
17
18    // Note: mettre le test directement dans la classe ne respecte pas les Java
19    // coding conventions, mais permet de gagner de la place ici...
20    public static void main(String[] args) {
21        PaireDouble paire = new PaireDouble(3.0, 4.0);
22        System.out.println("Premier: " + paire.getPremier() + ", Deuxième: " +
23                           paire.getDeuxieme());
24    }
25 }
```

Et pour une paire de chaînes de caractères, la même chose:

```

1  public class PaireString {
2      private String premier;
3      private String deuxieme;
4
5      public PaireString(String premier, String deuxieme) {
6          this.premier = premier;
7          this.deuxieme = deuxieme;
8      }
9
10     public String getPremier() {
11         return premier;
12     }
13
14     public String getDeuxieme() {
15         return deuxieme;
16     }
17
18     // Note: mettre le test directement dans la classe ne respecte pas les Java
19     // coding conventions, mais permet de gagner de la place ici...
20     public static void main(String[] args) {
21         PaireString paire = new PaireString("3.0", "4.0");
22         System.out.println("Premier: " + paire.getPremier() + ", Deuxième: " +
23             paire.getDeuxieme());
24     }
25 }
```

Et ainsi de suite... Java est un langage fortement typé, et c'est une bonne chose... Mais ce n'est pas une raison pour ré-écrire le même code.

2.b Exemple avec la classe Object

Un bon réflexe

Une première solution pour éviter de ré-écrire du code est d'utiliser l'héritage. Le seul problème est que les types primitifs (`int`, `double`, `boolean`, etc.) ne peuvent pas utiliser directement l'héritage. C'est pourquoi il existe la classe `Number` en Java de laquelle héritent `Int`, `Double`, `Boolean`, etc. (cf chapitre 15). On va donc utiliser ces classes pour créer notre classe `Paire`.

A partir du moment où l'on ne s'intéresse qu'à des données de la famille des références, et comme tout objet Java hérite de la classe `Object`, on peut définir une classe faussement *générique* `PaireObject` suivante:

```

1  public class PaireObject {
2      private Object premier;
3      private Object deuxieme;
4
5      public PaireObject(Object premier, Object deuxieme) {
6          this.premier = premier;
7          this.deuxieme = deuxieme;
8      }
9
10     public Object getPremier() {
11         return premier;
12     }
13
14     public Object getDeuxieme() {
15         return deuxieme;
16     }
17
18     // Note: mettre le test directement dans la classe ne respecte pas les Java
19     // coding conventions, mais permet de gagner de la place ici...
20     public static void main(String[] args) {
21         PaireObject paire = new PaireObject(new Double(3.0), new Double(4.0));
22     }
23 }
```

```

21     System.out.println("Premier: " + paire.getPremier() + ", Deuxième: " +
22         paire.getDeuxieme());
23 }

```

Note

On notera au passage que le constructeur `new Double(3.0)` à la ligne 21 du code précédent, est déconseillé en Java. Je l'ai laissé dans l'exemple pour souligner le fait que l'on avait affaire à des `Object` et non des types primitifs. On utilisera plutôt `Double.valueOf(3.0)`

Les écueils de cette solution

Dans l'exemple de test précédent, tout se passe comme pour les exemples `int`, `double` et `String`. Considérons à présent le code suivant de la classe `PaireObject`:

```

1 public class TestPaireObject {
2 // Ah, vous voyez, j'ai enfin fait une classe à part...
3     public static void main(String[] args) {
4         PaireObject paire = new PaireObject("abc", "xyz");
5         String x = (String) paire.getPremier();
6         Double y = (Double) paire.getDeuxieme();
7     }
8 }

```

On constate que l'on doit faire un cast explicite pour récupérer les valeurs de `premier` et `deuxième`. En effet, `getPremier()` et `getDeuxieme()` renvoient des références vers `Object`.

Or les erreurs de cast d'objets ne sont pas relevées à la compilation, mais seulement à l'exécution. Ici, le programme compile, mais à l'exécution soulève une `java.lang.ClassCastException` car on ne peut pas convertir une `String` en `Double` (il existe en fait des méthodes de `String` pour ça, mais le cast n'en fait pas partie).

Cette solution permet donc d'écrire des méthodes incohérente avec des paires d'objets de types différents...

Note

Vous remarquerez aussi que dans cette méthode, le programmeur sait qu'il manipule une paire de `String`, mais comme la classe `Paire` contient des `Objects`, il doit trans-typer (caster) la valeur renvoyée par `getPremier` pour pouvoir l'utiliser correctement. Cela ajoute donc une lourdeur au programme, même s'il n'y a pas d'erreur de cast.

2.c Déclaration et utilisation d'une classe générique

Dans le cas d'une classe générique, on va préciser qu'il existe un type, sans préciser lequel. En général, on utilise la lettre T pour représenter ce type, il est entouré de <>: <T>.

On obtient, avec l'exemple précédent, la classe suivante:

```

1 public class Paire<T> {
2     private T premier;
3     private T deuxième;
4
5     public Paire(T premier, T deuxième) {
6         this.premier = premier;
7         this.deuxieme = deuxième;
8     }
9 }

```

```

10   public T getPremier() {
11     return premier;
12   }
13
14   public T getDeuxieme() {
15     return deuxieme;
16   }
17
18   // Note: mettre le test directement dans la classe ne respecte pas les
19   // Java coding conventions, mais permet de gagner de la place ici...
20   public static void main(String[] args) {
21     Paire<Double> paire = new Paire<Double>(Double.valueOf(3.0), Double.
22       valueOf(4.0));
23     System.out.println("Premier: " + paire.getPremier() + ", Deuxième: "
24       + paire.getDeuxieme());
25   }
26 }
```

A linstanciation de notre classe `Paire`, le type utilisé doit être explicite. Par exemple, à la ligne 20, on utilise `Paire<Double>` pour la déclaration et linstanciation.

Note



Notez que lors de limplémentation du constructeur lui-même dans la classe `Paire<T>`, on ne met pas les chevrons `Paire<T>(T premier, T deuxieme)` n'est pas correct.

Note



La classe générique `Paire<T>` est stockée dans un fichier nommé `Paire.java` et non `Paire<T>.java...`

Si l'on reprend l'exemple de test précédent, on obtient:

```

1  public class TestPaireGeneric {
2  // Ah, vous voyez, j'ai enfin fait une classe à part...
3  public static void main(String[] args) {
4    Paire<String> paire = new Paire<String>("abc", "xyz");
5    String x = paire.getPremier();
6    Double y = paire.getDeuxieme();
7  }
8 }
```

On remarque d'une part que le cast n'est plus obligatoire à la ligne 5 étant donné que `paire` est une instance de `Paire<String>` et que donc `paire.getPremier()` renvoie une référence vers `String`. Du coup, la ligne 6 donne une erreur à la compilation.

3 Généricité et héritage

On peut, dans une classe générique, préciser que l'on souhaite que le type utilisé hérite d'un type particulier. Par exemple, `Paire<T extends Number> paireDeNombre = new Paire<Double>(3.0, 4.0);`.

Note



Utilisé comme tel, cela n'a pas beaucoup d'intérêt, cela équivaut à `Paire<Number> paierDeNombres = new Paire<Number>(3.0, 4.0);`.

Par contre, cela devient intéressant lorsque l'on a affaire à des interface. On peut utiliser des méthodes de linterface, quelle que soit la hiérarchie des classes utilisées. L'exemple ci-dessous illustre ce principe. Ce principe est généralement très utilisé pour les tris.

```
1 public class PaireComparable<T extends Comparable<T>> {
2     private T premier;
3     private T deuxieme;
4
5     public PaireComparable(T premier, T deuxieme) {
6         this.premier = premier;
7         this.deuxieme = deuxieme;
8     }
9
10    public T getPremier() {
11        return premier;
12    }
13
14    public T getDeuxieme() {
15        return deuxieme;
16    }
17
18    public T minimum() {
19        T res;
20        // Un type qui implémente l'interface Comparable aura forcément
21        // une méthode +compareTo(Comparable): int
22        if (premier.compareTo(deuxieme) < 0) {
23            res = premier;
24        } else {
25            res = deuxieme;
26        }
27
28        return res;
29    }
30
31
32    public static void main(String[] args) {
33        // Ici, on simplifie le test en utilisant le fait que les classes
34        // de la classe Number implémentent l'interface Comparable
35        PaireComparable<Double> paire = new PaireComparable<Double>(Double.
36            valueOf(3.0), Double.valueOf(4.0));
37        System.out.println("Le minimum entre " + paire.getPremier() + " et "
38            + paire.getDeuxieme() + " est: " + paire.minimum());
39    }
40}
```

API Collection

1 Principe

Les *collections* Java sont un ensemble de classes définissant des **structures de données** efficaces pour stocker, rechercher et manipuler des objets. De nombreuses structures existent couvrant les besoins les plus courants:

- séquences d'éléments (listes)
- ensembles quelconques, ensembles ordonnés
- ensembles ordonnés (files, piles, files à priorité)
- des dictionnaires associatifs entre des clés et des valeurs
- et d'autres conteneurs plus spécifiques...

Le choix d'une collection dépend de l'utilisation recherchée et des coûts des opérations principalement réalisées.



Avant de se lancer dans le développement éventuel de vos propres structures de données, il est toujours préférable de d'abord chercher parmi les collections du langage celles qui répondent le mieux à vos besoins. Dans la majorité des cas, les structures adéquates existent déjà, qui plus est implantées de manière efficace et validées.

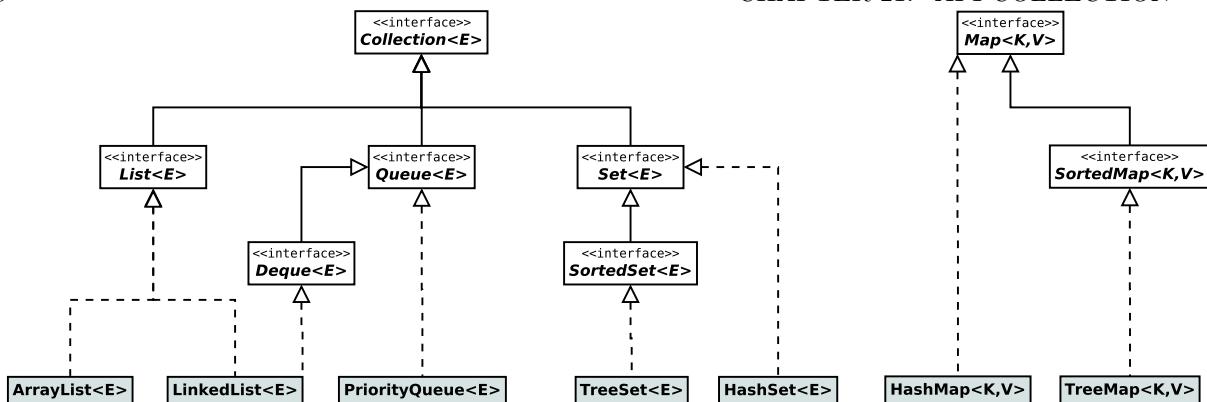
1.a Organisation

Ces différents *types de données abstraits* sont spécifiés dans des *interfaces* de haut niveau, puis implémentées dans différentes classes concrètes. Il existe en fait 2 hiérarchies de classes:

1. celle des collections proprement dites qui permettent de stocker des éléments. elles implémentent l'interface `Collection<E>`
2. celles des **dictionnaires associatifs** contenant des couples (*clé, valeur*) dont la racine est l'interface `Map<K, V>`.

Nous ne décrirons ici que les collections usuelles principales. De nombreuses ressources sont également disponibles, notamment l'API Doc¹ : toutes les classes et interface discutées ici sont dans le package `java.util`

¹<https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html>

**Note**

Les collections sont **génériques** : elles permettent de stocker des objets de tout type (`Object`, `String`, `Pangolin`, ...). Pour stocker des données d'un type de base (ce ne sont pas des objets), il faut utiliser des classes wrapper :`Integer` pour `int`, `Double` pour `double`, etc. La collection est déclarée avec le wrapper (par exemple `List<Integer>`), mais on peut y ajouter des données `int` sans besoin de conversion explicite (mécanisme d'auto-boxing).

1.b Parcours d'une collection: itérateur et `for each`

Toutes les collections possèdent au moins un point commun : le parcours de leur contenu. Le mécanisme d'itération permet de parcourir séquentiellement tous les éléments d'une collection, de manière uniforme et efficace. Il existe dans plusieurs langages, notamment Python ou C++.

En Java toute collection possède une méthode `Iterator<E> iterator()` qui retourne un itérateur permettant d'accéder aux éléments un par un en avançant dans la collection. Initialement, l'itérateur est placé avant le premier élément. A chaque accès, le prochain élément est retourné et l'itérateur avance à l'élément suivant. Lors du parcours complet d'une collection, chaque élément est retourné une et une seule fois, dans un ordre dépendant en fait du type de la collection.

Les deux méthodes principales d'un itérateur sont :

- `public boolean hasNext()` qui retourne true s'il reste des éléments dans l'itération ;
- `public E next()` qui **retourne** le prochain élément **et avance** dans l'itération. S'il n'y a plus d'élément, une `NoSuchElementException` est levée.

Le schéma d'utilisation est toujours le même, quelle que soit la collection :

```

1 Collection<E> coll = new ...;           // Collection existante, de type quelconque
2                                         // Elle contient des éléments de type E
3
4 Iterator<E> it = coll.iterator();       // Crée un nouvel itérateur sur la
   // collection,
5                                         // initialisé AVANT le 1er élément
6 while (it.hasNext()) {                  // Tant qu'il reste des éléments
7     E e = it.next();                   // Récupère le prochain élément et avance
8     ...
9 }
```



L'utilisation d'un itérateur nécessite d'écrire une boucle `while` et de ne pas oublier d'avancer dans l'itération avant de traiter l'élément (modèle de parcours en *avancer-traiter*). Parfait pour écrire une recherche ou un traitement particulier, le programme est plus compliqué que nécessaire pour un simple parcours, pour lequel on préférera l'utilisation d'un `for each`.

