

# Programmation Orientée Objets en Java

Olivier Camp

8 décembre 2022

# Table des matières

<b>1 Le langage Java</b>	<b>10</b>
1.1 Les origines de Java ? . . . . .	10
1.2 Un langage simple . . . . .	11
1.2.1 Un ensemble de types prédéfinis . . . . .	11
1.2.2 Des structures de contrôle . . . . .	12
1.2.2.1 Commentaires . . . . .	15
1.2.2.2 Formatage du code Java . . . . .	16
1.2.3 Un langage orienté objets . . . . .	16
1.2.4 Un langage interprété et portable . . . . .	17
1.3 Premiers programmes . . . . .	18
1.3.1 Une classe pour saluer l'utilisateur . . . . .	18
1.3.1.1 Quelques explications sur la définition de la classe <i>Hello</i> . . . . .	20
1.3.1.2 Les flux (ou canaux) d'entrée/sortie . . . . .	21
1.3.2 Manipulation des arguments passés sur la ligne de commande	23
1.3.3 Quelques exercices . . . . .	24
1.4 Classes et Instanciation . . . . .	25
1.4.1 Définition d'une classe . . . . .	25
1.4.1.1 Variables d'instance et variables de classe . . . . .	26
1.4.1.2 Méthodes d'instance et méthodes de classe . . . . .	27
1.4.1.3 Accesseurs et mutateurs . . . . .	27
1.4.1.4 La classe <i>Personne</i> . . . . .	28
1.4.1.5 Envoi de messages . . . . .	30
1.4.1.6 Un objet particulier : <i>this</i> . . . . .	30
1.4.1.7 L'instanciation et le constructeur par défaut . . . . .	31
1.4.1.8 Cas particulier de l'instanciation pour les classes "génériques" . . . . .	32
1.4.2 Quelques exercices . . . . .	33
1.5 Constructeurs . . . . .	35
1.5.1 Un premier constructeur . . . . .	36
1.5.1.1 Un mot sur la portée des variables . . . . .	37
1.5.2 Utilisation des constructeurs . . . . .	38
1.5.3 Un deuxième constructeur . . . . .	39
1.5.4 Factorisation du code des constructeurs avec <i>this(...)</i> . . . . .	39
1.5.5 À savoir absolument sur les Constructeurs . . . . .	40
1.5.6 Surcharge (overloading) de méthodes et constructeurs . . . . .	40
1.5.7 Quelques exercices . . . . .	41
1.6 Paquetages ( <i>packages</i> ) . . . . .	42

1.6.1	Paquetages et classes de l'API . . . . .	43
1.6.1.1	Nommage des paquetages . . . . .	43
1.6.1.2	Nommage complet des classes et importation de classes	44
1.6.2	Définition du paquetage d'une classe . . . . .	45
1.6.2.1	Paquetages et noms de répertoires . . . . .	45
1.6.2.2	Déclaration de l'appartenance d'une classe à un paquetage . . . . .	46
1.6.2.3	Compilation et exécution d'une classe dans un paquetage . . . . .	46
1.6.3	Quelques exercices . . . . .	47
1.7	Encapsulation . . . . .	48
1.7.1	Un exemple d'encapsulation : une pile de livres . . . . .	48
1.7.2	Les modificateurs de visibilité de Java . . . . .	51
1.7.2.1	Absence de modificateur . . . . .	51
1.7.2.2	Le modificateur <i>public</i> . . . . .	52
1.7.2.3	Le modificateur <i>private</i> . . . . .	53
1.7.2.4	Le modificateur <i>protected</i> . . . . .	54
1.7.3	Des bonnes pratiques . . . . .	56
1.7.4	Quelques explications sur la méthode <i>main</i> . . . . .	57
1.7.5	Quelques exercices . . . . .	58
1.8	L'héritage . . . . .	62
1.8.1	Héritage, envoi de messages et recherche de méthode . . . . .	65
1.8.2	Redéfinition (overriding) de méthodes . . . . .	65
1.8.3	La méthode <i>toString()</i> de la classe <i>java.lang.Object</i> . . . . .	65
1.8.3.1	Appel automatique de la méthode <i>toString()</i> . . . . .	66
1.8.4	L'objet <i>super</i> . . . . .	67
1.8.4.1	Exemple d'utilisation de l'objet <i>super</i> . . . . .	67
1.8.5	Masquage de variables d'instance . . . . .	69
1.8.6	Héritage et polymorphisme . . . . .	75
1.8.7	Héritage et constructeurs : le constructeur <i>super()</i> . . . . .	75
1.8.7.1	À savoir absolument sur les Constructeurs . . . . .	77
1.8.7.2	Héritages et constructeurs : on récapitule . . . . .	77
1.8.8	Quelques exercices . . . . .	79
1.9	Les classes abstraites . . . . .	81
1.9.1	Définition d'un classe abstraite . . . . .	83
1.9.2	Constructeurs et classes abstraites . . . . .	86
1.9.3	Quelques exercices . . . . .	86
1.10	Les interfaces . . . . .	89
1.10.1	Implémentation d'interfaces . . . . .	90
1.10.2	Héritage d'interfaces . . . . .	90
1.10.3	Utilisation d'une interface . . . . .	92
1.10.4	Un mot sur le framework de collections de Java . . . . .	93
1.10.4.1	Les listes de l'interface <i>java.util.List</i> . . . . .	93
1.10.4.2	Les ensembles de l'interface <i>Set</i> . . . . .	93
1.10.4.3	Les files de l'interface <i>Queue</i> . . . . .	94
1.10.4.4	Les dictionnaires de l'interface <i>Map</i> . . . . .	94
1.11	Les exceptions et la gestion d'erreur . . . . .	94
1.11.1	Les classes d'exceptions . . . . .	94
1.11.1.1	Fonctionnement et gestion des exceptions . . . . .	95

<b>I Annexes</b>	<b>96</b>
<b>A Nommage et indentation</b>	<b>97</b>
A.1 La notation <i>Camel Case</i> . . . . .	97
A.2 Des noms parlants . . . . .	98
A.3 Majuscules et minuscules . . . . .	98
A.3.1 Nom d'une classe ou d'une interface : . . . . .	98
A.3.2 Nom d'une variable : . . . . .	98
A.3.3 Nom d'une constante de classe : . . . . .	98
A.3.4 Nom d'un paquetage : . . . . .	98
A.4 Indentation d'un programme Java . . . . .	98
<b>B Java Google style</b>	<b>99</b>
B.1 Braces . . . . .	99
B.1.1 Braces are used where optional . . . . .	99
B.1.2 Nonempty blocks : K & R style . . . . .	99
B.1.3 Empty blocks : may be concise . . . . .	100
B.2 Block indentation : +2 spaces . . . . .	100
B.3 One statement per line . . . . .	100
B.4 Column limit : 80 or 100 . . . . .	100
B.5 Line-wrapping . . . . .	101
B.5.1 Where to break . . . . .	101
B.5.2 Indent continuation lines at least +4 spaces . . . . .	101
B.6 Whitespace . . . . .	101
B.6.1 Vertical Whitespace . . . . .	101
B.6.2 Horizontal whitespace . . . . .	102
B.6.3 Horizontal alignment : never required . . . . .	102
B.7 Grouping parentheses : recommended . . . . .	103
B.8 Specific constructs . . . . .	103
B.8.1 Enum classes . . . . .	103
B.8.2 Variable declarations . . . . .	103
B.8.2.1 One variable per declaration . . . . .	103
B.8.2.2 Declared when needed, initialized as soon as possible	103
B.8.3 Arrays . . . . .	104
B.8.3.1 Array initializers : can be "block-like" . . . . .	104
B.8.3.2 No C-style array declarations . . . . .	104
B.8.3.3 Switch statements . . . . .	104
B.8.3.4 Indentation . . . . .	104
B.8.3.5 Fall-through : commented . . . . .	104
B.8.3.6 The default case is present . . . . .	105
B.8.4 Annotations . . . . .	105
B.8.5 Comments . . . . .	105
B.8.5.1 Block comment style . . . . .	105
B.8.6 Modifiers . . . . .	106
B.8.7 Numeric Literals . . . . .	106

<b>C Les commentaires <i>javadoc</i></b>	<b>107</b>
C.1 JAVA DOCUMENTATION COMMENTS . . . . .	107
C.1.1 What is Javadoc ? . . . . .	107
C.1.2 The javadoc Tags : . . . . .	108
C.1.2.1 Example : . . . . .	109
<b>D De UML à Java</b>	<b>112</b>
D.1 Les classes . . . . .	112
D.1.1 Classes et méthodes abstraites . . . . .	113
D.2 Les associations entre classes . . . . .	114
D.3 L'héritage . . . . .	114
D.4 Les paquetages . . . . .	115
D.5 Les interfaces . . . . .	116

# Introduction

Un ordinateur ne sait exécuter que des programmes exprimés en binaire sous la forme de séquences de '0' et de '1'. Les développeurs de programmes informatiques, eux, des humains, utilisent des moyens plus élaborés pour exprimer des idées : le langage naturel avec lequel nous communiquons quotidiennement.

Ainsi, l'écriture d'un programme exécutable peut être vu comme une activité de traduction qui part d'une idée exprimée dans un langage naturel pour arriver à un programme exécutable exprimé sous la forme d'une séquence de chiffres binaires. Les premiers programmeurs, au début des années 1940, se chargeaient seuls d'effectuer cette traduction et devaient écrire leurs programmes directement en binaire. Ce n'était pas une tâche simple et les programmes n'étaient pas très lisibles (figure 1).

FIGURE 1 – Notation binaire

La notation hexadécimale qui permettait de regrouper 4 chiffres binaires en un chiffre hexadécimal a été une première solution pour simplifier cette tâche. Les programmes écrits en hexadécimal (figure 2) n'étaient toujours pas simples ni à écrire, ni à lire, mais il étaient moins longs.