```

1 for (E e: coll) {
2 ...
3 }
```

Par rapport au parcours avec un itérateur, un `for each` traite systématiquement tous les éléments de la collection. L'itérateur permet aussi d'enlever un élément pendant le parcours (la classe `Iterator` possède une méthode `remove()`, optionnelle, qui retire de la collection le dernier élément retourné par `next()`). Pour cet usage, le `for each` est insuffisant. Il n'est d'ailleurs pas possible de modifier la collection parcourue pendant son parcours via un `for each`.

2 Les séquences: List

L'interface `List<E>` définit une séquence d'éléments, indexés du 1er au dernier par leur position dans la séquence : un entier de 0 à `size() - 1`. Les méthodes permettent d'insérer au début, à la fin, ou à une position précise. Lors d'une itération, les éléments sont naturellement parcourus dans l'ordre de la séquence. Les deux principales classes réalisant cette interface sont :

2.a LinkedList<E>

Fondée sur une liste doublement chaînée, cette classe est intéressante en cas d'ajouts et suppressions en début ou en fin de séquence. En ce qui concerne les ajouts et suppressions en milieu de séquence, elles se font également en temps constant, mais nécessite d'accéder à la position de la cellule à ajouter ou supprimer, ce qui prend un temps $\frac{n}{2}$ (donc un temps linéaire) si la cellule est en milieu de liste. Un tel ajout / suppression n'est donc efficace que s'il se fait en cours de parcours (par exemple en supprimant la cellule à laquelle on est en train d'accéder avec un itérateur).

```

1 // Exemple: une LinkedList de points
2 LinkedList<Point> points = new LinkedList<Point>();
3 points.add(new Point(0, 0));           // par défaut, ajout en fin
4 points.addFirst(new Point(1, 1));
5 points.add(new Point(2, 2));
6 System.out.println(points);          // [(1,1), (0,0), (2,2)]
7 System.out.println(points.get(2));    // Attention : la complexité de la mé
8     thode get(int i)                // de la classe LinkedList<> est en O(n)
9         !                           // Car il faut parcourir la liste
10    points.remove(1);              // enlève la valeur en position 1
11    System.out.println(points);    // [(1,1), (2,2)]
12    points.remove(0);              // Suppression en tête. Coût : O(1).
13    System.out.println(points);    // [(2,2)]
```

2.b ArrayList<E>

Fondée sur un tableau redimensionnable, les `ArrayList` ont un coût constant pour les accès direct à un élément en fonction de sa position (ième). Par contre, les coûts des insertions/suppressions en position quelconque sont linéaires (décalage des valeurs aux indices suivants).

```

1 // Exemple: une ArrayList de caractères
2 // => notez le ArrayList<Character> et pas ArrayList<char>
3 //     Pour les types de base, il faut utiliser des classes "wrapper"
4 //     (Integer, Float, Double ...) dans la déclaration
5 // => le mécanisme d'"auto-boxing" permet d'utiliser des char normalement:
6 //     - lettres.add('a') est wrappé par lettres.add(new Character('a'))
7 //     - lettres.remove(i) retourne lettres.remove(i).getValue()
8 ArrayList<Character> lettres = new ArrayList<Character>();
9 for (char c = 'a'; c <= 'z'; c++) {
```

```

10     lettres.add(c);                      // ajout en fin
11 }
12 lettres.set(4, 'E');                   // remplace la 5ème lettre par sa majuscule.
13                                         Coût : O(1)
13 System.out.println("La 25eme lettre est " + lettres.get(24));
14 System.out.println(lettres.get(5));      // accès par indice en O(1) dans une
14                                         ArrayList<Character>
15 lettres.remove(6);                     // coût! (decalages)
16
17 // On affiche les 10 premières lettres
18 Iterator<Character> it = lettres.iterator();
19 int n = 0;
20 while (it.hasNext() && n++ < 10) {
21     char c = it.next();
22     System.out.print(c + " ");
23 }                                         // affichage: a b c d E f h i j k

```

3 Files et piles: Queue, Deque

L’interface `Queue<E>` définit typiquement une file (FIFO, First In First Out), avec deux méthodes `add(E e)` qui ajoute en fin et `E remove()` qui enlève et retourne le premier élément.

L’interface `Deque<E>` (Double Ended Queue) spécialise une `Queue<E>` avec des méthodes d’insertion, d’accès et de retrait en tête et queue de liste : `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst`, `getLast`. Ces méthodes permettent notamment d’utiliser `Deque` comme une pile (LIFO, Last In First Out), en considérant par exemple la tête comme le sommet de la pile.

La classe de référence réalisant ces interfaces est `LinkedList<E>`. Les opérations principales sont à coût constant `O(1)`.

```

1 Queue<Pangolin> file = new LinkedList<Pangolin> ();
2 file.add(new Pangolin("Gérard", 1542));
3 file.add(new Pangolin("Pierre", 1939));
4 file.add(new Pangolin("Heckel", 1));
5 System.out.println(file.remove());           // c'est Gérard
6 file.add(new PangolinALongueQueue("Jeckel", 2));
7 while (!file.isEmpty()) {
8     System.out.println(file.remove());       // Pierre, Heckel puis Jeckel
9 }

```

Note



Notez qu’ici, la `file` a été **déclarée** avec un type abstrait de haut niveau, l’interface `Queue`. Par contre, elle a été **instanciée** avec le type `LinkedList`. En effet, le type `Queue` ne peut pas être instancié. Il est cependant recommandé d’utiliser un type abstrait pour la déclaration si l’on n’utilise que les méthodes déclarées à ce niveau. En effet, l’information est plus précise dans ce cas: on sait que conceptuellement l’objet `file` est utilisé comme une `Queue` (alors que la classe `LinkedList` autorise en fait beaucoup plus de choses). De plus il est possible de modifier le type d’instanciation ultérieurement (`ArrayList` ou `LinkedList`) pour des raisons de performances selon les opérations utilisées, sans modifier le reste du code.

4 Files à priorités

`PriorityQueue<E>` est une file à priorité : l'ordre de sortie des éléments ne dépend plus de leur date d'insertion, comme une FIFO ou LIFO, mais d'une priorité entre éléments.

L'implantation de la classe `PriorityQueue<E>` repose sur un tas binaire. Les coûts des opérations d'insertion et de retrait de l'élément le plus prioritaire sont en $O(\log(n))$. L'opération d'accès à l'élément le plus prioritaire sans le retirer `peek()` est l'opération la plus efficace : elle est en $O(1)$ (ce qui fait tout l'intérêt de cette structure par rapport à un `TreeSet`). Attention par contre, la recherche ou la suppression d'un élément quelconque sont possibles mais en $O(n)$.

En pratique les éléments doivent être munis d'une relation d'ordre, et l'élément le plus prioritaire (le prochain à sortir) est le plus petit selon cet ordre.

Deux solutions sont possibles pour définir cette relation d'ordre :

1. La classe `E` des éléments peut implémenter l'interface `Comparable<E>`, qui définit l'ordre dit *naturel* entre des instances de `E`. Elle doit donc redéfinir la méthode `public int compareTo(E e)` qui retourne une valeur négative/nulle/positive si `this` est plus petit/égal/ plus grand que '`e`'.

```

1 // Exemple: une PriorityQueue avec l'implémentation de l'interface
2 Comparable
3 public class Etudiant implements Comparable<Etudiant> {
4     private String name;
5     private int note;
6
7     public Etudiant(String name, int note) {
8         this.name = name;
9         this.note = note;
10    }
11    @Override
12    public String toString() {
13        return "Je m'appelle " + name + " et j'ai eu une note de "
14            + note;
15    }
16    // On implémente la méthode compareTo qui prend en paramètre
17    // un autre objet Etudiant à comparer avec this
18    @Override
19    public int compareTo(Etudiant e) {
20        // On retourne un nombre positif
21        // si la note de this est supérieur à celle de e
22        if(this.note > e.note) {
23            return 1;
24        }
25        if(this.note < e.note) {
26            return -1;
27        }
28        return 0;
29    }
30    // equals(Object o) doit être cohérent avec compareTo(Etudiant e),
31    // comme expliqué dans la Javadoc de l'interface Comparable<E>.
32    // Ici, donc, 2 étudiants doivent être égaux au sens de equals()
33    // si et seulement si ils se comparent à 0 au sens de compareTo()
34    @Override
35    public boolean equals(Object other) {
36        if(other instanceof Etudiant) {
37            Etudiant et = (Etudiant) other;
38            return et.note == this.note;
39        }
40        return false;
41    }
42    public class ExemplePriorityQueueComparable {
43        public static void main(String argc[]) {

```

```

44     // Création d'une PriorityQueue, comme on implémente l'
45     // interface Comparable
46     // il n'y a pas besoin de spécifier avec quelle méthode on
47     // veut comparer,
48     // Le tri var directement appeler la méthode compareTo de
49     // la classe Etudiant
50     Queue<Etudiant> etudiants = new PriorityQueue<Etudiant>();
51     etudiants.add(new Etudiant("Pauline", 14));
52     etudiants.add(new Etudiant("Julia", 10));
53     etudiants.add(new Etudiant("Théo", 16));
54     etudiants.add(new Etudiant("Alphonse", 12));
55     etudiants.add(new Etudiant("Pierre", 8));
56     etudiants.add(new Etudiant("Isabelle", 20));
57     etudiants.add(new Etudiant("Olivier", 15));
58
59     // Remarque : puisqu'une PriorityQueue repose sur un tas,
60     // un itérator sur une PriorityQueue
61     // ne parcourt pas dans l'ordre définit par le
62     // comparateur.
63
64     // Par contre, principe du tas :
65     // l'élément en tête est toujours le plus grand
66     // (méthodes peek() et poll())
67
68     // Utilisation de la méthode peek pour accéder à l'élément
69     // en tête de la file triée
70     while(etudiants.peek() != null) { // on peut aussi écrire
71         if( etudiants.size() > 0 )
72             // On retire l'élément en tête de la file triée
73             Etudiant e = etudiants.poll();
74             System.out.println(e);
75             // Affiche :
76             // Je m'appelle Pierre et j'ai eu une note de 8
77             // Je m'appelle Julia et j'ai eu une note de 10
78             // Je m'appelle Alphonse et j'ai eu une note de 12
79             // Je m'appelle Pauline et j'ai eu une note de 14
80             // Je m'appelle Théo et j'ai eu une note de 16
81             // etc.
82             // c'est à dire les étudiants, par ordre croissant de leur note.
83             // Complexité de toute la boucle boucle : O( n * log(n) )
84         }
85         // maintenant, la file de priorité est vide...
86     }
87 }
```

2. Il est aussi possible de déléguer la comparaison de deux objets E à une tierce classe qui réalise l'interface `Comparator<E>`. Celle-ci définit une seule méthode: `public int compare(E e1, E e2)` qui retourne une valeur négative/nulle/positive si `e1` est plus petit/égal/plus grand que `e2`. Si une classe fille de `Comparator` est donnée à la `PriorityQueue`, c'est elle qui effectue la comparaison des éléments même si le type de E implémente `Comparable<E>`.

Cette seconde approche est surtout utilisée pour pouvoir comparer des éléments selon des critères différents. Il est par exemple possible de créer deux classes qui comparent des étudiants selon leur nom ou selon leur note de POO.

```

1 // Exemple: une PriorityQueue avec l'implémentation de l'interface
2 // Comparator
3 // L'ordre choisi pour cet exemple est l'ordre alphabétique des noms.
4 // Il faut implémenter l'interface Comparator dans une classe tierce, nommée
5 // par exemple EtudiantComparator
6 class EtudiantComparator implements Comparator<Etudiant> {
    // On implémente la méthode compare(),
    // qui est la seule méthode définie dans l'interface Comparator.
```

```
7     // Elle prend en entrée deux objets Etudiant e1 et e2 à comparer
8     public int compare(Etudiant e1, Etudiant e2) {
9         // On retourne un nombre positif
10        // si le nom de e1 est plus grand alphabétiquement à celle
11        // de e2
12        if(e1.getName().compareToIgnoreCase(e2.getName()) > 0) {
13            return 1;
14        }
15        else if(e1.getName().compareToIgnoreCase(e2.getName()) < 0)
16        {
17            return -1;
18        }
19        else {
20            return 0;
21        }
22    }
23
24    class Etudiant {
25        private String name;
26        private int note;
27
28        public Etudiant(String name, int note) {
29            this.name = name;
30            this.note = note;
31        }
32        public String getName() {
33            return name;
34        }
35        @Override
36        public String toString() {
37            return "Je m'appelle " + name + " et j'ai eu une note de "
38            + note;
39        }
40        // equals(Object o) doit être cohérent avec compare(Etudiant e1,
41        // Etudiant e2)
42        // Ici, donc 2 étudiants doivent être égaux au sens de equals()
43        // si et seulement si ils se comparent à 0 au sens du comparateur
44        @Override
45        public boolean equals(Object other) {
46            if(other instanceof Etudiant) {
47                Etudiant et = (Etudiant) other;
48                return et.name.equals(this.name);
49            }
50            return false;
51        }
52
53    public class ExemplePriorityQueueComparator {
54        public static void main(String argc[]) {
55            // Création d'une PriorityQueue, comme on implémente l'
56            // interface Comparator
57            // dans une classe tierce, il faut spécifier avec quelle mé
58            // thode on veut comparer.
59            // Il faut donc donner une instance de la classe
60            // qui implémente la méthode de comparaison à la
61            // PriorityQueue
62            Comparator comparator = (Comparator)(new EtudiantComparator
63            ());
64            Queue<Etudiant> etudiants = new PriorityQueue<Etudiant>(
65                comparator);
66            etudiants.add(new Etudiant("Pauline", 14));
67            etudiants.add(new Etudiant("Julia", 10));
```

```

1      etudiants.add(new Etudiant("Théo", 16));
2      etudiants.add(new Etudiant("Alphonse", 12));
3      etudiants.add(new Etudiant("Pierre", 8));
4      etudiants.add(new Etudiant("Isabelle", 20));
5      etudiants.add(new Etudiant("Olivier", 15));
6
7      // le cout de l'ajout de tous les étudiants est en O(n * log
8      // (n))
9
10
11     // Utilisation de la méthode peek pour accéder à l'élément
12     // en tête de la file triée
13     while(etudiants.peek() != null) {
14         // On retire l'élément en tête de la file triée
15         Etudiant e = etudiants.poll();
16         System.out.println(e);
17         // Affiche :
18         // Je m'appelle Alphonse et j'ai eu une note de 12
19         // Je m'appelle Isabelle et j'ai eu une note de 20
20         // Je m'appelle Julia et j'ai eu une note de 10
21         // Je m'appelle Olivier et j'ai eu une note de 15
22         // Je m'appelle Pauline et j'ai eu une note de 14
23         // Je m'appelle Pierre et j'ai eu une note de 8
24         // Je m'appelle Théo et j'ai eu une note de 16
25     }
26     // maintenant, la file de priorité est vide...
27 }
28 }
```

5 Ensembles: Set

A la différence des séquences ou des queues, un ensemble défini par l'interface `Set<E>` n'admet pas de doublons: un élément ne peut pas être présent 2 fois dans la collection. Cette égalité entre les éléments est testée via la méthode `public boolean equals(Object o)` héritée de la super classe `Object`, qui doit de ce fait être correctement redéfinie dans la classe `E` des éléments du `Set`.

5.a Ensemble quelconque

La classe `HashSet<E>` implémente l'interface `Set<E>` avec une implémentation de type table de hachage. Le coût amorti des opérations principales (ajout, retrait, recherche) est en $O(1)$. L'itération retourne bien toutes les valeurs mais dans un ordre quelconque.

En plus de redéfinir `equals`, les éléments doivent aussi redéfinir une autre méthode de la classe `Object`: `public int hashCode()`, qui doit retourner une clé de hachage entière. Cette méthode doit être cohérente avec la redéfinition de l'égalité : si deux objets sont égaux au sens de `equals`, alors leur méthode `hashCode` doit retourner la même valeur (cf chapitre 17).

Un exemple de redéfinition et d'utilisation de cette méthode est donné ci-dessous.