Les langages d’assemblage qui permettaient d’associer des mnémoniques à des instructions binaires ont par la suite contribué à simplifier davantage cette tâche. Par exemple, l’instruction *MOV* permettait de *déplacer* le contenu d’une cellule mémoire, *CALL* d’appeler une sous routine, *JMP* d’effectuer un saut inconditionnel vers une autre partie du programme ... Les langages d’assemblage permettaient également de définir des objets de plus haut niveau tels que des variables, des étiquettes, ... facilitant l’écriture de programmes. Bien que permettant d’écrire plus facilement du code plus

75	69	20	6a	65	20	76	61	69	73	20	70	61	72	6c	65
72	20	64	65	73	20	3c	69	3e	62	69	74	73	3c	2f	69
3e	20	65	74	20	64	65	73	20	3c	69	3e	63	74	65	
74	73	2c	65	3e	2c	20	50	61	72	63	65	20	71	75	
65	20	63	65	20	73	6f	74	20	64	65	73	20	74	65	
72	6d	65	73	20	71	75	65	20	74	66	75	74	20	6c	
20	6d	6f	6e	64	65	20	72	65	6e	63	6f	74	72	65	
20	74	6f	75	73	20	6c	65	73	20	6a	6f	75	72	73	20
65	74	20	71	75	65	20	6c	61	20	70	6c	75	70	61	72
74	20	64	65	73	20	75	74	69	63	73	61	74	65	75	
72	73	20	64	27	6f	72	64	69	6e	71	64	75	72	73	
20	6e	65	20	73	61	76	65	6e	74	20	6d	65	6d	65	20
70	61	73	20	64	65	20	71	75	6f	69	20	69	6c	20	72
65	74	6f	75	72	6e	65	2c	3c	62	72	20	2f	3e	0d	0a
3c	62	72	20	2f	3e	0d	0a	43	6f	6d	65	6e	63	6f	
6e	73	20	70	61	72	20	6c	65	20	3c	61	20	68	72	
66	3d	22	68	74	74	70	3a	2f	66	72	2e	77	69	65	
69	70	65	64	69	61	2e	6f	72	67	2f	77	69	6b	69	2f
42	69	74	22	20	74	61	72	67	65	74	3d	22	5f	62	6c
61	6e	6b	22	3c	62	69	74	3c	2f	61	3e	2c	20	71	75
65	20	73	69	67	6e	69	66	69	65	20	22	42	69	61	6e
72	79	20	64	69	67	69	74	22	2c	20	73	6f	69	74	20
63	68	69	66	66	72	65	20	62	69	66	61	69	72	65	2e
20	41	75	74	22	65	6f	6e	74	20	64	69	74	2c	20	70

FIGURE 2 – Notation hexadécimale

lisible par l'humain (figure 3), écrire un programme à l'aide d'un langage d'assemblage demeurait une tâche fastidieuse. En effet, on ne pouvait pas définir de types complexes, il n'existe pas de structures de contrôle (*while*, *if*, *repeat* ...), le passage des arguments à une sous-routine se faisait en utilisant la pile ... Le programme effectuant un calcul spécifique était très dépendant de la machine sur laquelle il devait s'exécuter. Il n'était pas possible de faire abstraction de l'architecture physique de cette machine.

```

MOU    DX,0130
MOU    AX,0900
INT    21
MOU    DX,015E
MOU    AX,0A00
INT    21
MOU    DX,0150
MOU    AX,0900
INT    21
MOU    DX,0160
INT    21
INT    20
POP    SS

```

FIGURE 3 – Programme en assembleur

```

PROGRAM probleme
implicit none

real x
integer i
x=1.
print*, 'x=' ,x
call pas_bien(x)
end PROGRAM probleme

SUBROUTINE pas_bien(x)
implicit none
integer x
print*, 'x=' ,x
return
end SUBROUTINE pas_bien

```

FIGURE 4 – Programme en Fortran

Les choses ont commencée à changer au début des années 1950 avec les premiers langages de programmation évolués qu'étaient Lisp (1958), Fortran (1954) et Cobol (1959). Ceux-ci introduisaient des types, des fonctions ou procédures, des structures de contrôle qui rendaient l'écriture d'un programme plus naturelle. Plus précisément, ces langages permettaient d'exprimer des algorithmes à l'aide de mécanismes proches de ceux utilisés par le raisonnement humain. Ils permettaient de s'abstraire par rapport à l'architecture de la machine sur laquelle les programmes devaient s'exécuter et produisaient des programmes dont la compréhension était plus naturelle (figure 4).

À partir de ce moment, de nombreux langages ont été créés pour pallier les problèmes des précédents, abaisser la complexité qu'il y avait à résoudre des problèmes avec les langages existants et proposer plus d'abstraction par rapport à l'architecture physique de la machine. En d'autres termes, les langages étaient créés pour que nous puissions exprimer des algorithmes en utilisant un langage proche de ceux que nous utilisions dans notre vie de tous les jours (le langage naturel) et proposant des concepts proches de ceux que nous utilisons dans nos raisonnements. À titre d'exemple, le site <http://www.99-bottles-of-beer.net> propose l'écriture d'un même programme avec plus de 1400 langages de programmation différents.

Avec l'accroissement de la puissance de calcul des ordinateurs, les humains ont pu

s'attaquer à la résolution informatique de problèmes devenant de plus en plus ardu : la simulation de systèmes complexes (de trafic routier par exemple), l'intelligence artificielle, le stockage de données complexes dans des bases de données... Or, les mécanismes d'abstraction des langages existants s'adaptaient mal à la complexité de ces problèmes.

En 1967, à Oslo en Norvège, Ole-Johann Dahl et Kristen Nygaard définissaient le langage Simula 67 (*Simple Universal Language*) offrant un support pour la simulation à événements discrets. Simula proposait une implémentation informatique des types abstraits (regroupement de variables, de structures de données et de procédures travaillant sur ce type) : des *classes*. Pour cette raison, Simula est souvent considéré comme le père de tous les langages orientés objets. Simula offrait :

- la possibilité de créer de nouveaux types en juxtaposant des types existants,
- la modularité des programmes,
- la surcharge d'opérateurs (possibilité de définir des sémantiques différentes pour un même opérateur en fonction du type de ses arguments).



FIGURE 5 – Les inventeurs de SIMULA : Dahl et Nygaard

En 1972, au centre de recherche de Xerox, le Palo Alto Research Center (Xerox PARC), l'équipe de Alan Kay, Dan Ingals, Ted Kaehler et Adele Goldberg s'inspirait de Simula et de Lisp pour créer le premier langage orienté objet : Smalltalk-72. Il implémentait les notions de classe, d'objets et d'envoi de message. En 1976, l'équipe du PARC sort Smalltalk-76 qui ajoute le concept d'héritage à Smalltalk 1972. Quatre années plus tard, Smalltalk 80 ajoutait un environnement de programmation complet et la notion de métaclass à Smalltalk 1976.

Pour ses travaux sur Smalltalk-80, Alan Kay reçut le Turing Award en 2003. Les norvégiens Ole-Johann Dahl et Kristen Nygaard avaient, eux aussi reçu le Turing Award en 2001 pour leurs travaux sur SIMULA.

À partir des années 1980, on tentait d'intégrer les concepts objets ou des concepts proches dans les langages de programmation :

- le langage Ada proposait en 1983 un concept proche de celui de classe : le paquetage ;
- C++ en 1986 proposait d'ajouter les notions de classe et d'héritage au langage de programmation C ;

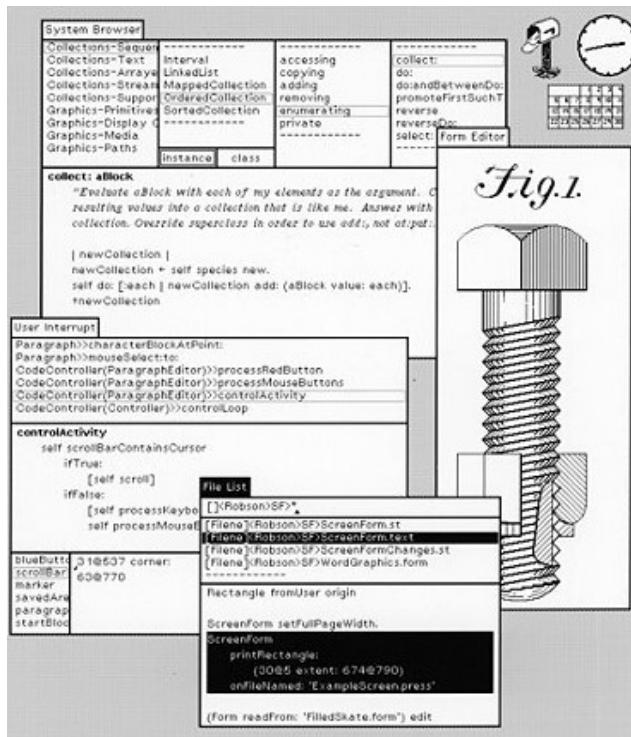


FIGURE 6 – Une interface graphique en Smalltalk

- en 1990, Bertrand Meyer du LORIA de l’Université de Nancy, proposait le langage Eiffel formalisait les notions de classe et d’héritage (notamment d’héritage multiple) dans un langage “propre” pour le génie logiciel.
- en 1995, Ada-95 voit le jour, il ajoute la notion d’héritage à Ada. Il est le premier langage orienté objets à avoir été normalisé.

C’est également en 1995, que James Gosling et Patrick Naughton créent le langage Java chez Sun Microsystems.

## Les 4 concepts fondamentaux de la programmation orientée objets

C’est depuis Smalltalk-80 qu’on considère que la programmation orientée objets s’appuie sur 4 concepts fondamentaux :

1. **la classe** : une classe représente une famille de “choses”. La classe des personnes représente la “famille” de toutes les personnes, la classe des rectangles celle de tous les rectangles.
2. **les instances et l’instanciation** : l’instanciation est l’opération qui permet de créer un “individu” (appelé instance ou objet) à partir de sa “famille” (une classe). Dans l’instanciation, on peut voir la classe comme le moule qui permet de créer les instances.
3. **l’envoi de message** : c’est le seul moyen de communiquer avec un objet. Pour

demander à un objet d'effectuer une action (d'appeler une fonction) on doit lui envoyer un message comprenant l'action qu'il doit effectuer et les éventuels arguments qu'on souhaite lui transmettre.