```

1 class Fleur {
2     private String name;
3     public Fleur(String name) {
4         this.name = name;
5     }
6     // la méthode equals à redéfinir pour que l'ajout
7     // dans un HashSet se passe bien
8     @Override
9     public boolean equals(Object o) {
10        if(o instanceof Fleur) {
11            Fleur f = (Fleur)o;
12            if(name.equals(f.name)) {
13                return true;
14            }
15        }
16    }
17 }
```

```
15         }
16     return false;
17 }
18 @Override
19 public int hashCode() {
20     // Dans la vraie vie, il suffirait de dire que le code de hachage d'une
21     // Fleur
22     // est le code de hachage de son nom.
23     // Bien sur, la classe String redéfinit la méthode hashCode() !
24     // Donc on écrirait donc :
25     //         return name.hashCode();
26
27     // Dans notre exemple, on implante une méthode de hachage un peu bête,
28     // pour bien expliquer comment les fleurs vont être
29     // rangées dans la table de hachage.
30     // Bien sûr, cette méthode de hachage n'est pas recommandable,
31     // car elle génère beaucoup de collisions.
32     int hashCode = 0;
33     for(int i = 0; i < name.length(); i++) {
34         hashCode += (int)name.charAt(i);
35     }
36     return hashCode;
37 }
38 public String toString() {
39     return name + " a un hash code qui vaut : " + hashCode();
40 }
41
42 public class ExempleHashSet {
43     public static void main(String args[]) {
44
45         // On utilise un HashSet<Fleur>, il faut implémenter :
46         // - la redéfinition de la méthode boolean equals(Object o)
47         // - la redéfinition de la méthode int hashCode()
48
49         // Capacité initiale (optionnelle) : 16 cases dans le tableau de
50         // hachage
51         Set<Fleur> bouquetFleurs = new HashSet<Fleur>(16);
52
53         bouquetFleurs.add(new Fleur("tulipe"));
54         Fleur rose = new Fleur("rose");
55         bouquetFleurs.add(rose);
56         bouquetFleurs.add(new Fleur("coquelicot"));
57         bouquetFleurs.add(new Fleur("quecolicot"));
58         bouquetFleurs.add(new Fleur("lantana"));
59         bouquetFleurs.add(new Fleur("litupe"));
60         bouquetFleurs.add(new Fleur("lantana"));
61
62         // le cout de l'ajout de toutes les fleurs est en O(n) seulement
63         // car l'ajout d'un élément dans un HashSet est en O(1) coût amorti
64
65         System.out.println("Taille du HashSet : " + bouquetFleurs.size());
66         // Affiche :
67         // Taille du HashSet : 6
68         // Ah, la dernière fleur "lantana" n'a pas été ajoutée.
69         // Pourquoi ?
70
71         // Comme dans un HashSet, les éléments sont uniques (au sens de equals
72         // (())),
73         // l'ajout a échoué.
74
75         // le parcours est fait dans un ordre apparemment "quelconque"
76         for(Fleur fleur: bouquetFleurs) {
```

```

75     System.out.println(fleur);
76     // affiche :
77     // tulipe a un hash code qui vaut : 659
78     // litupe a un hash code qui vaut : 659
79     // coquelicot a un hash code qui vaut : 1080
80     // quecolicot a un hash code qui vaut : 1080
81     // rose a un hash code qui vaut : 441
82     // lantana a un hash code qui vaut : 735
83 }
84
85 // Utilisation de la méthode contains avec un objet rose en entrée
86 System.out.println("Le bouquet de fleurs contient t-il une rose? : "
87     + bouquetFleurs.contains(new Fleur("rose")));
88 // Affiche:
89 // Le bouquet de fleurs contient t-il une rose? : true
90 // Coût de la méthode contains() : O(1)
91
92 // On pourrait aussi utiliser un TreeSet<String> à la place du HashSet
93 // => il faudrait munir la classe Fleur d'un comparateur
94 // => le coût de l'ajout de toutes les fleurs passerait en O(n * log(n))
95 // => l'ordre d'affichage de mots respecterait l'ordre défini par le
96 // comparateur.
97 // => le coût de la méthode contains() passerait en O(log(n))
98 }
99 }
```

5.b Ensemble ordonné

La classe `TreeSet<E>` définit un ensemble dont les valeurs sont ordonnées. Il est donc nécessaire de munir les éléments d'une relation d'ordre pour pouvoir les comparer. Comme précédemment, deux solutions sont possibles :

1. La classe E des éléments peut réaliser l'interface `Comparable<E>`. C'est donc la méthode `compareTo` qui est utilisée pour ordonner les éléments dans la collection.
2. Un objet de type `Comparator<E>` peut être donné au `TreeSet`. C'est alors lui qui réalise la comparaison des éléments même si la classe E réalise `Comparable<E>`.

Quelle que soit la méthode utilisée, la comparaison doit être compatible avec l'égalité : si `e1.equals(e2)` (respectivement `!equals`) alors `e1.compareTo(e2)` et/ou `compare(e1, e2)` doivent retourner 0 (respectivement une valeur non nulle).

L'implantation de cette classe repose sur un arbre équilibré (de type rouge-noir), avec des coûts en $O(\log(n))$ pour les opérations principales.

Note



Pour un ensemble ordonné ou un file à priorité, c'est la relation d'ordre (avec `Comparable` ou un `Comparator`) qui définit si un élément est déjà présent dans la collection. Le test repose sur le résultat de `compareTo` ou `compare`, s'il est nul ou non, et pas sur `equals`.

Par contre il est recommandé de toujours redéfinir `equals` (et même `hashCode`) pour pouvoir utiliser correctement un objet dans d'autres types de collections.

6 Dictionnaires: Map

L'interface `Map<K, V>` spécifie des associations entre une clé de type K et une valeur de type de V. Un Map ne peut pas contenir des clés identiques, et chaque clé n'est associée qu'à une et une seule valeur.

Les opérations principales sont l'ajout d'un couple (`put(K key, V value)`), l'accès à une valeur via sa clé (`V get(K key)`), la recherche de clé ou de valeur, la suppression d'une valeur, etc. L'interface `SortedMap<K,V>` étend `Map<K,V>`, en rajoutant une relation d'ordre sur les clés du dictionnaire.

Deux classes principales existent:

1. `HashMap<K,V>` : une table avec hachage sur les clés ;
2. `TreeMap<K,V>` : un ensemble ordonné sur les clés (implanté là encore avec un arbre rouge-noir équilibré).

Comme pour les `Set`, les méthodes `equals`, `hashCode` ainsi qu'une relation d'ordre doivent être correctement définies, en particulier sur le type `K` des clés.

Une `Map` peut être parcouru de différentes manières. En fait trois méthodes renvoient les clés et/ou les valeurs dans des collections, qui peuvent à leur tour être itérées.

1. la méthode `Collection values()` retourne une collection (dont on ne connaît pas le type dynamique, mais ce n'est pas nécessaire) contenant toutes les valeurs.
2. `Set<K> keySet()` retourne un ensemble contenant toutes les clés.
3. `Set<Map.Entry<K,V>> entrySet()` retourne un ensemble de tous les couples (clé, valeur). Ces couples sont de type `Map.Entry<K,V>`, où `Entry<K,V>` est une classe interne à `Map` fournissant principalement deux méthodes `K getKey()` et `V getValue()`.

Pour un `SortedMap`, les collections ci-dessus sont ordonnées suivant l'ordre défini sur les clés.

```

1 // ex: dictionnaire associant des étudiants (clé, une chaîne)
2 // à des notes (valeurs entières)
3 Map<String, Integer> annuaire = new HashMap<String, Integer> ();
4
5 String mc = new String("Matthieu");
6 String sb = new String("Sylvain");
7 String nc = new String("Nicolas");
8 annuaire.put(mc, 4);
9 annuaire.put(sb, 18);
10 annuaire.put(nc, 12);
11 annuaire.put(mc, 14); // pas de doublons,
12 // mais remplace l'ancienne valeur associée à mc
13
14 // affichage avec toString() : {Nicolas=12, Sylvain=18, Matthieu=14}
15 // (ordre indéterminé, c'est du hachage)
16 System.out.println("L'annuaire contient: " + annuaire);
17
18 // et quelle est la note de Catherine?
19 Integer note = annuaire.get("Catherine");
20 System.out.println("La note de Catherine est: " + note); // null!
21
22 // ensemble des clés : [Nicolas, Sylvain, Matthieu]
23 Set<String> cles = annuaire.keySet();
24 System.out.println("Les clés sont: " + cles);
25
26 // collection des valeurs : [12, 18, 14]
27 Collection<Integer> notes = annuaire.values();
28 System.out.println("Les valeurs sont: " + notes);
29
30 // parcours avec un itérateur sur les couples
31 // (prévoir une aspirine pour la syntaxe...)
32 System.out.println("Les couples sont:");
33 Set<Map.Entry<String, Integer>> couples = annuaire.entrySet();
34 Iterator<Map.Entry<String, Integer>> itCouples = couples.iterator();
35 while (itCouples.hasNext()) {
36     Map.Entry<String, Integer> couple = itCouples.next();
37     System.out.println("\t" + couple.getKey()
38             + " a la note " + couple.getValue());
39 }
```


Méthodes et variables statiques

1 Exemple: la classe Math

Supposons que vous vouliez écrire une méthode `int abs(int x)` dans une classe `Math` qui calcule la valeur absolue d'un nombre `x`. Le comportement de cette méthode ne dépend pas de la valeur des attributs de la classe `Math`. Aussi, pourquoi devrait-on créer un objet de la classe `Math` pour utiliser cette méthode ?

La classe `Math` existe en Java, ainsi que la méthode `abs()`. Effectivement, il n'est pas possible d'instancier cette classe:

```

1 public class TestMath {
2     public static void main(String[] args) {
3         Math objetMath = new Math();
4     }
5 }
```

L'instruction précédente renvoie l'erreur suivante à la compilation:

```

1 > javac TestMath.java
2 TestMath.java:3: error: Math() has private access in Math
3     Math objetMath = new Math();
4                         ^
5 1 error
```

2 Méthodes statiques

Bien que Java soit un langage objet, il existe de rares cas où une instance de classe est inutile.

2.a Déclaration et appel de méthodes statiques

En Java

Le mot clé `static` permet alors à une méthode de s'exécuter sans avoir à instancier la classe qui la contient.

L'appel à une méthode `static` se fait alors en utilisant **le nom de la classe** plutôt que le nom de la référence à un objet.

Par exemple, en écrivant une classe `MaClassMath` qui contient une méthode statique:

```

1 public class MaClassMath {
2     public static int min (int a, int b) {
3         // retourne la plus petite valeur entre a et b
4         return ...;
5     }
6 }
```

Lorsque l'on souhaite utiliser la méthode `static min`, on l'appelle sur le nom de la classe `MaClassMath`:

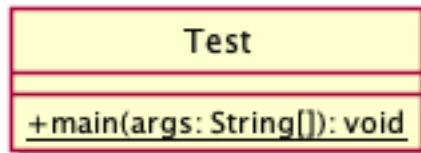
```

1 public class TestMaClassMath {
2     public static void main(String[] args) {
3         int x = MaClassMath.min(21, 4);
4     }
5 }
```

En Java, seule les attributs et les méthodes peuvent être *static*. Les classes, elles, ne sont pas *static*¹. De manière générale, une classe possédant des méthodes statiques n'est pas conçue pour être instanciée. Mais cela ne signifie pas qu'une classe possédant une méthode `static` ne doit jamais être instanciée : si d'autres méthodes de la classe ne sont pas statiques, alors l'instanciation doit être possible.

En UML

En UML, une méthode statique est déclarée soulignée:



2.b Méthodes statiques et attributs

Considérons le code suivant:

```

1 public class Chien {
2     private int taille;
3
4     public Chien(int taille) {
5         this.taille = taille;
6     }
7
8     public static void aboyer() {
9         if (taille < 20) {
10             System.out.println("kai kai");
11         } else {
12             System.out.println("Ouf Ouf");
13         }
14     }
15 }
```

La compilation de ce code produit l'erreur suivante:

```

1 > javac Chien.java
2 Chien.java:5: error: non-static variable taille cannot be referenced from a
   static context
3     if (taille < 20) {
4         ^
5 1 error
```

En effet, Comme la méthode `aboyer()` est `static`, elle peut être appelée sans qu'aucune instance n'ait été créée. Or la valeur de l'attribut `taille` ne prend une valeur que lors d'une instanciation.

¹à l'exception des classes internes *inner class* que nous ne voyons pas dans ce cours et qu'il faut en général éviter)

Note

Une méthode **static** ne peut pas utiliser d'attribut de la classe dans laquelle elle est déclarée.

Note

Pour les même raisons, une méthode **static** ne peut pas utiliser une méthode non-statique (à moins de créer une instance de la classe en question).

2.c Une méthode statique célèbre

Le programme principal Java est appelé par la JVM sans instanciation initiale:

```
1 public static void main(String[] args)
```

Le mot clé **public** indique que la méthode **main** peut être exécutée par la JVM. **static** indique qu'il n'y a pas besoin d'instanciation pour exécuter la méthode **main**. **void** signifie que la méthode de renvoie rien. **main** est le nom de la méthode **args** est le nom du paramètre de la méthode (qui peut être indiqué lorsque l'on lance le programme en ligne commande `java NomDeLaClassePrincipale [paramètres optionnels]`) **String[]** est le type du paramètre, c'est-à-dire tableau de chaînes de caractères.

3 Variables statiques

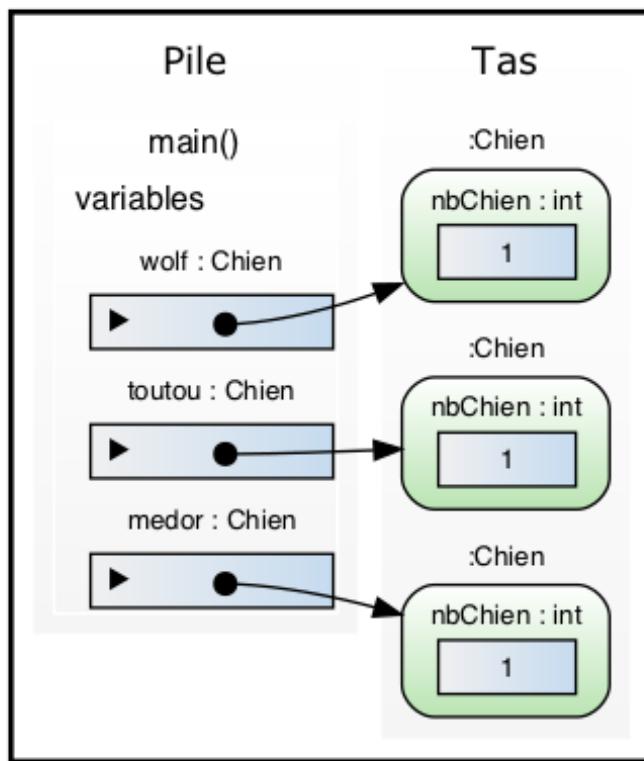
Il existe également des attributs statiques que l'on appelle **variable de classe** par opposition aux variables d'instances (attributs). Une variable de classe est partagée par toutes les instances de la classe.

Considérons dans un premier temps le code suivant:

```
1 public class Chien {  
2     private int nbChien;  
3  
4     public Chien() {  
5         nbChien++;  
6     }  
7 }
```

```
1 public class Test {  
2     public static void main(String[] args) {  
3         Chien wolf    = new Chien();  
4         Chien toutou = new Chien();  
5         Chien medor  = new Chien();  
6     }  
7 }
```

On obtient le diagramme APO suivant:



Si l'on souhaite compter le nombre d'instances de `Chien` on peut transformer l'attribut `nbChien` en variable de classe en la déclarant `static`.

On obtient le code suivant:

```

1 public class Chien
2     private static int nbChien;
3
4     public Chien() {
5         nbChien++;
6     }

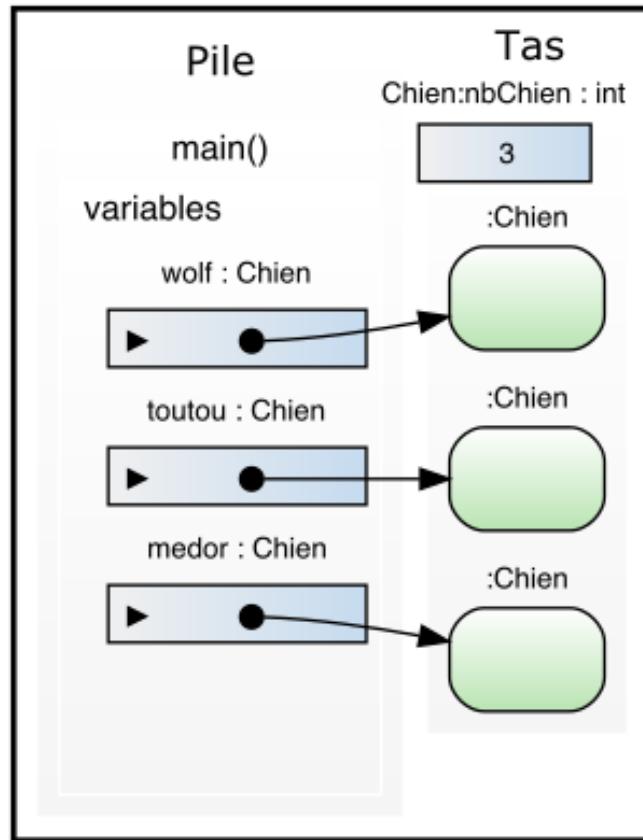
```

```

1 public class Test {
2     public static void main(String[] args) {
3         Chien wolf    = new Chien();
4         Chien toutou = new Chien();
5         Chien medor  = new Chien();
6     }
7 }

```

On obtient le diagramme APO suivant:



Les variables de classe sont initialisées lorsque la classe est chargée. C'est la JVM qui choisit le moment où la classe va être chargée. En général, le chargement intervient juste avant la création d'un objet de cette classe ou l'invocation d'une méthode statique.

Les règles utilisées pour l'initialisation des variables de classe sont les mêmes que pour les variables d'instances.

Part D

Ordre supérieur en Java

Fonctions: l'ordre supérieur en java

Fiche

<u>Prérequis</u>	<ul style="list-style-type: none"> Chapitre 14 Java: La classe ! Chapitre 16 Héritage et Polymorphisme Chapitre 19 Interfaces Chapitre 20 Généricité
<u>Objectif pédagogique</u>	Connaître l'API fonctionnelle en Java 8+
<u>Intérêt</u>	<ul style="list-style-type: none"> Écrire des lambda expressions Utiliser des fonctions d'ordre supérieur Comprendre l'API fonctionnelle de java 8+
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> Être capable de créer des lambda expressions Être capable d'utiliser des fonctions d'ordre supérieur existantes Être capable de créer une fonction d'ordre supérieur

1 Principe

Jusqu'à maintenant, nous n'avons utilisé que des fonctions du premier ordre. Une fonction du premier ordre est une fonction (au sens général du terme) qui prend en paramètres et en type de retour des valeurs de types primitifs ou objets (type références) usuels.

Ces fonctions sont les méthodes statiques ou dynamiques de l'ensemble des classes vues jusqu'alors.

Il est néanmoins nécessaire dans la plupart des langages de programmation de pouvoir créer et utiliser des fonctions d'ordre supérieur, c'est-à-dire des fonctions prenant en paramètres ou en type de retour d'autres *fonctions*.



Ici, nous utilisons le terme *fonction* dans le sens de la programmation fonctionnelle. En java, cela correspond à n'importe quelle méthode, quel que soit son type de retour (même `void`).

2 Intérêt

Pour mieux comprendre l'intérêt de l'ordre supérieur, nous évoquerons 2 cas d'usages très répandus.

2.a Un exemple : les Threads

Le premier exemple est l'utilisation de threads (des processus qui s'exécutent en parallèle).

Pour tirer partie des ordinateurs actuels (qui embarquent nombre de CPU ou cœurs), une pratique usuelle est de construire un ensemble de processus qui vont exécuter différentes tâches en parallèle. En java, un tel ensemble est appelé un *thread pool*.

Ainsi, si l'on a un ensemble de 10000 tâches à exécuter sur un machine embarquant suffisamment de cœurs, on peut créer un ensemble de 8 threads et lui demander d'exécuter les 10000 tâches. Chaque fois qu'une tâche sera terminée, le thread libéré prendra en charge la tâche suivante. Ainsi, les 10000 tâches seront exécutées en parallèle avec un maximum de 8 tâches exécutées simultanément.

En java, cet ensemble de thread est un objet implémentant l'interface `Executor`, qui définit la méthode `execute(...)`. Le paramètre à passer à cette méthode sera une tâche.

Il y a donc nécessité à pouvoir manipuler un objet représentant une **tâche**, qui sera passée en paramètre à l'`Executor`.

2.b Un exemple : les Listeners

Le second exemple montrant l'utilité de l'ordre supérieur est visible dès que l'on construit une interface graphique.

Supposons que l'on construise une simple interface graphique contenant un *bouton*. Un bouton va devoir exécuter une tâche *lorsque l'utilisateur cliquera dessus*, il est donc nécessaire d'indiquer à l'interface graphique quelle tâche sera exécutée lors de ce clic.

Note



Il est important à ce stade de comprendre que c'est *au moment de la construction de l'interface* que l'on doit transmettre la tâche qui sera exécutée *au moment où l'utilisateur cliquera sur le bouton*.

L'objet bouton implémente la méthode `addActionListener(...)` qui prend en paramètre la tâche à exécuter. Ici aussi, il y a nécessité à pouvoir manipuler un objet représentant une **tâche**, qui sera passée en paramètre au bouton.

3 Ordre supérieur en java avant java 8

Dans les 2 cas d'usages décrits ci dessus, il y a nécessité de pouvoir manipuler un objet qui représente une tâche. Avant java 8, l'astuce consiste à définir une tâche comme une instance d'une classe qui ne définit qu'une et une seule méthode particulière dont le nom est connu de ceux qui l'utilisent.

Pour les deux cas d'usages ci dessus, java défini les interfaces :

- `Runnable` qui définit l'unique méthode `public void run()`.
- `ActionListener` qui définit l'unique méthode `public void actionPerformed(ActionEvent event)`. qui prendra en paramètre l'action de l'utilisateur.

Ainsi, une tâche est vue comme une classe qui implémente l'une de ces interfaces et pour laquelle on construit une instance qui sera passée en paramètre.

On peut donc passer un objet représentant la tâche sans l'exécuter. L'objet est passé, et, au moment voulu, le thread ou le bouton appelleront la méthode implementée par cet objet (ils exécutent la tâche voulue).

L'usage de ce genre de tâche est tel que, pour éviter d'avoir à créer une classe particulière pour chaque tâche possible dans un programme, java avait introduit la notion de *classe anonyme*. Un exemple d'une telle classe anonyme est donnée ci dessous :

```

1 ...
2 executor.execute(new Runnable() {
3     @Override
4     public void run() {
5         ... // le code de la tâche à exécuter
6     }
7 });
8 ...

```

Dans cet exemple, nous appelons la méthode `execute(...)` de l'objet `executor` en lui passant en paramètre une instance d'une classe implémentant `Runnable` qui n'a pas de nom et qui est définie *en ligne*. La méthode `run()` définie dans l'interface `Runnable` est implémentée et c'est cette méthode qui sera appelée le moment voulu.

Cette technique permet ainsi de *réifier* une tâche (manipuler une tâche comme un objet).

4 Ordre supérieur en java depuis java 8

La solution utilisée avant java 8 est intéressante dans son principe, mais donne lieu à du code très verbeux. Dans l'exemple précédent, seul le *contenu* de la méthode `run` est utile, pas le reste, qui nuit en général à la lisibilité du programme.

De plus, d'autre cas d'usages ont été définis qui sont amenés à se généraliser (notamment l'API des streams de java 8).

Java 8 a donc décidé d'ajouter l'ordre supérieur à sa panoplie et a défini la notion d'*interface fonctionnelle*, qui permet de manipuler des *fonctions*¹ en tant qu'objet standard.

4.a Les lambda expressions

Pour obtenir un ordre supérieur, il faut pouvoir manipuler une variable dont la *valeur est une fonction*.

Java a donc défini un moyen de dénoter facilement une fonction : une lambda expression.

Exemple

Une lambda expression a 2 parties séparées par l'opérateur `->`. A gauche de l'opérateur se trouve la liste des paramètres de la fonction et à droite de l'opérateur se trouve le corps de la fonction. Par exemple :

```

1 (x, y) -> Math.pow(x, y)

```

Cette construction définit une fonction à 2 arguments x et y et qui retourne x^y .



Il faut ici bien distinguer la différence qu'il y a entre une variable qui contient *une fonction* et une variable qui contient le *résultat de l'exécution d'une fonction*. Dans le code ci dessus, la méthode `Math.pow` n'est pas exécutée.

L'objet fonction ainsi construit peut être affecté à une variable :