4. **L'héritage** : les classes peuvent être organisées hiérarchiquement selon des relations de spécialisation (ou de généralisation). Par exemple, on imagine bien qu'un "étudiant" est une "personne" dotée de caractéristiques spécifiques, c'est-à-dire que le concept d'"étudiant" est plus spécialisé que celui de "personne" (on peut aussi voir la "personne" comme étant un concept plus général que celui d'"étudiant"). L'héritage permet d'organiser les classes hiérarchiquement selon ce type de relation de spécialisation (ou de généralisation).

Naturellement, le langage Java, étant un langage orienté objets, il met en œuvre ces 4 concepts fondamentaux. Nous verrons dans la suite de ce document comment ils peuvent être utilisés en Java.

## En savoir plus sur l'histoire de la programmation

Si vous vous intéressez à l'histoire de la programmation vous trouverez des informations intéressantes sur les sites web suivants :

- Une histoire de l'informatique :  
<http://dept-info.labri.fr/~dicky/HistInfo.html>
- Une histoire des langages de programmation :  
[https://www.youtube.com/watch?v=OiqL9W9\\_Hlw](https://www.youtube.com/watch?v=OiqL9W9_Hlw)

# Chapitre 1

## Le langage Java

### 1.1 Les origines de Java ?

Avant sa sortie des laboratoires de Sun Microsystems le projet s'appelait Oak, mais le nom avait déjà été déposé par Oak Technology. Il était donc nécessaire de trouver un nom pour ce nouveau langage. Après plusieurs séances de brainstorming, le nom de *Java* s'est imposé ; peut-être en hommage à l'île de Java dont venait le café qui avait été bu par ses concepteurs lors de son développement. Cette légende expliquerait aussi le choix du logo de Java (Figure 1.1).



FIGURE 1.1 – Une bonne tasse de Java bien chaud

Java était initialement un langage de programmation dédié aux appareils électroménagers. Pour cela, un même programme devait pouvoir s'exécuter sur tout type de puce : depuis celle d'un réfrigérateur, jusqu'à celle d'une cafetière électrique. Cette portabilité du langage Java en a tout naturellement fait le langage idéal pour le World Wide Web. En effet, étant multi plate-forme, un seul et même programme Java peut s'exécuter sur des ordinateurs possédant des architectures et des systèmes d'exploitation différents et donc potentiellement sur toute machine de l'Internet.

Le langage Java, syntaxiquement proche de C++ (et donc du langage C) sans toutefois avoir les inconvénients et la complexité de ce dernier était conçu pour être multi plate-forme, sécurisé en terme de programmation et de réseau et multi-tâche. Plus précisément, le livre blanc publié par Sun Microsystems en 1995 [3], annonçait un langage simple, orienté objets, réparti, interprété, robuste, sûr, indépendant de l'architecture, portable, efficace, multi-thread et dynamique.

## 1.2 Un langage simple

Pour faciliter l'apprentissage du langage Java, les concepteurs ont choisi de s'appuyer sur un langage de programmation bien connu de tous, le langage C++, pour définir la syntaxe de Java. Dans un but de simplification, les concepts de C et de C++ les moins faciles à comprendre n'ont pas été intégrés à Java. Ainsi :

- Java n'a **pas de préprocesseur** (pas de `#include` ou de `#define` comme en C) : le code transmis au compilateur est très exactement celui qui a été écrit par le programmeur. Aucun mécanisme de réécriture ne s'applique sur le code source original avant la compilation ;
- Java ne propose **pas de type union ou de structure** : le seul type de données qui peut être défini en Java est la classe. D'un point de vue structurel, une classe ressemble à une structure dans le sens où elle comprend des champs nommés. Une classe permet également de définir les fonctions (des méthodes) qui s'appliquent sur le type de données défini par la classe ;
- Java ne propose **pas de pointeurs** : le langage fournit des références sur certaines données. Les références ressemblent aux pointeurs sauf qu'elles ne sont pas modifiables et interdisent notamment l'arithmétique de pointeurs ;
- dans un programme Java il **n'est pas nécessaire de gérer explicitement la mémoire** (malloc, free) : la gestion de la mémoire se fait automatiquement par le langage. Les allocations mémoire se font implicitement et la libération de la mémoire est faite par un programme spécialisé "le ramasse miettes" (garbage collector) chargé de scruter la mémoire pour y détecter et libérer la mémoire allouée qui n'est plus utilisée ;
- Java ne propose **pas d'instruction goto** : l'instruction `goto` est présente dans de nombreux langages mais, depuis [1], tout programmeur sait qu'il ne doit pas utiliser cette instruction. Pour totalement interdire aux programmeurs d'utiliser cette instruction de saut inconditionnelle elle ne figure pas dans le jeu d'instructions de Java ;
- Java **n'autorise pas la surcharge d'opérateur** : certains opérateurs de Java sont surchargés. C'est le cas par exemple de l'opérateur '+' qui calcule la somme de deux nombres et concatène deux chaînes de caractères. Cependant, le langage n'offre pas la possibilité au programmeur de définir de tels opérateurs ;
- Java **ne permet pas l'héritage multiple** : les concepteurs de Java ont considéré que l'héritage multiple apportait parfois plus de problèmes que de solutions. Ils ont ainsi décidé de ne pas proposer l'héritage multiple dans Java et ont préféré proposer un moyen (les *interfaces*) permettant de s'en affranchir.

### 1.2.1 Un ensemble de types prédéfinis

Java propose deux catégories de types : les types primitifs et les types référence.

**Types primitifs :** Java est doté des mêmes types primitifs prédéfinis que le langage C. Ainsi, on retrouve les types suivants :

- `char` : caractères Unicode sur 16 bits ;
- `short` : entiers signés sur 16 bits ;
- `int` : entiers signés sur 32 bits ;
- `long` : entiers signés sur 64 bits ;
- `float` : nombres flottants sur 32 bits ;
- `double` : nombres flottants sur 64 bits .

Type	Taille	Valeurs
<i>boolean</i>	1 bit	<i>true</i> et <i>false</i>
<i>byte</i>	8 bits	de -128 à 127
<i>short</i>	16 bits	de -32768 à 32767
<i>int</i>	32 bits	de -2147483648 à 2147483647
<i>long</i>	64 bits	de -9223372036854775808 à 9223372036854775807
<i>float</i>	32 bits	de $-3.40282346638528860 \times 10^{38}$ à $-1.40129846432481707 \times 10^{-45}$ et de $1.40129846432481707 \times 10^{-45}$ à $3.40282346638528860 \times 10^{38}$
<i>double</i>	64 bits	de $-1.79769313486231570 \times 10^{308}$ à $-4.94065645841246544 \times 10^{-324}$ et de $4.94065645841246544 \times 10^{-324}$ à $1.79769313486231570 \times 10^{308}$
<i>char</i>	16 bits	caractères unicode de \u0000 à \uffff

TABLE 1.1 – types, tailles et valeurs

Deux types ont été ajoutés à cette liste pour représenter les données booléennes et les octets :

- *boolean* : booléens sur 1 bit;
- *byte* : octets sur ... 8 bits (bien sûr).

Contrairement à d'autres langages dans lesquels la taille d'un type peut varier en fonction de la cible, les tailles des types de Java sont définies de façon stricte dans la spécification [2] et sont toujours les mêmes. Ce point est nécessaire pour assurer la compatibilité du langage d'une plate-forme à l'autre.

Toute variable d'un type prédéfini est toujours manipulée par valeur. Le tableau 1.1 résume les différents types prédéfinis de Java, leurs tailles et les valeurs extrêmes qu'ils peuvent prendre.

**Les types référence :** d'autres types existent en Java, notamment les *classes* et les *tableaux*. Ces derniers sont appelés *types référence* et sont, comme leurs noms l'indiquent, toujours manipulés par référence.

**À retenir**

*Il n'est pas nécessaire, en Java, lorsqu'on définit une fonction, de spécifier le mode de passage des paramètres ; il dépendra du type de chaque paramètre.*

## 1.2.2 Des structures de contrôle

Java propose un ensemble de structures de contrôle identique à celui du langage C.

Les mêmes règles qu'en C s'appliquent concernant les accolades de début et de fin de bloc. Cependant, pour éviter des surprises à l'exécution, il est recommandé et considéré comme une bonne pratique de programmation Java, pour éviter toute déconvenue, de faire figurer les accolades systématiquement. Que ce soit autour d'un bloc d'instructions ou d'une unique instruction.

**La structure conditionnelle** Java propose une structure conditionnelle semblable en tout point à celle du langage C :

```
if (expressionBooléenne){
    alternantSiVrai1;
    alternantSiVrai2;
```

```

...
alternantSiVraiN;
} else {
alternantSiFaux1;
alternantSiFaux2;
...
alternantSiFauxM;
}

```

Si le résultat de `expressionBooléenne` est égal à `true` les instructions `alternantSiVrai1` à `alternantSiVraiN` sont exécutées. Si en revanche le résultat de `expressionBooléenne` est égal à `false` ce sont les instructions `alternantSiFaux1` à `alternantSiFauxM` qui sont exécutées.

**La boucle “tant que” (while)** La syntaxe de la boucle “tant que” en Java est identique à celle du langage C :

```

while (conditionItération){
    instruction1;
    instruction2;
    ...
    instructionN;
}

```

Tant que le résultat de `conditionItération` est égal à `true` les instructions `instruction1` à `instructionN` sont exécutées. La condition d’itération est évaluée avant chaque passage dans la boucle. Ainsi, si la condition est initialement fausse aucune des instructions du corps de boucle ne sera exécutée.

**La boucle “faire tant que” (do while)** La syntaxe de la boucle “faire tant que” en Java est identique à celle du langage C :

```

do {
    instruction1;
    instruction2;
    ...
    instructionN;
} while (conditionItération);

```

Les instructions `instruction1` à `instructionN` sont exécutées tant que le résultat de `conditionItération` est égal à `true`. La condition d’itération n’est évaluée qu’après le passage dans la boucle. Ainsi, les instructions du corps de boucle seront exécutées une fois si le résultat de `conditionItération` est initialement faux.

**La boucle “pour” (for)**

**La version classique** La syntaxe de la boucle “pour” en Java est identique à celle du langage C :

```

for (initialisation;
    conditionItération;
    incrémentation) {

```

```

instruction1;
instruction2;
...
instructionN;
}

```

Avant d'entrer dans la boucle pour la première fois, les instructions d'`initialisation` sont exécutées. Avant de débuter l'exécution du corps de boucle la `conditionItération` est vérifiée puis, si c'est le cas, les instructions du corps de boucle sont exécutées. Après chaque exécution du corps de boucle, les instructions d'`incrémentation` sont exécutées. Chacune des parties : `initialisation`, `conditionItération`, `incrémentation` peuvent être constituées de plusieurs instructions séparées par des virgules “;”.

**À retenir** Java autorise la déclaration des variables de boucle au moment de leurs initialisations.

```

for (int i=0;i<n;i++){
    ...
}

```

**La boucle for-each** Il existe en Java une autre version de la boucle “`for`” appelée la boucle “`for-each`” qui facilite le parcours d'une collection d'éléments (une liste, un tableau ou toute autre structure agrégeant un ensemble d'éléments). Cette version de la boucle “`for`” permet de parcourir la collection sans devoir accéder via leurs indices aux différents éléments qu'elle contient. On pourrait par exemple, procéder comme suit pour appliquer un traitement à tous les éléments d'une collection :

```

/*
 * Soit collectionElement une collection
 * (tableau, liste ...) d'éléments de type
 * TypeElement.
 */
for (TypeElement e : collectionElement){
    /* Pour chaque element e */
    traiter(e)
}

```

Par exemple, si `tab` est un tableau d'entiers, on pourra égrainer les différentes valeur du tableau comme suit :

```

/* Pour chaque entier x dans tab */
for (int x: tab){
    ...
}

```

**L'instruction de “branchement” (switch)** La syntaxe de l'instruction de branchement en Java est identique à celle du langage C :

```

switch (expression){
    case valeur1 : instruction1;
}

```

```

        instruction1_2;
        ...
        instruction1N1;
        break;
    case valeur2 : instruction21;
        instruction22;
        ...
        instruction2N2;
        break;
        ...
    default : instructionM1;
        instructionM2;
        ...
        instructionMNM;
}

```

Les instructions suivant le bloc `case` (correspondant au résultat de l'évaluation) sont exécutées jusqu'à ce qu'une instruction `break` soit rencontrée (les blocs `case` sont envisagés dans leur ordre d'apparition). Si aucun cas ne correspond au résultat de l'expression, les instructions qui suivent le bloc `default` sont exécutées.

Le type du résultat de l'évaluation de l'expression ne peut être qu'un caractère, un nombre entier, une variable appartenant à un type énuméré ou (depuis la version 7 du langage) une chaîne de caractères.

### 1.2.2.1 Commentaires

Le langage Java propose deux façons de commenter le code source : des commentaires sur une ligne et d'autres permettant d'écrire un commentaire sur plusieurs lignes.

Pour écrire un commentaire sur une seule ligne il suffit, comme nous l'avons vu dans les exemples de code ci-dessus, de le préfixer avec les caractères “`//`” de la façon suivante :

```
x=1+2; // ce commentaire se termine en fin de ligne
```

Si on souhaite être plus prolifique dans les commentaires, il est préférable d'utiliser un autre style de commentaire permettant d'écrire plusieurs lignes. Il suffit pour cela de commencer le bloc de commentaires par les caractères “`/*`” et de le terminer par les caractères “`*/`”, de la façon suivante :

```

/*
 * Ceci est la première ligne d'un commentaire qui
 * peut s'étendre sur plusieurs lignes.
 * Bien que les "*" en début de chaque ligne
 * ne soient pas nécessaires il est d'usage de
 * les utiliser afin de mieux identifier
 * les commentaires multi-lignes.
 * Les commentaires multi-lignes se terminent par le
 * caractère "*" suivi de "/".
 */
x=1+2;

```

La section B.8.5 de l'annexe B présente le formatage à utiliser pour les différents styles de commentaires.

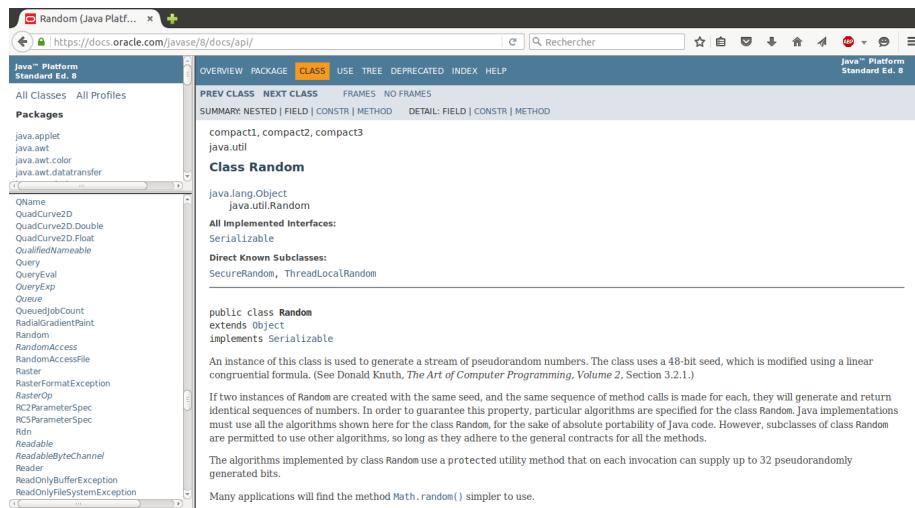


FIGURE 1.2 – La documentation complète de l’API

Les commentaires, lorsqu’ils sont écrits selon un certain format, permettent à l’outil *javadoc*, livré avec le kit de développement Java, de générer de la documentation sous la forme de fichiers HTML. Toutes les classes de l’API de Java sont documentées de la sorte. Ainsi, Oracle propose une documentation officielle en ligne de toutes les classes de l’API sur le site d’Oracle<sup>1</sup>. Elle est présentée telle que représenté par l’image de la figure 1.2. Une brève présentation du format des commentaires de documentation figure en annexe C.

### 1.2.2.2 Formatage du code Java

Il existe de nombreuses façons de formater ou de présenter le code source Java. Les règles de formatage que nous utiliserons dans le cadre de ce cours sont celles préconisées par Google. Une description précise et complète du *style Google* de programmation Java se trouve sur un dépôt `\emph{github}`<sup>2</sup>. Le chapitre traitant du formatage du code se trouve également dans ce document en Annexe B.

### 1.2.3 Un langage orienté objets

Un programme Java est un ensemble de classes organisées hiérarchiquement. Le kit de développement de Java (le JDK : Java Development Kit) est livré avec un ensemble de classes pouvant être utilisées dans tout programme. Ces classes, proposant des implémentations de concepts pouvant être utiles à tout programmeur, constituent l’API (*Application Programming Interface*) de Java. Les classes de l’API sont réparties dans des paquetages organisés hiérarchiquement selon les thématiques qu’elles abordent.

L’arbre d’héritage des classes de Java a une unique racine : la classe *Object*.

1. <https://docs.oracle.com/javase/8/docs/api/>
2. <http://google.github.io/styleguide/javaguide.html>

Version	Année	Nb de classes	Nb de paquetages
1.0	1996	212	8
1.1	1997	504	23
Java 2 (SDK 1.2)	1998	1520	59
Java 2 (SDK 1.3)	2000	1842	76
Java 2 (SDK 1.4.0)	2002	2991	135
Java 2 SE 5.0 (JDK 1.5.0)	2004	3279	166
Java SE 6 (JDK 1.6.0)	2006	3793	203
Java SE 7 (JDK 1.7.0)	2011	4024	209
Java SE 8 (JDK 1.8.0)	2014	4240	217
Java SE 9 (JDK 1.9.0)	2017	6005	315
Java SE 10 (JDK 1.10.0)	2018	6002	314
Java SE 11 (JDK 1.11.0)	2018	4413	223

TABLE 1.2 – Évolution du contenu des versions de Java.

Le langage Java permet de mettre en œuvre tous les concepts fondamentaux de la programmation orientée objets que sont : les classes, l’instanciation, l’héritage et l’envoi de messages.

#### 1.2.4 Un langage interprété et portable

Deux outils sont absolument indispensables pour écrire et exécuter un programme Java : un compilateur Java et une machine virtuelle Java.

Un programme Java doit être écrit dans un fichier d’extension “.java”.

Le compilateur traduit le code source Java en un code intermédiaire : le *byte-code*. Le *byte-code* Java est un langage de bas niveau conçu pour être facilement interprété par une architecture matérielle quelconque. Les fichiers contenant du *byte-code* ont pour extension : “.class”.

Une *machine virtuelle Java* (*Java virtual Machine, JVM*) est chargée d’interpréter des programmes en byte-code sur une architecture matérielle donnée. C’est en définitive, la machine virtuelle qui se charge de l’exécution du code produit par le compilateur en l’interprétant. Une *machine virtuelle Java* est en fait un interpréteur de *byte-code* pour une plate-forme donnée. C’est elle qui assure la portabilité du code Java et qui garantit qu’une application Java compilée (un fichier contenant du *byte-code*) peut s’exécuter sur toute plate-forme, pourvu qu’il existe une machine virtuelle pour cette plate-forme. En pratique, il est préférable de tester un programme Java sur plusieurs plate-forme, si on souhaite assurer sa portabilité. C’est particulièrement le cas pour la portabilité des interface personne/machine.

Plusieurs implémentations du compilateur et de la machine virtuelle Java sont disponibles sur internet. Une de ces implémentations, le Java SE Development Kit d’Oracle , dont il existe des versions pour Linux, Mac OS et Windows, est téléchargeable sur le site web d’Oracle (un moteur de recherche quelconque vous aidera à le trouver). D’autres implémentations existent, par exemple : *openjdk*, *jikes* ... En règle générale, ces autres implémentations n’intègrent pas toutes les fonctionnalités de la dernière version proposée par Oracle.

Dans ce document nous utiliserons la version 8 de l’implémentation d’Oracle : *Java SE Development Kit 8*. Il contient tous les outils nécessaires pour programmer en Java.