```

1 f = (x, y) -> Math.pow(x, y)

```

Bien évidemment, il faudra donner à la variable le bon type. Dans la ligne ci dessus, la variable `f` est une instance d'une *interface fonctionnelle* que nous définirons dans le paragraphe 4.b.

La syntaxe des lambda expressions

La syntaxe générale des lambda expression est la suivante :

`<<paramètres>> -> <<corps>>`

¹Ici aussi, la notion de fonction est prise dans son sens large, tel que défini habituellement dans un langage fonctionnel. Il peut s'agir d'une fonction avec 0, 1 ou plusieurs paramètres et avec ou sans valeur de retour.

Les paramètres Une lambda expression peut avoir 0 ou plusieurs paramètres. Dans le cas général, ces paramètres sont spécifiés entre parenthèses. Il est possible, mais pas obligatoire de spécifier le type des paramètres.

Exemple	Remarques
<code>() -> {return Math.random();}</code>	Une fonction sans paramètres et fournissant une valeur de type double (aléatoire).
<code>(x) -> {return x + 1;}</code>	Une fonction avec un paramètre de type non précisé et fournissant un résultat.
<code>(x) -> {System.out.println(x);}</code>	Une fonction avec un paramètre de type non précisé et ne fournissant aucun résultat.
<code>(String x) -> {return x + 1;}</code>	Une fonction avec un paramètre de type String et fournissant un résultat (dont on peut inférer qu'il est de type String).
<code>(x, y) -> {return x + y;}</code>	Une fonction avec deux paramètres de type non précis et fournissant un résultat. les paramètres sont séparés par une virgule.
<code>(String x, String y) -> {return x + y;}</code>	Une fonction avec deux paramètres de type String et fournissant un résultat (dont on peut inférer qu'il est de type String).

Note



Le type des paramètres ne sera pas obligatoirement spécifié, néanmoins le compilateur java devra déterminer le type des paramètres en fonction du contexte dans lequel la lambda expression est définie. Ce type devra donc être spécifié lorsque cela est nécessaire (pour vous ou pour le compilateur). De plus, soit tous les types des paramètres sont précisés, soit aucun.

Dans le cas particulier où il n'y a qu'un paramètre et que son type n'est pas précisé, les parenthèses peuvent être omises.

Exemple	Remarques
<code>x -> {return x + 1;}</code>	Une fonction avec un paramètre de type non précisé et fournissant un résultat.
<code>x -> {System.out.println(x);}</code>	Une fonction avec un paramètre de type non précisé et ne fournissant aucun résultat.

Le corps d'une lambda expression Dans le cas général, le corps d'une lambda expression est fourni entre accolades et est conforme en tous point à un bloc Java. Si ce corps contient un `return` avec une valeur, on peut inférer que la fonction retourne une valeur et son type.

Il est possible d'avoir plusieurs instructions (voire une boucle ou n'importe quelle autre structure de contrôle Java) dans le corps de la fonction.

Exemple	Remarques
<code>x -> {return x + 1;}</code>	Une fonction avec un paramètre de type non précisé et fournissant un résultat.
<code>x -> { int r = x + 1; return r; }</code>	Une fonction avec un paramètre de type non précisé et fournissant un résultat entier.
<code>(double x, int y) -> { double r = x; for(int i = 0; i < y; i++) { r = r * x; } return r; }</code>	Une fonction avec deux paramètres et fournissant un résultat double .

Cas d'une lambda expression simple Dans le cas le plus fréquent, le corps de la méthode est réduit à une unique expression. Dans ce cas, on peut omettre les accolades et le mot clé **return** pour ne préciser que l'expression (dont le résultat sera le résultat de la fonction).

Exemple	Remarques
<code>() -> Math.random()</code>	Une fonction sans paramètres et fournissant une valeur de type double (aléatoire). ATTENTION : il y a une différence fondamentale entre <code>() -> Math.random()</code> qui représente un objet fonction qui pourra être appelé ultérieurement et <code>Math.random()</code> qui représente une valeur double, résultat d'un appel à la méthode <code>random()</code> de l'objet <code>Math</code> .
<code>x -> x + 1</code>	Une fonction avec un paramètre de type non précisé et fournissant un résultat.
<code>x -> System.out.println(x)</code>	De même pour une fonction ne fournissant aucun résultat.
<code>(String x) -> x + 1</code>	Une fonction avec un paramètre de type String et fournissant un résultat (dont on peut inférer qu'il est de type String).
<code>(x, y) -> x + y</code>	Une fonction avec deux paramètres de type non précisé et fournissant un résultat. les paramètres sont séparés par une virgule.
<code>(String x, String y) -> x + y</code>	Une fonction avec deux paramètres de type String et fournissant un résultat (dont on peut inférer qu'il est de type String).

Cas atomique d'une référence à une méthode Parfois, on veut manipuler une méthode d'un objet ou une méthode statique de classe en tant que valeur d'une variable de type fonctionnelle. Dans ce cas, on utilisera la syntaxe d'une référence de méthode, qui utilise l'opérateur `:::`.

Il existe 4 contextes différents dans lesquels on peut utiliser une référence de méthode.

Contexte	Syntaxe	Exemple
Une méthode statique <code>m</code> de la classe <code>C</code>	<code>C::m</code>	<code>Math::random</code>
Une méthode <code>m</code> de l'instance <code>o</code>	<code>o::m</code>	<code>s::indexOf</code>
Un méthode <code>m</code> d'une instance quelconque de la classe <code>C</code>	<code>C::m</code>	<code>String::indexOf</code>
Un constructeur de la classe <code>C</code>	<code>C::new</code>	<code>String::new</code>

Nous donnons quelques exemples commentés, avec leur équivalent sous forme de lambda expression.

Exemple	lambda équivalente	Remarques
Math::random	() -> Math.random()	La méthode random() de l'objet Math est une fonction statique sans arguments.
s::indexOf	x -> s.indexOf(x)	En supposant que s est une instance de String, nous faisons référence ici à la méthode indexOf(String x) appliquée à l'objet s.
String::indexOf	(s, x) -> s.indexOf(x)	Vous remarquerez que dans ce cas, on fait référence à une méthode non statique sans préciser l'instance sur laquelle elle doit s'appliquer. Dans ce cas, l'instance est le premier paramètre de la lambda expression et les paramètres suivant représentent les paramètres de la méthode.
String::new	(() -> new String() (x) -> new String(x) ...)	Dans le cas d'un constructeur ou d'une méthode surchargés ^a le compilateur devra déterminer de quelle méthode il s'agit grâce au contexte de type dans lequel la lambda expression est définie. Les constructeurs sont des méthodes qui sont souvent surchargés. ^a Rappel: une méthode est dite surchargée lorsqu'il existe plusieurs signatures différentes (différents paramètres) pour un même nom de méthode. Par exemple, dans la classe String, la méthode substring est surchargée (substring(int from) et substring(int from, int to)).

4.b Notion d'interface fonctionnelle

Le type d'une variable contenant une fonction est appelé une *interface fonctionnelle*. Il est possible de construire sa propre interface fonctionnelle.

Définition



En java, une interface fonctionnelle est une interface qui ne définit qu'une et une seule méthode.

Cette définition peut sembler étonnante, mais une interface fonctionnelle n'a rien de particulier qui la distingue d'une interface non fonctionnelle, si ce n'est le nombre de méthodes définies.

Prenons quelques exemples :

```

1 public interface MonInterface {
2     public double faitCa(double a, double b);
3 }
4 public interface MonAutreInterface {
5     public int faitCi();
6     public String faitCa(String x);
7 }
```

Dans cet exemple, MonInterface est une interface fonctionnelle, tandis que MonAutreInterface n'en est pas une. Une lambda expression peut être affectée à une *variable de type interface fonctionnelle* si sa signature est compatible (même type de retour, même nombre d'arguments de même types respectifs). Par exemple :

```
1 MonInterface f = (x, y) -> Math.pow(x, y);
```

Dans cette déclaration de variable, on définit une lambda expression (x, y) -> Math.pow(x, y). Cette lambda expression prend deux paramètres de types non précisés et renvoie le résultat de la méthode Math.pow(x, y). Le compilateur Java sait que pow est une méthode statique de la classe Math qui prend en paramètres 2 doubles et renvoie 1 double. Il est donc capable d'inférer que les paramètres seront de type castable en double et que le résultat sera un double.

Cette lambda expression peut être affectée à une variable du type MonInterface. En effet, il s'agit d'une interface fonctionnelle dont l'unique méthode est compatible avec la lambda expression (deux arguments de type double et un résultat de type double).

Cette déclaration est donc parfaitement valide.

Définir une interface fonctionnelle

Comme nous venons de le voir, rien ne distingue une interface fonctionnelle d'une interface "normale" si ce n'est le nombre de méthode qu'elle définit. Néanmoins Java 8 fourni une annotation `@FunctionalInterface` destinée à la définition d'interfaces fonctionnelles. Cette annotation force le compilateur à vérifier que l'interface annotée est bien une interface fonctionnelle.

Ainsi, dans le listing ci dessous, il y aura une erreur de compilation car l'interface `MonAutreInterface` n'est pas fonctionnelle.

```

1  @FunctionalInterface
2  public interface MonInterface {
3      public double faitCa(double a, double b);
4  }
5  @FunctionalInterface
6  public interface MonAutreInterface {
7      public int faitCi();
8      public String faitCa(String x);
9 }
```

Par contre, il n'y aura aucune différence entre une interface définie avec le mot clé et une interface fonctionnelle définie sans le mot clé. Il s'agit juste d'un outil qui rend explicite aux développeurs l'intention de l'interface définie et qui ne permettra pas une modification de l'interface la rendant non fonctionnelle.

Évaluer une fonction

Lorsque l'on souhaite évaluer (*exécuter*) une fonction `f`, il suffira d'appeler la méthode définie par l'interface.

Dans notre cas, il s'agit de la méthode `faitCa`.

```

1 MonInterface f = (x, y) -> Math.pow(x, y);
2
3 double res = f.faitCa(2, 3); // res contient 8
```

4.c Interfaces fonctionnelles prédéfinies en Java 8+

Il est possible de définir ses propres interfaces fonctionnelles en Java 8+, mais cela n'est pas nécessaire dans une grande partie des cas. En effet, Java a défini des interfaces fonctionnelles courantes qui sont utilisées dans l'API des Streams et que l'on peut utiliser dans nos propres projets.

Le tableau suivant présente un sous ensemble des interfaces fonctionnelles définies dans Java 8+.

Interface	Exemple	Description
Function<T,R>	x -> x.toString()	Une fonction prenant un paramètre de type T et renvoyant un résultat de type R.
UnaryOperator<T>	x -> x + x	Une opération sur un paramètre de type T et renvoyant un résultat de même type T. Cette interface hérite de Function<T,T>.
Predicate<T>	x -> "".equals(x)	Une prédictat prenant un paramètre de type T et renvoyant un booléen.
Consumer<T>	x -> System.out.println(x)	Une fonction prenant un paramètre de type T et ne renvoyant aucun résultat.

Toutes ces interfaces existent aussi en version avec 2 paramètres.

BiFunction<T,U,R>	(x,y) -> x.toString() + y.toString()	Une fonction prenant deux paramètres de type respectifs T et U et renvoyant un résultat de type R.
BinaryOperator<T>	(x,y) -> x + y	Une opération sur deux paramètres de type T et renvoyant un résultat de même type T. Cette interface hérite de BiFunction<T,T,T>.
BiPredicate<T,U>	(x,y) -> x.equals(y)	Une prédictat prenant deux paramètres de type respectifs T et U et renvoyant un booléen.
BiConsumer<T,U>	(x,y) -> System.out.println(x + y)	Une fonction prenant deux paramètres de type respectifs T et U et ne renvoyant aucun résultat.

Il existe aussi des fonctions sans paramètres.

Supplier<T>	() -> new T()	Une fonction sans paramètre renvoyant un résultat de type T.
-------------	---------------	--

Java définit aussi des interfaces équivalentes qui prennent ou retournent des paramètres de types primitifs. En effet, la généréricité ne peut être utilisée avec les types primitifs. Par exemple : LongToIntFunction pour une fonction prenant un long et retournant un int ou ToIntFunction<T> pour une fonction prenant un objet de type T et renvoyant un int.

4.d Un exemple commenté

Prenons un exemple plus complet.

```

1 public interface Operateur {
2     public double calcule(double a);
3 }
4
5 public class Util {
6     public static double[] appliqueATout(double[] t, Operateur f) {
7         double[] res = new double[t.length];
8         for (int i = 0; i < t.length; i++) {
9             res[i] = f.calcule(t[i]);
10        }
11        return res;
12    }
13 }
14
15 public class Main {
16     public static void main(String args[]) {
17         Operateur f = x -> x*2;
18         Operateur g = x -> x*x;
19
20         double[] data = new double[10];
21         for (int i = 0; i < data.length; i++) {
22             data[i] = i;
23         }
24         data = Util.appliqueATout(data, f);
25     }
26 }
```

```

25     data = Util.appliqueATout(data, g);
26     for (int i = 0; i < data.length; i++) {
27         System.out.print(data[i] + " ");
28     }
29 }

```

La classe `Util` définit une méthode statique qui prend en paramètre un `double[] t` et un `Operateur f` et renvoie un nouveau `double[] r` tel que $\forall i r_i = f(t_i)$.

Note



La méthode `appliqueATout` est une méthode dite *d'ordre supérieur*. En effet, elle prend en paramètre **une fonction** ce qui la distingue des méthodes vues auparavant.

Le programme principal crée 2 opérateurs $f : x \rightarrow 2x$ et $g : x \rightarrow x^2$. Il crée ensuite un tableau tel que $\forall i r_i = i$ puis lui applique l'opérateur f . On obtient donc un tableau tel que, $\forall i r_i = f(i) = 2i$. On applique ensuite l'opérateur g , pour obtenir une tableau tel que $\forall i r_i = g(2i) = (2i)^2$.

Le résultat obtenu est affiché : 0.0 4.0 16.0 36.0 64.0 100.0 144.0 196.0 256.0 324.0

Essayez de dessiner le diagramme APO de cette exécution pour bien comprendre ce qui se passe.

Ce qui est intéressant dans ce programme, c'est qu'il suffit de changer la définition d'un opérateur (changer une lambda expression) pour changer l'exécution de ce qui est calculé dans la boucle de la classe `Util`.

Note



C'est un des nombreux aspects intéressants avec l'ordre supérieur. Lorsque l'on sait définir et manipuler une fonction, il devient possible de passer une fonction à une librairie qui l'appliquera à tous les éléments d'une collection, sans avoir à écrire le moindre parcours (et donc sans bug...). De plus, si l'application de la fonction peut se faire indépendamment pour chaque élément, la librairie pourra éventuellement l'appliquer en exploitant le parallélisme de la machine...

On remarque que ce programme construit un nouveau tableau pour chaque application d'un opérateur. Ceci peut aussi être amélioré grâce à l'ordre supérieur. Reprenons l'exemple en rajoutant une méthode à la classe `Util`.

```

1 public class Util {
2     public static double[] appliqueATout(double[] t, Operateur f) {
3         // inchangé
4     }
5
6     public static Operateur fPuisG(Operateur f, Operateur g) {
7         return x -> g.calcule(f.calcule(x));
8     }
9 }

```

Cette nouvelle méthode permet de composer les 2 opérateurs f et g pour définir un nouvel opérateur équivalent à $g \circ f$ (rappel: $(g \circ f)(x) = g(f(x))$).

Note



Si vous regardez attentivement cette méthode, vous remarquerez que ni `f` ni `g` ne sont évalués. Pour vous en convaincre, demandez-vous quelle est la valeur de `x` dans cette méthode. Cette méthode construit simplement une nouvelle fonction à partir des deux précédentes.

On reprend ensuite le programme principal qui calcule le même résultat, sans avoir recours à un tableau intermédiaire.