Pour l'instant ce qui nous intéresse dans le contenu du JDK sont les outils suivants : le compilateur, la machine virtuelle et les classes de l'API.

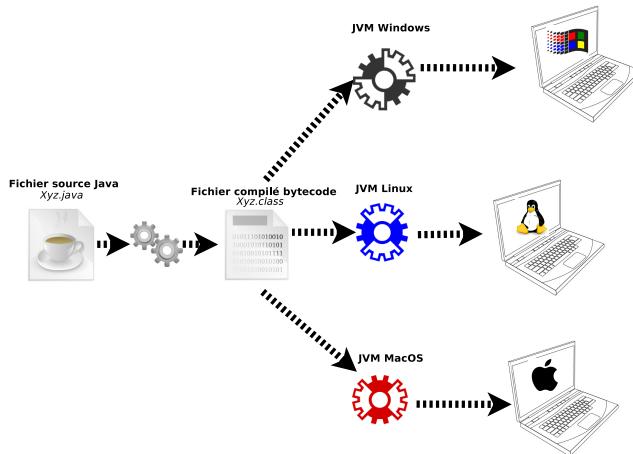


FIGURE 1.3 – Write once, run anywhere

## 1.3 Premiers programmes en Java

Nous développerons nos premiers programmes Java de manière “artisanale” : le code source sera écrit grâce à un éditeur de texte standard (par exemple *gedit* ou *kate*, *vi*, *emacs* sous Linux, *notepad* sous Windows et *TextWrangler*, *TextEdit* sous Mac OS), la compilation et l'exécution se feront depuis un terminal grâce au compilateur *javac* et à la machine virtuelle *java*. Les exemples donnés dans ce document sont tous écrits sous Linux.

### 1.3.1 Une classe pour saluer l'utilisateur

Comme dans tout langage de programmation notre premier programme (on verra plus tard que c'est une classe) s'appellera *Hello* et se contentera de saluer son utilisateur.

1. Dans le fichier *Hello.java*, écrire le code suivant :

---

#### Exemple 1.1 Définition de la classe Hello

---

```

class Hello {
    public static void main (String[] args){
        System.out.println("Bonjour cher utilisateur");
    }
}
  
```

---

Ce code correspond à la définition d'une classe de nom *Hello*, elle doit nécessairement se trouver dans le fichier *Hello.java*.



```

1 class Hello {
2     public static void main (String[] args){
3         System.out.println("Bonjour cher utilisateur");
4     }
5 }
```

FIGURE 1.4 – Écriture de la classe *Hello*

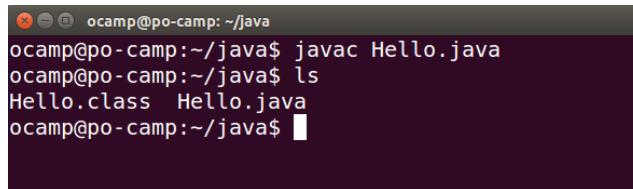
1. Dans un terminal, en utilisant la commande *cd*, déplacez vous dans le répertoire dans lequel vous avez rangé le fichier *Hello.java* pour le compiler en tapant la ligne de commande suivante :

```
javac Hello.java
```

La compilation du fichier *Hello.java* créera le fichier *Hello.class* contenant le byte-code de la classe *Hello*.

Pour que le fichier *Hello.java* puisse se compiler sans erreur il faut que son contenu respecte scrupuleusement la syntaxe du langage Java : qu'il soit syntaxiquement correct. Si le code contient des erreurs de syntaxe, la compilation est un échec et le compilateur indique les erreurs qu'il a rencontrées.

**Attention :** Si un fichier Java se compile sans erreur c'est qu'il respecte la syntaxe du langage. Ça ne veut pas dire qu'il va fonctionner correctement.



```

ocamp@po-camp:~/java$ javac Hello.java
ocamp@po-camp:~/java$ ls
Hello.class  Hello.java
ocamp@po-camp:~/java$
```

FIGURE 1.5 – Compilation du fichier *Hello.java*

2. Dans le terminal, soumettez la classe compilée *Hello* à la machine virtuelle afin de l'exécuter :

```
java Hello
```

```
ocamp@po-camp:~/Java
ocamp@po-camp:~/java$ java Hello
Bonjour cher utilisateur
ocamp@po-camp:~/java$
```

FIGURE 1.6 – Exécution de la classe *Hello***À retenir :**

1. *La compilation s’applique à un fichier contenant du code source Java. Ainsi, l’argument passé à la commande javac est un nom de fichier et doit donc comporter l’extension “.java”.*  
*On peut compiler plusieurs fichiers en même temps en passant la liste de leurs noms en arguments au compilateur :*

```
javac Fichier1.java Fichier2.java Fichier3.java
```

2. *La machine virtuelle est chargée de l’exécution d’une classe. Ainsi, l’argument passé à la commande java est un nom de classe et ne doit donc pas comporter d’extension.*

**1.3.1.1 Quelques explications sur la définition de la classe *Hello***

Un programme Java est constitué de plusieurs classes, chacune définie dans un fichier portant le même nom que la classe et portant l’extension “.java”. Ainsi, la classe *Hello* détaillée dans l’Exemple 1.1 doit être écrite dans le fichier *Hello.java*.

**Définition d’une classe** La définition d’une classe se fait à l’aide de la construction suivante :

```
class <NomDeClasse> {
    // Définitions de la structure et du comportement
    // de la classe
    // ....
}
```

C’est à l’intérieur des accolades que seront définies la structure et le comportement des objets de la classe à l’aide de “variables” et de “fonctions” qu’on appelle *attributs* et *méthodes* en programmation orientée objets.

**Définition d’une classe exécutable** Pour qu’elle soit exécutable par la machine virtuelle Java, une classe doit nécessairement définir une méthode (les “fonctions” de la programmation orientée objets) principale dont la signature doit être exactement :

```
public static void main (String [] args)
```

- Le mot-clé *public* indique que la méthode est accessible et utilisable partout dans le programme ;

- Dans un premier temps on fera abstraction de la signification du mot-clef *static* et on se contraindra à le faire figurer devant la définition de la méthode principale ;
- La méthode principale ne retourne pas de résultat son type de retour est donc *void* ;
- Le nom de la méthode principale doit nécessairement être *main* (c'est également le nom de la fonction principale d'un programme en C) ;
- Le paramètre de la fonction principale doit être déclaré comme étant un tableau de chaîne de caractères (la classe *String* de l'API de Java permet de représenter les chaînes de caractères) qu'on nomme *args* par convention. Ce paramètre prendra pour valeur un tableau contenant tous les arguments qui auront été passés au programme sur la ligne de commande. Par exemple, le code suivant permet de passer 3 arguments à la classe *Hello* :

```
java Hello argument1 argument2 arguments3
```

Les trois arguments (des chaînes de caractères de la classe *java.util.String*) seront transmis à la méthode *main* de la classe *Hello* dans les trois premières cellules du tableau *args*(le paramètre de la méthode *main*).

### 1.3.1.2 Les flux (ou canaux) d'entrée/sortie

Lorsqu'il s'exécute, un programme Java dispose de trois canaux lui permettant de communiquer avec l'utilisateur. Ces canaux sont appelés des *flux* : deux d'entre eux sont des *flux de sortie* qui permettent au programme d'afficher des messages à l'utilisateur. Par défaut les messages s'affichent sur l'écran. Le troisième est un *flux d'entrée* et permet à l'utilisateur de donner des informations au programme. Par défaut le flux d'entrée récupère les caractères tapés par l'utilisateur au clavier. Ces trois moyens de communiquer avec un programme Java sont empruntés à Unix qui, lui aussi, associe à tout processus ces trois mêmes flux :

- le flux de sortie standard appelé *stdout* par Unix : ce flux est utilisé par le processus lors de son fonctionnement pour afficher des messages à l'utilisateur,
- le flux d'erreur standard appelé *stderr* par Unix : ce flux est utilisé par le processus pour signaler des dysfonctionnements à l'utilisateur,
- le flux d'entrée standard appelé *stdin* par Unix : ce flux sert à l'utilisateur pour transmettre des informations au processus.

En Java les trois flux standard se nomment (voir Figure 1.7) :

- *System.out* : sortie standard,
- *System.err* : erreur standard,
- *System.in* : entrée standard.

Ainsi, la formule magique *System.out.println ("message")* a pour effet d'afficher la chaîne de caractères “*message*” en utilisant *println ("message")*, sur la sortie standard *System.out*. De même, *System.err.println ("message")* correspondra également à une demande d'affichage d'un message, mais sur la sortie erreur standard *System.err*.

Nous aborderons ultérieurement la saisie par l'utilisateur lorsque nous aborderons les flux plus en détail.

**Redirection des flux d'entrée/sortie** Plutôt que d'afficher des messages à l'écran on peut vouloir les écrire dans des fichiers. Par ailleurs, au lieu de lire ses entrées au

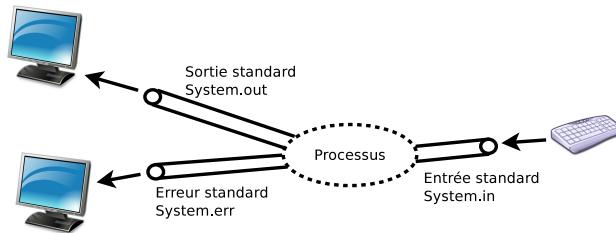


FIGURE 1.7 – Flux d'entrée/sortie standard

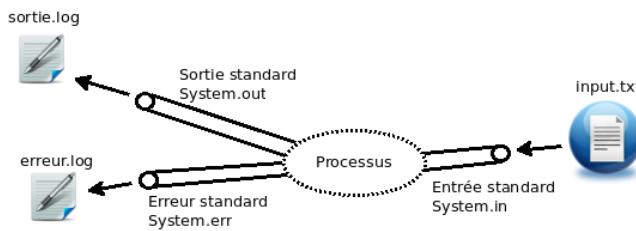


FIGURE 1.8 – Flux d'entrée/sortie redirigés

clavier on peut vouloir les lire dans un fichier. Pour obtenir ces résultats chacun des flux standard peut être redirigé vers un fichier.

Ainsi, par exemple, on pourrait vouloir écrire tous les messages affichés à l'écran par une classe Exemple dans le fichier `sortie.log`, afficher les erreurs standard dans le fichier `erreur.log` et lire les données dans le fichier `input.txt`. Pour cela on doit rediriger la sortie standard vers le fichier `sortie.log`, la sortie erreur standard vers le fichier `erreur.log` et l'entrée standard vers le fichier `input.txt`. Avec l'interpréteur de commande *bash* sous Unix ou depuis une “invite de commande” sous Windows, ce résultat (représenté dans la Figure 1.8) s'obtient en tapant la ligne de commande suivante :

```
java Exemple >sortie.log 2>erreur.log <input.txt
```

- Le caractère '`>`' permet d'indiquer le fichier vers lequel on souhaite rediriger la sortie standard. Les instructions Java `System.out.println` auront alors pour effet d'écrire du texte dans le fichier dont le nom suit le caractère '`>`' sur la ligne de commande. Dans les cas où le fichier vers lequel on souhaite rediriger la sortie existe déjà, il sera effacé. Si on souhaite conserver le fichier et rajouter les nouveaux messages à la fin, on utilisera les caractères '`>>`' pour la redirection.
- La séquence de caractères '`2>`' permet d'indiquer le fichier vers lequel on souhaite rediriger la sortie erreur standard. Les instructions Java `System.err.println` auront alors pour effet d'écrire du texte dans le fichier dont le nom suit la séquence '`2>`' (on utilisera '`2>>`' si on souhaite ajouter à la fin du fichier plutôt que de l'effacer),
- Le caractère '`<`' permet d'indiquer le fichier vers lequel on souhaite rediriger l'entrée standard. Les instructions Java qui attendent que l'utilisateur saisisse quelque chose au clavier iront le lire dans le fichier spécifié après le caractère '`<`'.

**Attention :**

1. Les indications de redirection doivent figurer sur la ligne de commande après les éventuels arguments passés à la classe Java.
2. Les indications de redirection ne sont pas considérées comme des arguments par la machine virtuelle Java.

**1.3.2 Manipulation des arguments passés sur la ligne de commande**

Considérons la classe *Bonjour* définie dans l'exemple 1.2

**Exemple 1.2** Définition de la classe *Bonjour*

```

1 class Bonjour {
2     public static void main (String [] args){
3         System.out.println("Bonjour cher utilisateur");
4         System.out.println("Voici les arguments passés"
5                             + "en ligne de commande :");
6         for (int i=0;i<args.length;i++){
7             System.out.println("Argument Numéro "
8                             + (i+1) + " : " + args[i]);
9         }
10    }
11 }
```

Avant toute chose, le fichier *Bonjour.java* doit être compilé grâce à la ligne de commande :

**Exemple 1.3** Compilation d'une classe

```
javac Bonjour.java
```

Une fois créée par le compilateur, la classe *Bonjour* peut être exécutée grâce à la ligne de commande suivante (dans l'exemple suivant les 3 arguments *pomme*, *abeille* et *soleil* sont passés à la classe *Bonjour*) :

**Exemple 1.4** Exécution d'une classe

```
java Bonjour pomme abeille soleil
```

Elle provoquera l'affichage des messages suivants :

```

Bonjour cher utilisateur
Voici les arguments passés en ligne de commande :
Argument Numéro 1 : pomme
Argument Numéro 2 : abeille
Argument Numéro 3 : soleil
```

**À retenir :** L'opérateur '+' peut s'appliquer à des chaînes de caractères. Dans ce cas c'est un opérateur de concaténation de chaînes de caractères. Par exemple :

```
"bonjour" + "cher" + "utilisateur"
= "bonjourcherutilisateur"
```

### 1.3.3 Quelques exercices

Les exercices suivants ont pour objectifs :

1. d'organiser l'espace disque dans lequel sera stocké votre travail en Java,
2. de vous faire découvrir et manipuler la syntaxe de Java,
3. de vous faire écrire vos premières classes Java, de les compiler et de les exécuter.

**Exercice 1 :** Cet exercice a pour objectif de vous faire créer les répertoires nécessaires au stockage de vos travaux en Java.

1. Créer, si ce n'est pas encore fait, le répertoire "PDLO" dans lequel seront stockés tous les travaux faits dans le cadre du PDLO.
2. Créer le répertoire "java", sous répertoire du dossier "PDLO" .
3. Créer deux sous répertoires du répertoire "java" : "exercices" et "projet"

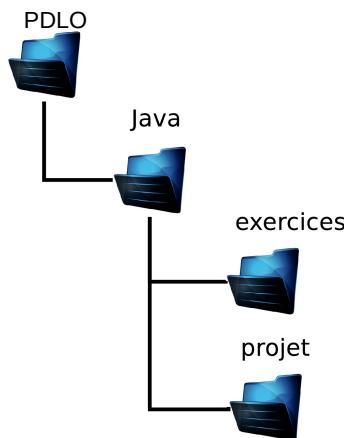


FIGURE 1.9 – Structure de répertoires de l'espace disque du PDLO

**Exercice 2 :** Écrire la classe *Hello* de l'exemple 1.1 dans le fichier *Hello.java*. Compiler le fichier et exécuter la classe obtenue.

**Exercice 3 :** Écrire la classe *Bonjour* de l'exemple 1.2 dans le fichier *Bonjour.java*.

1. Compiler le fichier et exécuter la classe obtenue. On prendra bien soin d'observer le comportement de la classe lorsqu'elle est exécutée sans et avec un ou plusieurs arguments.

2. Modifier la définition de la classe *Bonjour* afin qu'elle :

- (a) identifie les cas où elle est appelée sans argument et qu'elle le signale à l'utilisateur. On aura ainsi pour l'appel sans argument suivant :

```
Bonjour cher utilisateur  
Aucun argument n'a été passé en ligne de  
commande
```

- (b) traite de manière particulière le cas d'un appel avec un seul argument et qu'elle affiche un message grammaticalement correct à l'utilisateur. On aura ainsi pour l'appel suivant avec un seul argument :

```
java Bonjour unSeulArgument
```

l'affichage des messages :

```
Bonjour cher utilisateur  
Voici le seul argument passé en ligne de  
commande :  
Argument : unSeulArgument
```

- (c) ne change quasiment pas son comportement lorsqu'elle est appelée avec plusieurs arguments (la seule différence par rapport au code de l'exemple 1.2 est l'affichage du nombre précis d'arguments passés). Ainsi, un appel avec plusieurs arguments tel que :

```
java Bonjour pomme abeille soleil
```

provoque l'affichage des messages suivant :

```
Bonjour cher utilisateur  
Voici les 3 arguments passés en ligne de  
commande :  
Argument Numéro 0 : pomme  
Argument Numéro 1 : abeille  
Argument Numéro 2 : soleil
```

## 1.4 Classes, Objets et Instanciation

Les classes permettent de regrouper données et comportement au sein d'une même entité. Comme ça a été évoqué plus tôt, une classe peut être vue comme une "famille" de choses semblables, dont les "membres" sont des objets de la classes ou instances.

Les instances d'une classe, sont dotées de leurs propres attributs et savent déclencher l'exécution des méthodes de leur classe lorsqu'ils sont sollicités par envoi de message.

### 1.4.1 Définition d'une classe

Les données de la classe sont représentées par ses attributs (variables d'instance, variables de classe, variables membres). Son comportement est représenté par l'ensemble

FIGURE 1.10 – Diagramme de classe UML de la classe *Personne*

de ses méthodes.

**Important :** Le nommage des différents éléments de Java doit respecter certaines règles très simples qui permettent de rendre le programme facilement compréhensible par tout programmeur. Ces règles sont décrites dans l'annexe A et il est **impératif** de les respecter.

Dans ce qui suit nous considérerons une classe chargée de représenter des personnes. Celle-ci considérera qu'une personne est représentée par son identité constituée de ses *nom* et *prénom*, de son *année de naissance* et de sa *nationalité*. D'autre part, on fera en sorte que la classe garde trace du *nombre de personnes* qui ont été créées. La classe *Personne* peut être représentée par le diagramme de classe UML de la figure 1.10.

#### 1.4.1.1 Variables d'instance et variables de classe

Les variables définies par une classe peuvent être de deux types :

- des variables et méthodes portant sur (ou s'appliquant aux) instances de la classe : des variables et méthodes d'*instance*
- des variables et méthodes portant sur (ou s'appliquant à) la classe elle-même : des variables et méthodes de *classe*.

Pour la classe *Personne* on comprendra aisément que le nom, le prénom et l'année de naissance doivent être des informations propres à chaque personne. En effet, Mohamed Ali né en 1942 et de nationalité américaine, n'a pas le même nom, prénom, année de naissance et nationalité que Marcel Cerdan qui, lui, est né en 1916 et est de nationalité française. Si des objets devaient représenter ces deux personnes, les variables qui représentent leurs noms, prénoms, années de naissance et nationalité devraient donc pouvoir prendre une valeur différente pour chacune des deux personnes : ces variables seraient des **variables d'instance**.

En revanche, le nombre total de personnes qui ont été créées n'est pas une information propre à chaque personne. C'est une information générale et qui doit être commune à toutes les personnes : cette information serait alors une **variable de classe** ; sa valeur serait unique et stockée dans la classe.

Le mot-clé *static* de Java permet de différencier ces deux types de variables et il devra précéder la déclaration d'une variable de classe comme on le verra dans l'exemple 1.1.

Certaines *variables* de classe peuvent avoir des valeurs constantes qui ne changeront pas pendant toute la durée de vie de la classe. On les appelle des **constantes de classe**. Le mot-clef *final* permet d'indiquer leur caractère immuable de la variable (qui devient ainsi une constante). La déclaration d'une constante de classe sera donc précédée du mot-clef *static* indiquant que c'est une *variable* de classe suivie du mot-clef *final* indiquant que sa valeur ne changera pas et que c'est donc une **constante de classe**. La valeur d'une constante de classe ne pouvant pas être modifiée pendant la durée de vie de la classe, elle doit donc être initialisée lors de sa déclaration. Par exemple, si nous souhaitons définir que la nationalité par défaut d'une personne est la nationalité française, il est nécessaire de déclarer la constante de classe *NATIONALITE\_PAR\_DEFAUT* et de lui donner pour valeur la chaîne de caractères "française". Cette déclaration se fait grâce à la déclaration suivante :

```
static final String NATIONALITE_PAR_DEFAUT="française";
```

Le fait que ce nom comporte uniquement des majuscules permet d'indiquer que c'est une constante (voir l'annexe A).