```

1 public class Main {
2
3     public static void main(String args[]) {
4         Operateur f = x -> x*2;

```

```
5     Operateur g = x -> x*x;
6
7     Operateur gDefDeX = Util.fPuisG(f, g);
8
9     double[] data = new double[10];
10    for (int i = 0; i < data.length; i++) {
11        data[i] = i;
12    }
13
14    // On applique maintenant la composition de f et g
15    data = Util.appliqueATout(data, gDefDeX);
16
17    for (int i = 0; i < data.length; i++) {
18        System.out.print(data[i] + " ");
19    }
20}
21}
```

Ce qu'il faut retenir ici, c'est qu'un fonction *est un objet*. Cette notion n'est pas entièrement nouvelle puisque, par le biais d'une interface, il était possible de voir un objet comme un comportement (comme par exemple l'interface Comparable<T> que nous utilisions dans un tri générique). Depuis java 8, néanmoins, cette possibilité de manipuler un objet fonction est devenu un usage courant.

API des Streams: l'ordre supérieur et les collections Java

Fiche

<u>Prérequis</u>	<ul style="list-style-type: none"> Chapitre 14 Java: La classe ! Chapitre 16 Héritage et Polymorphisme Chapitre 20 Généricité Chapitre 21 API Collections Chapitre 23 Fonctions et ordre supérieur
<u>Objectif pédagogique</u>	Connaître l'API des Streams de Java 8+
<u>Intérêt</u>	<ul style="list-style-type: none"> Utiliser des fonctions d'ordre supérieur Comprendre l'API des Streams de java 8+
<u>Compétences à acquérir</u>	<ul style="list-style-type: none"> Écrire des traitements complexes comme des opérations de Streams

1 Introduction

Dans le chapitre précédent, nous avons vu qu'il est possible de manipuler des fonctions comme des objets. Cette possibilité permet d'envisager des traitements simples où l'on pourrait appliquer une même fonction à tous les éléments d'une collection.

C'est d'ailleurs un cas d'usage que nous avons esquissé dans des exemples.

Cette idée est l'idée sous-jacente de l'API des Streams de Java 8. Permettre de résoudre un problème algorithmique en le décomposant en un certain nombre de traitements simples que l'on applique aux éléments d'une collection ou d'une séquence.



Attention, le terme de *Stream* est ambigu. Il est à l'origine plutôt utilisé pour désigner les flux de données de bas niveau (ex: `FileInputStream`). Ici, nous faisons référence à l'API introduite par Java 8 qui permet d'abstraire une collection en un flux de données sur lesquelles sont appliqués des traitements successifs.

2 Un exemple préliminaire commenté : l'abréviation majuscule

Prenons un exemple assez simple qui est la construction d'une abréviation majuscule à partir d'un chaîne de caractère. Une abréviation majuscule s'obtient en supprimant tous les caractères non majuscules de la chaîne de

180 CHAPTER 24. API DES STREAMS: L'ORDRE SUPÉRIEUR ET LES COLLECTIONS JAVA
caractère. De plus, les majuscules accentuées sont transformées en leur équivalent sans diacritique. Nous donnons quelques exemples :

- “VIent nous Rejoindre dU côté Sombre” → “VIRUS”
- “Gare À Tout Autres Chanteur Anémique” → “GATACA”
- “PIerRe À touT Éviscéré” → “PIRATE”

2.a Solution algorithmique usuelle

Nous donnons ici, sans la justifier, une solution algorithmique usuelle. Il s’agit d’un simple schéma de parcours de la chaîne donnée.

```
1 public static String abrevMaj(String m) {  
2     String result = "";  
3     for(int i=0; i < m.length(); i++) {  
4         if (Character.isUpperCase(m.charAt(i))) {  
5             result += sansAccent(m.charAt(i));  
6         }  
7     }  
8     return result;  
9 }
```

Ici, nous ne donnerons pas la réalisation de la fonction `sansAccent` que nous supposerons être une méthode statique de la classe courante renvoyant le caractère passé sans son éventuel accent.

Comme vous pouvez le voir, cette méthode n’est pas très compliquée et pourtant elle agrège un certain nombre de préoccupations différentes :

- Parcourir (correctement) la séquence donnée,
- filtrer les majuscules,
- appliquer la méthode `sansAccent` à tous les caractères filtrés,
- collecter les caractères sans accents dans la chaîne résultat.

Ce code reste heureusement assez simple, mais le changement d’une spécification obligera le développeur à vérifier que le changement d’une des préoccupations de ce programme n’entraînera pas de nouveau bug dans les autres parties.

2.b Solution grossière utilisant une logique de flux

Dans la solution basée sur une logique de flux, nous allons nous appuyer sur un certain nombre de fonctions d’ordre supérieures intermédiaires. Pour simplifier notre propos nous supposerons que ces fonctions sont définies sur les séquences de caractères¹ :

- `map(Function<Character, Character> f)`: applique la fonction à tous les caractères de la chaîne courante et collecte les résultats dans une séquence résultat,
- `filter(Predicate<Character> p)`: collecte les caractères de la séquence qui satisfont le prédictat `p` et les mets dans une séquence résultats.

Si l’on dispose de ces 2 méthodes, il devient facile de construire la solution suivante :

```
1 public static String abrevMaj(String m) {  
2     return m.filter(Character::isUpperCase).map(MaClasse::sansAccent);  
3 }
```

Outre le fait que cette écriture est plus compacte, elle a aussi l’avantage de séparer différentes préoccupations.

- le parcours filtré est fait dans la méthode `filter`,
- le prédictat sélectionnant les caractères est la méthode statique `isUpperCase` de la classe `Character`.
- le parcours complet des caractères filtré est fait dans la méthode `map`,
- la fonction appliquée à chaque caractère est décrite dans la méthode `sansAccent`.

¹Ce n’est bien évidemment pas le cas sur le type `String` défini en java, mais nous pourrions définir notre propre classe de séquence de caractères.



Cet exemple n'est pas une bonne solution ! Il est uniquement là pour vous donner l'intuition de comment utiliser une logique de flux, mais les méthodes l'API Streams de java ne sont absolument pas implémentées de cette manière!!!
Lisez bien la suite du cours pour éviter les fausses croyances...

Cette solution permet de comprendre qu'en logique de flux, nous allons écrire des programmes qui *ressemblent* à un enchaînement d'actions atomiques appliquée à un flux de données. Dans cette exemple nous avons enchaîné un filtrage des majuscules et une suppression des accents.

Mais la solution telle que décrite est une mauvaise solution. En effet, telle que nous l'avons décrite, nous utilisons plus de mémoire que nécessaire, puisque nous avons construit une séquence intermédiaire (la séquence filtré).

Dans des cas réels où nous traitons des séquences très grandes et nous enchaînons de nombreux traitement, cela représente un coût en mémoire non négligeable (une grande séquence à chaque étape intermédiaire) et surtout parfaitement inutile puisque la solution naïve résout le problème sans créer de séquence intermédiaire.

2.c Solution à base d'itérateurs

Pour comprendre cette solution, il faut se rappeler qu'il y a une différence fondamentale entre une *séquence* (un objet conservant un certain nombre de données, par exemple un `ArrayList`) et le *parcours* d'une séquence.

Note



Pour bien comprendre cette différence, rappelez vous que l'on peut parcourir une séquence *qui n'est pas conservée en mémoire*, comme par exemple une séquence d'éléments générés au hasard ou une séquence de mots dans un texte.

Pour cette troisième solution, nous allons définir 3 itérateurs :

- un itérateur sur la séquence de *caractère* donnée,
- un itérateur sur les *caractères en majuscules* de la séquence donnée,
- un itérateur sur les *caractère majuscules sans accents* de la séquence donnée.

Itérateur sur la séquence Le premier itérateur est très simple et il s'appuie sur la séquence donnée (une `String`):

```

1 public class StringIterator implements Iterator<Character> {
2
3     private String texte;
4     private int i;
5
6     public StringIterator(String texte) {
7         this.texte = texte;
8         i = 0;
9     }
10
11    @Override
12    public boolean hasNext() {
13        return i < texte.length();
14    }
15
16    @Override
17    public Character next() {
18        return texte.charAt(i);
19        i++;
20    }
21 }
```

Il s'agit ici d'une implémentation très simple de l'interface `Iterator` sur une `String`.²

²Pour bien comprendre cette implémentation, rappelez-vous le fait que java va implicitement *mettre en boîte* le char renvoyé par la méthode `charAt` pour retourner une instance de la classe `Character`.

Itérateur filtrant Le second itérateur va filtrer et ne donner accès qu'aux caractères en majuscule. Pour le réaliser, l'astuce consiste à ne pas s'appuyer sur la chaîne originale, mais sur un itérateur de caractères quelconque. Nous verrons plus tard que c'est là une chose fondamentale.

```

1  public class UpperCaseIterator implements Iterator<Character> {
2
3      private Iterator<Character> text;
4      private Character next;
5      private boolean hasNext;
6
7      public UpperCaseIterator(Iterator<Character> text) {
8          this.text = text;
9          advanceToNext();
10     }
11
12     private void advanceToNext() {
13         hasNext = false; next = null;
14         while(text.hasNext() && !Character.isUpperCase(next = text.next())) ;
15         if (next != null && Character.isUpperCase(next)) {
16             hasNext = true;
17         }
18     }
19
20     @Override
21     public boolean hasNext() {
22         return hasNext;
23     }
24
25     @Override
26     public Character next() {
27         Character current = next;
28         advanceToNext();
29         return current;
30     }
31 }
```

Cet itérateur est un peu plus délicat à implémenter car il ne donne pas accès à tous les éléments de l'itérateur sous-jacent.³



Si vous regardez l'implémentation de cet itérateur vous verrez que le critère de filtrage n'apparaît qu'à un seul et unique endroit, dans le `while` de la méthode `advanceToNext()`. Tout le reste du code n'est là que pour implémenter l'interface `Iterator` avec un filtrage.

Itérateur modifiant Le dernier itérateur va devoir donner accès à tous les caractères majuscules sans accents. Pour cela nous allons construire un itérateur qui applique la fonction `sansAccents` à tous les éléments qu'il fournit. Comme l'itérateur précédent, lui-même va consommer ses éléments à partir d'un itérateur amont.

```

1  public class SansAccentIterator implements Iterator<Character> {
2
3      private Iterator<Character> text;
4
5      private static Character sansAccents(Character c) {
6          // ...
7      }
8      public SansAccentIterator(Iterator<Character> text) {
9          this.text = text;
10     }
11 }
```

³Pour bien comprendre cette implémentation, rappelez vous la sémantique exacte de l'expression `next = text.next()` (une assignation renvoie la valeur assignée). Vous constaterez de plus que le corps du `while` est vide, ce qui n'est pas très lisible, mais hélas nécessaire car la méthode `next` des itérateurs est une *fonction ayant un effet de bord*.

```

12  @Override
13  public boolean hasNext() {
14      return text.hasNext();
15  }
16
17  @Override
18  public Character next() {
19      return sansAccents(text.next());
20  }
21

```

Programme principal Le programme principal va se contenter de mettre en cascade ces 3 itérateurs puis itérer en récupérant à chaque fois un élément du résultat.

```

1  public static String abreviationMajuscule(String s) {
2      String result = "";
3      SansAccentIterator sai = new SansAccentIterator(new UpperCaseIterator(new
4          StringIterator("AbCdEfGhI")));
5      while(sai.hasNext()) {
6          result += sai.next();
7      }
8      return result;
}

```

Cette solution n'utilise pas de structure intermédiaire inutile puisqu'elle s'appuie sur une cascade d'itérateurs. Pour mieux comprendre le fonctionnement de cette solution, la séquence des appels de la cascade d'itérateurs est détaillée dans la figure 24.1.

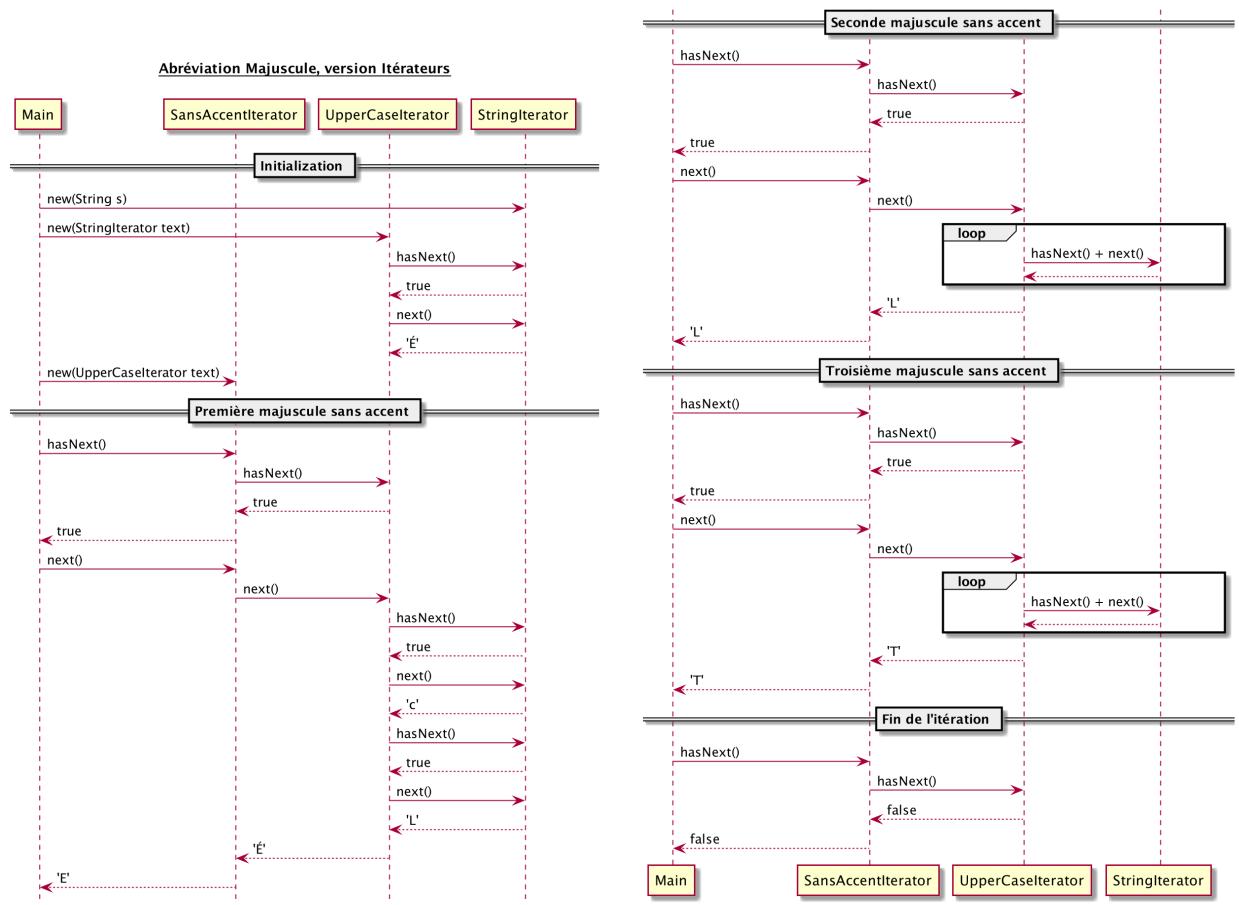


Figure 24.1: Séquence des appels de la cascade d'itérateurs sur la chaîne “ÉcLaTé”



Cette solution peut sembler complexe à écrire, mais vous aurez remarqué que pour résoudre un problème similaire, il suffira de changer simplement le prédictat de filtrage et la fonction à appliquer à chaque élément. Tous le reste du programme restera inchangé !

2.d Généralisation de la solution précédente

La solution à base d'itérateurs est intéressante mais elle mélange encore 2 préoccupations dans chaque itérateur: le filtrage/parcours et le prédictat/la fonction à appliquer. Heureusement, nous disposons de tous les ingrédient pour séparer ces 2 préoccupations.

Itérateur filtrant générique Afin de séparer la *mécanique* de filtrage du *critère* de filtrage, il suffit de se donner un prédictat générique s'appliquant sur les éléments de l'itérateur amont. Nous avons vu au chapitre 23 comment écrire des classes/méthodes d'ordre supérieur. Nous pouvons donc définir un itérateur filtrant sur n'importe quel critère. Nous en profitons pour rendre cet itérateur générique sur le type d'élément itérés.