**Le mot-clef *final*** Le mot-clef *final* peut précéder la déclaration de plusieurs types d'entités Java : les variables d'instance ou de classe, les méthodes, les classes. Il permet d'indiquer le caractère immuable de l'entité qui le suit ; ce qui signifie :

- pour une variable d'instance ou de classe, que la valeur avec laquelle elle a été (nécessairement) initialisée ne peut pas être modifiée,
- pour une méthode ou une classe, qu'il y aura des restrictions en terme d'héritage (nous aborderons ces restriction dans la section 1.8 qui traite de l'héritage).

#### 1.4.1.2 Méthodes d'instance et méthodes de classe

De la même façon qu'il existe des *variables d'instance* qui permettent de caractériser chacune des instances d'une classe et des *variables de classe* qui caractérisent l'ensemble des instances de la classe ; il existe deux types de méthodes : les *méthodes d'instance* et les *méthodes de classe*.

Les **méthodes d'instance** sont des méthodes dont l'exécution est déclenchée par une instance lorsque celle-ci reçoit un message. A contrario, l'exécution d'une **méthode de classe** est déclenchée par une classe lorsque cette dernière reçoit un message.

Dans la classe *Personne* représentée dans le diagramme de classe de la figure 1.10, il est clair que des méthodes telles que *getNom()* ou *setPrenom(...)* doivent être exécutées par des instances pour retourner le *nom* ou modifier le *prenom* de l'instance et, de ce fait, sont des méthodes d'instance. En revanche, les méthodes *getNbPersonnes()* ou *incrementNbPersonnes()* doivent, elles, être exécutées par la classe *Personne* elle-même afin de retourner ou d'incrémenter le nombre de *Personne* créées par la classe. Ainsi, ces deux méthodes sont des méthodes de classe.

#### 1.4.1.3 Accesseurs et mutateurs

Des méthodes spécifiques sont définies pour lire les valeurs des variables d'instance d'un objet : les accesseurs (*getters*). D'autres permettent d'affecter de nouvelles valeurs aux attributs : les mutateurs (*setters*).

Pour la classe *Personne* il est donc nécessaire de fournir des accesseurs et des mutateurs pour les variables d'instance *nom*, *prenom*, *anneeDeNaissance* et *nationalite*. Par

exemple, pour la variable *anneeDeNaissance* on écrira l'accesseur *getAnneeDeNaissance* et le mutateur *setAnneeDeNaissance*.

Les noms des accesseurs et des mutateurs permettent de déterminer ce qu'il sont (accesseurs ou mutateurs) et sur quelles variables ils portent. Il devront être de la forme :

- *getNomDeVariable()* pour un accesseur à la variable *nomDeVariable*,
- *setNomDeVariable(...)* pour un mutateur de la variable *nomDeVariable*.

#### 1.4.1.4 La classe Personne

Listing 1.1 – La classe *Personne*

```

1  class Personne {
2      // Constantes de classe
3      static final String NATIONALITE_PAR_DEFAUT="française";
4      // Variables de classe
5      static int nbPersonnes;
6      // Variables d'instance
7      String nom;
8      String prenom;
9      int anneeDeNaissance;
10     String nationalite;
11
12    // Méthodes
13    // Accesseurs et Mutateurs
14    String getNom() {
15        return this.nom;
16    }
17
18    void setNom(String nom) {
19        // affecte l'argument "nom" à
20        // son attribut "nom" (this.nom)
21        this.nom = nom;
22    }
23
24    String getPrenom() {
25        return this.prenom;
26    }
27
28    void setPrenom(String prenom) {
29        // affecte l'argument "prenom"
30        // à son attribut "prenom" (this.prenom)
31        this.prenom = prenom;
32    }
33
34    int getAnneeDeNaissance() {
35        return this.anneeDeNaissance;
36    }
37
38    void setAnneeDeNaissance(int anneeDeNaissance) {
39        // affecte l'argument "anneeDeNaissance"
40        // à son attribut
41        // "anneeDeNaissance" (this.anneeDeNaissance)
42        this.anneeDeNaissance = anneeDeNaissance;
43    }
44
45    String getNationalite() {
46        return this.nationalite;
47    }
48
49    void setNationalite(String nationalite) {
```

```

50     this.nationalite = nationalite;
51 }
52
53 // Autres méthodes //
54 // Méthodes de classe
55
56 static int getNbPersonnes(){
57     return nbPersonnes;
58 }
59
60 static void incrementeNbPersonnes(){
61     nbPersonnes++;
62 }
63
64 // Méthodes d'instance
65 int age(int annee){
66     // Retourne l'âge de la Personne l'année donnée
67     // le jour de son anniversaire.
68     return annee - getAnneeDeNaissance();
69 }
70 }
```

**Constante de classe de la classe *Personne*** La constante de classe *NATIONALITE\_PAR\_DEFAUT* aura pour valeur la chaîne de caractères “française” pendant toute la durée de vie de la classe *Personne*.

**Variable de classe de la classe *Personne*** La classe *Personne* possède également une variable de classe entière, *nbPersonnes*, lui permettant de garder un compte des *Personne* existantes.

**Variables d’instance de la classe *Personne*** Dans l’exemple 1.1 la classe *Personne* considère qu’une personne peut être représentée par son *nom*, son *prenom*, son *anneeDeNaissance* et sa *nationalité*. Ces informations sont représentées par les variables d’instance de la classe :

- les chaînes de caractères (*String*) *nom*, *prenom* et *nationalite* ;
- l’entier *anneeDeNaissance*.

### Méthodes d’instance de la classe *Personne*

**Accesseurs** Des accesseurs sont définis pour chacune de ces variables d’instance :

- *getNom()* permet de demander son nom à une *Personne*. Il sera retourné sous la forme d’une chaîne de caractères (*String*).
- *getPrenom()* permet de demander son prénom à une *Personne*. C’est aussi une chaîne de caractères (*String*).
- *getAnneeDeNaissance()* permet de demander son année de naissance à une *Personne*. L’accesseur retourne un entier (*int*).
- *getNationalite()* permet de demander sa nationalité à une *Personne*. C’est encore une chaîne de caractères (*String*).

**Mutateurs** La classe définit également des mutateurs pour chacune de ses variables d’instance :

- *setNom(String nom)* permet de donner un nom à une *Personne*.
- *setPrenom(String prenom)* permet de donner un prenom à une *Personne*.

- `setAnneeDeNaissance(int annee)` permet de fixer l'année de naissance d'une *Personne*.
- `setNationalite(String nationalite)` permet de fixer la nationalité d'une *Personne*.

**Autre méthode** En outre, la classe *Personne* définit une méthode permettant de calculer l'âge qu'aura une personne, une année donnée, le jour de son anniversaire : la méthode `int age(int annee)`.

**Méthodes de classe de la classe Personne** La variable de classe `nbPersonnes` est accessible en lecture et en écriture grâce à deux méthodes de classe :

- `getNbPersonnes()` qui retourne la valeur de la variable de classe `nbPersonnes`,
- `incrementeNbPersonnes()` qui permet d'incrémenter la variable de classe `nbPersonnes`.

#### 1.4.1.5 Envoi de messages

**Déclenchement d'une méthode d'instance** Pour déclencher l'exécution d'une méthode d'instance définie dans une classe, on doit envoyer un message à une instance de la classe.

On distingue plusieurs éléments dans un envoi de message :

- le récepteur du message : c'est un objet ;
- le sélecteur du message : c'est le nom de la méthode que l'on souhaite que l'objet exécute ;
- les arguments : ce sont les arguments qui doivent être passés à la méthode.

```
receveur.sélecteur(arg1, arg2, ..., argN);
```

**Déclenchement d'une méthode de classe** Le déclenchement d'une méthode de classe se fait de la même façon, à la différence que le receveur du message est une classe. Par exemple pour déclencher la méthode `incrementeNbPersonnes()` de la classe *Personne*, il suffit d'envoyer un message de sélecteur `incrementeNbPersonnes` à la classe *Personne* de la façon suivante :

```
Personne.incrementeNbPersonnes();
```

#### 1.4.1.6 Un objet particulier : *this*

Toute méthode d'instance connaît un objet particulier : l'objet *this*, celui qui exécute la méthode.

Cet objet peut être utilisé en tant que receveur d'un message ou comme argument passé à la méthode identifiée par le sélecteur du message.