```

1  public class FilteringIterator<T> implements Iterator<T> {
2
3      private Iterator<T> upstream;
4      private T next;
5      private boolean hasNext;
6      private Predicate<T> filter;
7
8      public FilteringIterator(Iterator<T> upstream, Predicate<T> filter) {
9          this.upstream = upstream;
10         this.filter = filter;
11         advanceToNext();
12     }
13
14     private void advanceToNext() {
15         hasNext = false; next = null;
16         while(upstream.hasNext() && ! filter.test(next = upstream.next())) ;
17         if (next != null && filter.test(next)) { hasNext = true; }
18     }
19
20     @Override
21     public boolean hasNext() { return hasNext; }
22
23     @Override
24     public T next() {
25         T current = next;
26         advanceToNext();
27         return current;
28     }
29 }
```

Itérateur modifiant générique De la même manière, on peut utiliser l'ordre supérieur pour séparer la mécanique de parcours et la fonction de modification de l'itérateur modifiant. Nous en profitons ici aussi pour rendre cet itérateur générique sur le type des éléments en amont et sur le type des éléments en aval.⁴

```

1  public class MapIterator<T, R> implements Iterator<R> {
2
3      private Iterator<T> upstream;
4      private Function<T, R> transformer;
5
6      public MapIterator(Iterator<T> upstream, Function<T, R> transformer) {
7          this.upstream = upstream;
8          this.transformer = transformer;
9      }
```

⁴En effet, la fonction de transformation peut avoir une image différente de son domaine. En termes Java, la fonction n'est pas forcément un opérateur.

```

10
11     @Override
12     public boolean hasNext() { return upstream.hasNext(); }
13
14     @Override
15     public R next() {
16         return transformer.apply(upstream.next());
17     }
18 }
```

Le programme principal Ici aussi le programme principal se contentera de créer les itérateurs puis de réaliser l’itération sur le résultat. La principale différence est que c’est lui qui indiquera le critère souhaité à l’itérateur filtrant et la transformations souhaitée à l’itérateur modifiant.

```

1  public static String testIterator2(String text) {
2      String result = "";
3      StringIterator original = new StringIterator(text);
4      FilteringIterator<Character> filtrage = new FilteringIterator<>(original,
5          Character::isUpperCase);
6      MapIterator<Character, Character> transformer = new MapIterator<>(filtrage,
7          c -> sansAccents(c));
8      while(transformer.hasNext()) {
9          result += transformer.next();
10     }
11     return result;
12 }
```

Vous avez maintenant tous les ingrédients d’une solution adaptée. En effet, elle répond au problème (ce qui est le moindre des choses), mais aussi et surtout, elle sépare les préoccupations de manière idéale :

- l’itérateur filtrant ne contient que le code de filtrage
- le critère de filtrage est défini dans une classe indépendante de l’itérateur
- l’itérateur modifiant ne contient que le code de parcours
- la transformation à appliquer à chaque élément est définie dans une classe indépendante de l’itérateur
- le `StringIterator` est le seul itérateur qui ait accès à la chaîne initiale.



Vous remarquerez de plus que les classes `FileringIterator` et `MapIterator` ne dépendent en rien du programme principal. Ils peuvent être réutilisé pour n’importe quel problème que l’on peut résoudre avec une logique de flux.

2.e Quelques remarques sur cette solution

L’API des Streams de Java *ressemble* dans son principe à la solution esquissée ci dessus. Elle se base sur la mise en cascade d’itérateurs qui appliquent des filtrages, transformations ou autres opérations. La différence majeure de l’API des Streams est qu’elle est bien plus complexe dans son implémentation⁵ que la solution esquissée ici. En effet, elle n’utilise pas de simples itérateurs car elle permet, lorsque cela est possible d’appliquer des traitement concurrents (en parallèle) qui permettent de tirer profit des capacités multi-cœurs de la quasi totalité des processeurs actuels.

Néanmoins les ingrédients de l’API des Streams sont à peu de chose près les même que ceux de la solution à base d’itérateurs génériques ci dessus. On distingue 3 catégories d’ingrédients dans cette solution :

- Tout d’abord, une Collection d’élément et un premier itérateur sur cette collection (ici `StringIterator`),
- une cascade d’un ou plusieurs itérateurs qui filtrent ou transforment le flux d’itération,
- une boucle principal qui consomme les éléments fournis au travers de la cascade précédente.

Nous verrons que ces 3 catégories d’ingrédients se retrouveront dans les outils de l’API des Streams de Java.

⁵mais plus simple dans son utilisation

Note

Vous aurez remarqué que dans cette chaîne de traitement, seule la boucle principale *consomme* des éléments de la collection. Si cette boucle n'est pas exécutée ou si elle ne parcourt pas tous les éléments (cas d'un schéma de recherche par exemple), alors les éléments restant ne sont jamais consommés, filtrés ou transformés. On dit que ce modèle est *paresseux* (lazy en anglais) car il ne consomme les éléments que lorsque cela s'avère nécessaire.

3 Les Streams en Java

3.a Un exemple d'utilisation des Streams en Java

L'API Streams permet d'effectuer une cascade de traitement sur les éléments d'une collection. Dans les exemples suivants, nous utiliserons une classe hypothétique **Ville**, munie des attributs **nom** (une **String**), **population** (un **long**) et **département** (un **int**), ainsi que des constructeurs et accesseurs nécessaires.

```

1 List<String> villes = Arrays.asList(
2     new Ville("Valence", 63714, 26),
3     new Ville("Grenoble", 158454, 38),
4     new Ville("Privas", 8266, 07),
5     new Ville("Saint Marcellin", 7903, 38),
6     new Ville("Romans sur Isère", 33160, 26),
7     new Ville("Saint Martin d'Hères", 38487, 38),
8     ...
9
10    // nombre de villes de l'Ardèche
11    long nbVilleArdeche = villes
12        .stream()
13        .filter(v -> v.getDepartement() == 7)
14        .count();
15    System.out.println(nbVilleArdeche);
16    // 1
17
18    // Population totale des villes de la Drôme
19    long populationDrome = villes
20        .stream()
21        .filter(v -> v.getDepartement() == 26)
22        .mapToLong(Ville::getPopulation)
23        .sum();
24    System.out.println(populationDrome);
25    // 96874
26
27    // Afficher les noms des villes de l'Isère de plus de 30000 habitant, en
28    // majuscules
29    villes.stream()
30        .filter(v -> v.getDepartement() == 38)
31        .filter(v -> v.getPopulation() >= 30000)
32        .map(v -> v.getNom().toUpperCase())
33        .forEach(System.out::println);
34    // GRENOBLE
35    // SAINT MARTIN D'HERES

```

Dans chacun de ces exemples, on identifie 3 catégories de méthodes distinctes :

- La création d'un **Stream**. Ici en utilisant la méthode **stream** définie sur toutes les **List** de l'API des collections Java.
- Plusieurs opérations (en cascade) sur le stream créé. Ici, les méthodes **filter**, **map**, **mapToLong**, que Java appelle des *opérations intermédiaires*.
- Une (et une seule) *opération terminale* qui contient l'itération consommant le Stream. Ici, les méthodes **count**, **sum** et **forEach**.



Il est à noter que les opérations terminales consomment les éléments fournis par le Stream, puis elles le ferment. Ainsi, il est impossible d'utiliser 2 opérations terminales sur un même Stream, la seconde lancera une `java.lang.IllegalStateException`

```

1 Stream<String> stream = villes.stream()
2   .filter(v -> v.getDepartement() == 38)
3   .filter(v -> v.getPopulation() >= 30000)
4   .map(v -> v.getNom().toUpperCase());
5 stream.forEach(System.out::println); // OK
6 stream.count(); // java.lang.IllegalStateException

```



Attention, ce cours n'est en rien une référence exhaustive de l'API des Streams de Java. Cette API contient de nombreuses méthodes et classes non abordées ici.

4 Crédation d'un Stream

La première opération nécessaire est de *créer* un stream. Un Stream peut être créé à partir

- d'une Collection,
- d'un tableau,
- d'un ensemble de données,
- d'un fichier,
- d'une librairie tiers,
- ...

Il est aussi possible de créer un Stream à partir d'une fonction fournissant les données. Cela permet d'ailleurs de définir des Stream de taille *infinie*.

4.a Les méthodes statiques de construction de la classe Stream

Stream.of()

La méthode `of` de la classe `Stream` a un nombre d'arguments variables (obtenu avec le mot clé `...`).

Exemple	Remarques
<code>Stream<String> s = Stream.of("Bonjour", "à", "tous");</code>	Un Stream donnant accès à 3 chaînes.
<code>Stream<String> s = Stream.of(new String[]{"Bonjour", "à", "tous"});</code>	Idem à partir d'un tableau.
<code>Stream<Integer> s3 = Stream.of(1,2,3,4,5);</code>	Attention, c'est un Stream d' <code>Object</code> on a donc des <code>Integer</code> et non des <code>int</code> .
<code>IntStream s4 = IntStream.of(1,2,3,4,5);</code>	Il existe des classes de Stream pour les types primitifs.

Stream.iterate()

La méthode `iterate` de la classe `Stream` permet de construire une suite d'éléments à partir d'un élément initial et d'une fonction calculant la valeur suivante en fonction de la précédente.

Exemple	Remarques
<code>Stream<Integer> entiers = Stream.iterate(0, n -> n+1);</code>	Un Stream infini donnant les valeurs entières successives. Attention, il s'agit toujours d' <code>Object</code> , donc des <code>Integer</code> et non des <code>int</code> .
<code>IntStream entiers2 = IntStream.iterate(1, n -> 2*n);</code>	La méthode est aussi définie sur les streams de primitifs. Ici la série des puissances de 2.

La méthode `generate` de la classe `Stream` permet de construire une suite d'éléments à partir d'un `Supplier`.

Exemple	Remarques
<code>Stream<Double> doubles = Stream.generate(Math::random);</code>	Un Stream infini donnant les valeurs <code>Double</code> aléatoire. Attention, il s'agit toujours d' <code>Object</code> , donc des <code>Double</code> et non des <code>double</code> .
<code>DoubleStream doubles2 = DoubleStream.generate(Math::random);</code>	La méthode est aussi définie sur les streams de primitifs.

4.b Créer un `Stream` à partir d'une collection

L'ensemble des collections de l'API standard Java permettent de créer un stream :

- `stream()` permet de créer un `Stream` standard à partir des éléments de la collection,
- `parallelStream()` permet de créer un `Stream` pouvant être consommé en parallèle,

5 Opérations intermédiaires

L'API des Streams en java défini de nombreuses opérations intermédiaires. Nous reprenons ici les plus utilisées⁶:

Opérations <i>stateless</i> (s'appliquant à chaque élément indépendamment)	
Exemple	Remarques
<code>map(Function<T,R> f)</code>	Applique la fonction <code>f</code> à tous les éléments du <code>Stream</code> amont.
<code>mapToInt(ToIntFunction<T> f)</code>	lorsqu'une fonction <code>f</code> produit le type primitif <code>int</code> .
<code>mapToDouble(ToDoubleFunction<T> f)</code>	lorsqu'une fonction <code>f</code> produit le type primitif <code>double</code> .
<code>flatMap(Function<T,Stream<R>> mapper)</code>	La fonction <code>mapper</code> produit un <code>Stream</code> à partir d'un élément. Le Stream résultant du <code>flatMap</code> est la <i>concaténation</i> des Streams produits pour chaque élément.
<code>filter(Predicate<T> p)</code>	ne conserve que les éléments satisfaisant <code>p</code> .
<code>peek(Consumer <? super T> c)</code>	La fonction <code>c</code> est appliquée à chaque élément, mais le <code>Stream</code> passe de manière inchangée. <code>c</code> est un <code>Consumer</code> , elle n'a d'intérêt que pour ses effets de bord.

⁶Attention, dans ce tableau et dans les suivants, les profils de fonctions ont été simplifiés afin de gagner de la place (enlevant notamment les détails d'héritages de types dans les types génériques)

Opérations <i>statefull</i> (tenant en compte l'ensemble des éléments)	
Exemple	Remarques
<code>distinct()</code>	Élimine les doublons du stream amont.
<code>sorted()</code>	les éléments du Stream résultat seront fournis dans leur ordre naturel (ils doivent être Comparable).
<code>sorted(Comparator<? super T> c)</code>	les éléments du Stream résultat seront fournis dans l'ordre défini par le comparateur.
<code>limit(long n)</code>	le Stream se terminera après, au plus, les n premiers éléments (bien utile sur un stream infini par exemple).
<code>skip(long n)</code>	les n premiers éléments seront ignorés.
<code>sequential()</code>	rendre séquentiel le Stream (utile s'il était parallèle par exemple),
<code>parallel()</code>	rendre parallèle le Stream (les actions suivantes pourront être exécutées en parallèle sur différents éléments),
<code>unordered()</code>	indiquer que l'ordre n'a pas d'importance sur le Stream. Attention, cela ne "mélange" pas le Stream, mais si il est consommé en parallèle, l'API ne fera pas l'effort de maintenir l'ordre des éléments (ceux ci arriveront dans l'ordre où ils sont disponibles après les traitements).
<code>onClose()</code>	permet d'enregistrer une action à faire lorsque le Stream sera fermé (par un appel à la méthode <code>close()</code>), ce qui est le cas lorsque l'opération terminale a fini d'itérer.

5.a les opérations `map()` et `flatMap()`

L'opération intermédiaire la plus délicate à maîtriser est l'opération `flatMap()`. Pour la comprendre, nous allons la comparer à la méthode `map()`, plus facile à comprendre.

Pour cela, nous allons utiliser la lambda expression : `n -> Arrays.asList(n, n*n)` qui prend un Integer `n` en paramètre et renvoie un `List<Integer>` constitué du nombre `n`, suivi de son carré. Il s'agit donc d'une fonction de type `Integer -> List<Integer>`.

Dans un premier temps, appliquons cette lambda expression à un `Stream<Integer>` et affichons chaque élément du Stream résultant :

```

1 List<Integer> nombres = Arrays.asList(1, 2, 3, 4);
2 nombres.stream()
3   .map(n -> Arrays.asList(n, n*n))
4   .forEach(System.out::println);
5 // [1, 1]
6 // [2, 4]
7 // [3, 9]
8 // [4, 16]
```

Dans cet exemple, après la méthode `map()`, le Stream est un `Stream<List<Integer>>`, en effet, chaque élément du Stream amont (chaque `Integer`) est substitué par une `List<Integer>`. L'effet est donc l'affichage de 4 listes.

Maintenant, appliquons là en utilisant la méthode `flatMap()`:

```

1 List<Integer> nombres = Arrays.asList(1, 2, 3, 4);
2 nombres.stream()
3   .flatMap(n -> Arrays.asList(n, n*n).stream())
4   .forEach(System.out::println);
5 // 1
6 // 1
7 // 2
8 // 4
9 // 3
10 // 9
11 // 4
12 // 16
```

Ici, nous avons adapté la lambda expression afin qu'elle renvoie le Stream associé à la liste $[n, n^2]$, ce qui est le type de retour attendu par la méthode `flatMap()`.

Chaque application de la lambda expression créer une séquence, mais ici, la séquence est concaténée dans le flux résultat qui reste un `Stream<Integer>`. L'effet est donc l'affichage de 8 valeurs entières.

6 Opérations terminales

Les opérations terminales sont celles qui *consomment* les éléments du `Stream`. Rappelez vous que c'est l'opération qui conclue la chaîne de traitement. Rappelez vous aussi que les `Streams` sont consommés de manière paresseuse et qu'aucun élément ne sera consommé ni aucun traitement n'aura lieu en l'absence d'une opération terminale en fin de chaîne.

Les opérations terminales peuvent renvoyer des résultats de différents type (primitifs ou non), ou collecter les éléments du `Stream` dans une collection. Pour certaines d'entre elles, le résultat peut ne pas être défini, comme par exemple calculer l'élément maximum d'un `Stream` vide. Dans ce cas, la méthode renvoie un objet `Optional<R>` qui représente un résultat optionnel de type `R`. Nous détaillerons cette classe dans la suite.

Nous reprenons ici (de manière non exhaustive) les principales fonctions terminales.

Exemple	Remarques
<code>void forEach(Consumer<T> action)</code>	Applique l' <code>action</code> à tous les éléments du <code>Stream</code> sans renvoyer de résultat.
<code>void forEachOrdered(Consumer<T> action)</code>	Applique l' <code>action</code> à tous les éléments du <code>Stream</code> sans renvoyer de résultat. Cette méthode maintiendra l'ordre d'application à l'ordre des éléments du <code>Stream</code> contairement à la précédente qui ne garantit pas l'ordre de traitement en cas de <code>Stream</code> arallèle.
<code>T[] toArray()</code>	renvoie un tableau contenant les éléments du <code>Stream<T></code> .
<code>Optional<T> reduce(BinaryOperator<T> accumulator)</code>	réduit le <code>Stream</code> en une valeur en appliquant successivement la fonction <code>accumulator</code> sur les éléments. S'il n'y a pas d'élément, n'a pas de résultat.
<code>T reduce(T identity, BinaryOperator<T> accumulator)</code>	réduit le <code>Stream</code> en une valeur en appliquant successivement la fonction <code>accumulator</code> sur les éléments, à partir d'une valeur initial <code>identity</code> . S'il n'y a pas d'élément, le résultat est <code>identity</code> .
<code>R collect(Collector<T,A,R> collector)</code>	collecte les éléments du <code>Stream</code> en une collection construite par l'objet de classe <code>Collector</code> fourni. L'API Java définit un certain nombre de <code>Collector</code> prédéfinis.
<code>Optional<T> min(Comparator<T> comparator)</code>	renvoie le plus petit élément selon l'ordre défini par le <code>Comparator</code> fourni.
<code>Optional<T> max(Comparator<T> comparator)</code>	renvoie le plus grand élément selon l'ordre défini par le <code>Comparator</code> fourni.
<code>long count()</code>	renvoie le nombre d'éléments du <code>Stream</code> .
<code>boolean anyMatch(Predicate<T> p)</code>	renvoie vrai si un élément du <code>Stream</code> satisfait <code>p</code> .
<code>boolean allMatch(Predicate<T> p)</code>	renvoie vrai si tous les éléments du <code>Stream</code> satisfont <code>p</code> .
<code>boolean noneMatch(Predicate<T> p)</code>	renvoie vrai si aucun élément du <code>Stream</code> ne satisfait <code>p</code> .
<code>Optional<T> findFirst()</code>	renvoie le premier élément du <code>Stream</code> . Si celui-ci ne garantit pas d'ordre, n'importe quel élément pourra être renvoyé (le premier disponible en cas de traitement parallèle).
<code>Optional<T> findAny()</code>	renvoie un élément quelconque du <code>Stream</code> .
<code>Iterator<T> iterator()</code>	renvoie un itérateur sur les éléments du <code>Stream</code> .

Dans la suite, nous détaillons les éléments les plus délicats.

6.a Les Optional

Comme indiqué ci dessus, certaines méthodes n'ont pas de résultat défini dans certaines conditions. Ces fonctions retournent un résultat de type `Optional<T>`.

Un `Optional<T>` est un objet qui peut avoir deux états :

- soit il ne contient pas de valeur de résultat
- soit il contient une valeur de type `T`

Un `Optional<T>` fournit les méthodes suivantes:

- `boolean isPresent()` retourne vrai si l'`Optional` a une valeur,
- `T get()` retourne la valeur si elle est présente, sinon lève une exception,
- `void ifPresent(Consumer<? super T> consumer)` applique `consumer` à la valeur si elle est présente,
- `T orElse(T other)` retourne la valeur de l'`Optional` si présente, `other` sinon.

Il est intéressant de noter qu'un `Optional` se comporte presque comme un `Stream` à 0 ou 1 élément et définit les fonctions `map()`, `flatMap()` et `filter()`.

Si vous devez construire un `Optional`, vous pouvez le faire avec les méthodes statiques suivantes :

- `Optional.empty()` retourne un `Optional` vide,
- `Optional.of(T value)` retourne un `Optional` contenant la valeur `value`,
- `Optional.ofNullable(T value)` retourne un `Optional` contenant la valeur `value` si `value` est non `null` ou vide si `value` est `null`.

6.b L'opération `reduce`

L'opération la plus délicate à comprendre est l'opération `reduce()`. Cette opération permet de consommer les éléments du `Stream` en calculant une valeur au fur et à mesure de l'avancement dans le `Stream`.

Son principe de base est que pour chaque élément consommé dans le `Stream`, il faut réaliser un calcul qui tient compte de l'élément et de la valeur intermédiaire calculée pour les éléments précédents.

Nous illustrons cette opération sur des exemples manipulant des `Villes` telles que définies au paragraphe 3.a:

reduce(BinaryOperator<T>) Il s'agit de la forme la plus simple. Les éléments consommés et l'accumulateur ont le même type.

Par exemple, obtenir la population totalisée par les villes de la Drôme:

```
1 villes.stream()
2   .filter(v -> v.getDepartement() == 38)
3   .map(Ville::getPopulation)
4   .reduce((n,m) -> n + m);
```

Dans cet exemple, l'opération `reduce` s'applique sur un `Stream<Integer>` et son résultat final est un `Integer`. Après filtrage des villes de l'Isère et obtention de leurs populations, on obtient un `Stream` qui fournira les valeurs 158454, 7903 et 38487. L'opération `reduce` consomme la première valeur qui devient la valeur initiale d'un accumulateur. Elle consomme ensuite la seconde valeur et appelle la fonction avec l'accumulateur en premier argument `n` et l'élément courant en second argument `m`. Le résultat (166357) est conservé dans l'accumulateur et ainsi de suite jusqu'au dernier élément.

On peut utiliser cette même opération pour obtenir la population maximale des villes de la Drôme :

```
1 villes.stream()
2   .filter(v -> v.getDepartement() == 38)
3   .map(Ville::getPopulation)
4   .reduce((n,m) -> n > m ? n : m);
```

Dans cet exemple, la fonction d'accumulation se contente de renvoyer la valeur la plus grande entre l'accumulateur et l'élément courant. Ainsi, l'accumulateur contiendra toujours la valeur maximale des éléments déjà vus et après consommation totale du `Stream` elle contiendra la valeur maximale des éléments du `Stream`.



Cette version de l'opération `reduce` n'a pas de valeur si le `Stream` est vide. C'est la raison pour laquelle elle renvoie un `Optional<T>`. Ceci est tout à fait souhaitable dans le second exemple, où la valeur maximale *n'est pas définie* pour un `Stream` vide. Par contre, la somme d'une séquence d'entier est définie même pour une séquence vide. Dans ce cas, nous préférerons la seconde forme décrite ci dessous.

reduce(T identity, BinaryOperator<T>) Cette forme prend deux arguments. Ici aussi, les éléments consommés et l'accumulateur ont le même type.

Ré-écrivons l'exemple permettant d'obtenir la population totalisée par les villes de la Drôme:

```
1 villes.stream()
2   .filter(v -> v.getDepartement() == 38)
3   .map(Ville::getPopulation)
4   .reduce(0, (n,m) -> n + m);
```

Dans cette version, le premier argument donne la valeur initiale de l'accumulateur (qui sera donc la valeur de retour si la séquence est vide). Cette version renverra la valeur 0 si la séquence est vide alors que la version précédente aurait renvoyé un `Optional` vide.

6.c Les Collectors

Afin de transformer un `Stream` en une collection via l'opération terminal `collect`, Java fourni des `Collector` prédéfinis via les méthodes statiques de la classe `Collectors`⁷.

Nous illustrons ces `Collectors` sur des exemples manipulant des `Villes` telles que définies au paragraphe 3.a.

- `Collectors.toSet()` pour accumuler les éléments dans un ensemble. Par exemple, obtenir l'ensemble des villes d'Ardèche (on profite qu'un ensemble n'a pas d'ordre pour utiliser un stream non ordonné et parallèle):

```
1  villes.parallelStream().filter(v -> v.getDepartement() == 7)
2    .collect(Collectors.toSet());
```

- `Collectors.toList()` pour accumuler les éléments dans une liste. Par exemple, obtenir la liste des villes de la Drôme triées par ordre alphabétique:

```
1  villes.stream()
2    .filter(v -> v.getDepartement() == 26)
3    .sorted((x, y) -> x.getNom().compareTo(y.getNom()))
4    .collect(Collectors.toList());
```

- `Collectors.toCollection(Supplier<Collection<T>> supplier)` renvoie la collection produite par `supplier`, remplie des éléments du `Stream`. Par exemple, créer un `TreeSet`⁸ contenant les villes de la Drôme en conservant leur ordre alphabétique:

```
1  villes.stream()
2    .filter(v -> v.getDepartement() == 26)
3    .sorted((x, y) -> x.getNom().compareTo(y.getNom()))
4    .collect(Collectors.toCollection(TreeSet::new));
```

- `Collectors.toMap(Function<T,K> keyMapper, Function<T,U> valueMapper)` renvoie une `Map<K, U>` construite en calculant clé et valeur en appliquant respectivement la fonction `keyMapper` et `valueMapper` à chaque élément. Par exemple, pour créer une `Map<String, Integer>` associant à chaque ville Iséroise sa population:

```
1  villes.stream()
2    .filter(v -> v.getDepartement() == 38)
3    .collect(Collectors.toMap(Ville::getNom, Ville::getPopulation));
```

Dans cette opération, si deux éléments ont la même clé (au sens de la méthode `equals()`), une exception sera levée.

- `Collectors.toMap(Function<T,K> keyMapper, Function<T,U> valueMapper, BinaryOperator<U> merge)` renvoie une `Map<K, U>` construite en calculant clé et valeur en appliquant respectivement la fonction `keyMapper` et `valueMapper` à chaque élément. Dans cette opération, si deux éléments ont la même clé (au sens de la méthode `equals()`), la fonction `merge` sera appelée pour gérer le conflit.

Par exemple, pour créer une `Map<Integer, String>` associant à chaque valeur de population le ou les nom(s) de ville(s) (séparés par une virgule) :

```
1  villes.stream()
2    .filter(v -> v.getDepartement() == 38)
3    .collect(Collectors.toMap(
4      Ville::getPopulation,
5      Ville::getNom,
6      (n1, n2) -> n1 + ", " + n2));
```

- `joining()` et `joining(CharSequence delimiter)` permettent de renvoyer une chaîne de caractère en concaténant la représentation en `String` de chaque élément dans l'ordre où il est consommé (en les séparant par `delimiter` le cas échéant). Par exemple, pour créer une chaîne de caractère composée des noms de villes de l'Ardèche séparés par une virgule:

⁷Attention: la classe `Collectors` (avec un "s") fournit des instance de `Collector` (sans "s").

⁸Rappel: un `TreeSet` est un ensemble qui conserve l'ordre dans lequel ses éléments ont été insérés.

```
1  villes.stream()
2      .filter(v -> v.getDepartement() == 07)
3      .map(Ville::getNom)
4      .collect(Collectors.joining(" ", " "));
```