Lorsque l'objet *this* est utilisé en tant que receveur du message, il peut être omis. Bien qu'il ne soit pas strictement nécessaire, certains programmeurs préfèrent le faire figurer dans le message pour insister sur l'objet auquel le message est envoyé. Ainsi, la méthode :

```
void bar(){
    // Faire figurer this dans un message
    // permet d'insister sur le fait que le message
    // est envoyé à l'objet en train d'exécuter la
    // méthode bar
    this.foo();
}
```

est strictement équivalente à :

```
void bar(){
    // Le sélecteur foo est envoyé à l'objet
    // en train d'exécuter la méthode bar
    // mais ce n'est pas explicite dans le code.
    foo();
}
```

#### 1.4.1.7 L'instanciation et le constructeur par défaut

Une classe est chargée de la création de ses instances. En fait, on peut voir une classe comme le moule qui permet de créer ses instances. L'opération de création d'objets est appelée l'instanciation. Elle est prise en charge par les constructeurs de la classe. Toute classe possède au moins un constructeur. La classe *Personne* de l'Exemple 1.1, par exemple, possède un constructeur permettant de créer des instances de *Personne* dont les noms, prénoms, date de naissance et nationalité ont des valeurs par défaut : la chaîne de caractères vide "" pour le type *String* et 0 pour le type *int* : pas très intéressant ! On appelle ce constructeur le constructeur par défaut de la classe *Personne*. Il est appelé à l'aide de l'opérateur *new* de la façon suivante :

```
Personne unePersonne = new Personne();
```

La ligne de Java ci-dessus ne se contente pas de créer une instance de la classe *Personne*, elle prend également la peine de sauvegarder l'instance créée dans la variable *unePersonne* de type *Personne* (bien sûr). Elle est en fait constituée de deux parties :

1. la déclaration d'une variable de type *Personne* dont le nom est *unePersonne* :

```
Personne unePersonne;
```

2. la création d'une instance grâce au constructeur par défaut de la classe *Personne* et sa sauvegarde dans la variable *unePersonne* :

```
unePersonne = new Personne();
```

Une fois qu'elle a été créée par un constructeur, on peut commencer à envoyer des messages à l'instance *unePersonne*. Notamment, pour appeler ses mutateurs et initialiser ses attributs. Par exemple, si la variable *unePersonne* doit représenter M. Jean Durand né en 1990 on l'initialisera comme suit :

```
unePersonne.setPrenom("Jean");
unePersonne.setNom("Durand");
unePersonne.setAnneeDeNaissance(1990);
unePersonne.setNationalite("belge");
```

Si on souhaite maintenant utiliser la variable de classe *nbPersonnes* pour tenir à jour le nombre d'instances de la classe *Personne* il sera nécessaire d'utiliser la méthode de classe prévue à cette effet.

```
Personne.incrementeNbPersonnes();
```

Une fois qu'elle a été initialisée la variable *unePersonne* pourra décliner son identité grâce aux accesseurs de la classe. Ainsi, la ligne de code :

```
System.out.println(unePersonne.getPrenom()
+ " " + unePersonne.getNom()
+ " - " + unePersonne.getAnneeDeNaissance()
+ " " + unePersonne.getNationalite());
```

affichera le message :

```
Jean Durand - Année De Naissance : 1990 Nationalité : belge
```

De la même manière, la classe *Personne* pourra indiquer le nombre de ses instances grâce à la ligne de code :

```
System.out.println("Nombre de Personnes : "
+ Personne.getNbPersonnes());
```

qui affichera le message :

```
Nombre de Personnes : 1
```

#### 1.4.1.8 Cas particulier de l'instanciation pour les classes “génériques”

Certaines classes en Java sont dites génériques. Les classes génériques sont des classes qui sont paramétrées par un autre type. C'est, en particulier, le cas des classes représentant des collections : les listes (*ArrayList<T>* par exemple), les dictionnaires (*HashMap<K,V>* par exemple), etc. Ces collections doivent être paramétrées par les types des objets qu'elles contiennent.

Dans le cas d'une *ArrayList<T>* le type précis de la liste dépendra du type *T* des éléments qu'elle contient. Pour une *HashMap<K,V>* son type précis dépendra du type *K* des clefs du dictionnaire et du type *V* des valeurs que l'on souhaite stocker dans le dictionnaire.

On utilisera ainsi, par exemple :

- une variable de type *ArrayList<String>* pour représenter une liste de chaînes de caractères ;

- une variable de type `HashMap<String,Personne>` pour représenter un dictionnaire dont la clef est une chaîne de caractères et le contenu une *Personne*.

Le constructeur d'une classe générique comprend à la fois le nom du type générique et les types qui le paramètrent. On aura ainsi les constructeurs `ArrayList<String>()` pour créer des listes de chaînes de caractères et `HashMap<String,Personne>()` pour construire des instances de dictionnaires dont la clef est une chaîne de caractères et les valeurs des *Personne*.

Pour déclarer et créer des instances de classes génériques on procédera comme suit :

```
/*
 * Déclaration et création d'une liste
 * de chaînes de caractères
 */
ArrayList<String> l=new ArrayList<String>();

/* Déclaration et création d'un dictionnaire
 * contenant des Personnes et indexés par
 * des chaînes de caractères (le nom de la
 * Personne par exemple)
 */
HashMap<String, Personne> annuaire
    = new HashMap<String,Personne>();
```

**Important :** L'opérateur `new` permettant de créer des instances d'une classe, sert en fait dans la création de toute variable d'un type référence.

Les tableaux étant des types référence, l'opérateur `new` est utilisé pour leur création :

```
int[] tab; // Déclaration d'un tableau d'entiers
tab = new int[10]; /* Création d'un tableau de
                     * 10 entiers et affectation
                     * de ce tableau à la
                     * variable tab
                     */
```

**Remarque :** la classe `java.util.ArrayList<T>` permet de représenter des tableaux dynamiques dont la taille n'est pas fixée par avance et qui s'adaptent au nombre d'éléments qu'on leur ajoute.

#### 1.4.2 Quelques exercices

Les exercices suivants ont pour objectifs :

1. de vous familiariser avec l'écriture d'une classe,
2. de créer des instances à l'aide du constructeur par défaut de la classe,
3. d'envoyer des messages à des objets pour leur demander d'exécuter des méthodes.

**Exercice 1 :** Écrire la classe *Personne* de l'exemple 1.1 dans le fichier *Personne.java* et compiler le fichier.

**Exercice 2 :** Dans la classe *Personne*, ajouter la méthode *identite()* qui retourne une chaîne de caractères de la forme :

```
<prénom> <nom> - Né(e) : <année> Nationalité: <nationalité>
```

**Exercice 3 :** Écrire la classe exécutable *PersonneEssai* dont la méthode principale (*main*) crée trois instances, les initialise avec des données au choix et affiche l'identité des instances créées. Compiler le fichier et vérifier que la classe produite s'exécute correctement.

**Exercice 4 :** On souhaiterait maintenant qu'une *Personne* puisse avoir un compte sur un système informatique dont le nom d'utilisateur ("user identifier" ou "user id") :

- est constitué :
  - des 5 premières lettres du nom de la personne ou de la totalité de son nom si il fait moins de 5 caractères,
  - suivi des trois premières lettres de son prénom ou de la totalité de son prénom si celui ci fait moins de 3 caractères,
  - suivi des deux derniers chiffres de son année de naissance.
- ne comprend aucun espace, apostrophe, lettre majuscule ou caractère accentué :
  - les caractères accentués seront remplacés par leurs équivalents sans accent :
    - 'é', 'è', 'ê', 'ë' par 'e',
    - 'ô' par 'o',
    - 'û', 'ù', 'ü' par 'u',
    - 'à', 'â' par 'a',
    - 'î', 'î' par 'i',
    - 'ç' par 'c'.
  - les espaces et apostrophes seront tout simplement supprimés.

Quelques exemples de noms d'utilisateurs :

- Jean Durand né en 1987 aura pour nom d'utilisateur *duranjea87*
- Al Capone né en 1899 aura pour nom d'utilisateur *caponal99*
- James Bond né en 1920 aura pour nom d'utilisateur *bondjam20*
- Jean de la Fontaine (s'il était) né en 2001 aurait pour nom d'utilisateur *delafjea01*

Dans la classe *Personne* ajouter une méthode *userId()* qui retourne le nom d'utilisateur de la *Personne*. La méthode *userId()* pourra s'appuyer sur tout autre méthode auxiliaire que vous jugerez utile d'écrire. L'ajout de méthodes permettant de nettoyer les nom et prénom (les transformer en lettres minuscules, supprimer les espaces et les apostrophes et supprimer les caractères accentués) devrait s'avérer utile. On consultera la documentation de la classe *String* de l'API de Java<sup>3</sup> pour y trouver les méthodes de manipulation de chaînes de caractères qui pourront servir dans ces traitements.

**Exercice 5 :** Modifier la méthode principale de la classe *PersonneEssai* pour qu'elle affiche maintenant l'identifiant de chaque *Personne* après son identité ;

On pensera à afficher le nombre total d'instances de la classe *Personne* créées.

---

3. <https://docs.oracle.com/javase/8/docs/api/index.html>

**Exercice 6 :** Écrire la classe *UserId* à qui on passe au moins trois arguments : un nom, un prénom, une nationalité et, éventuellement, une année de naissance (dans cet ordre) et qui affiche l'*identité* de la personne telle qu'elle est retournée par la méthode *identite()* de la classe *Personne*, suivi de son nom d'utilisateur.

Par exemple, on souhaite que l'appel :

```
java UserId Noam Chomsky américaine 1928
```

affiche le message suivant :

```
Noam Chomsky - Né(e) : 1928 Nationalité : américaine => chomsnoa28
```

Dans les cas où aucune année de naissance n'est passée on souhaite que la valeur 0 soit affectée à cet attribut pour indiquer que sa valeur est manquante. Pour les utilisateurs dont l'année de naissance n'est pas connue, on souhaiterait que la partie de l'identifiant utilisateur provenant de l'année de naissance soit "XX". Ainsi, l'appel :

```
java UserId Einstein Albert suisse
```

affichera le message :

```
Albert Einstein - Né(e) : 0 Nationalité : suisse => einstalbXX
```

Il est donc nécessaire de modifier la méthode *userId()* de la classe *Personne* pour qu'elle tienne compte de ce cas de figure.

**Transférer une chaîne de caractères :** Seules des chaînes de caractères peuvent être passées sur la ligne de commande. Dans l'exemple ci-dessus, le troisième paramètre ("1928") est donc passé en tant que chaîne de caractères. Si on veut utiliser cette information en tant qu'*anneeDeNaissance* d'une *Personne* il est nécessaire de la convertir : depuis la chaîne de caractères "1928" vers l'entier 1928. La méthode *parseInt* de la classe de l'API *Integer* est toute indiquée (voir documentation officielle de l'API).

## 1.5 Constructeurs

Dans l'état actuel de nos connaissances, la création d'une instance de classe spécifique est une opération en deux phases :

1. création d'une instance *vide* à l'aide du constructeur par défaut :

```
Personne personne=new Personne();
```

2. incrémentation du nombre d'instances créées :

```
Personne.incrementeNbPersonnes();
```

3. initialisation des valeurs de ses attributs grâce aux mutateurs de la classe :

```
personne.setPrenom("Jean");
personne.setNom("de la Fontaine");
personne.setAnneeDeNaissance(2005);
personne.setNationalite("française");
```

C'est une tâche un peu fastidieuse, surtout si elle doit être répétée de nombreuses fois. Il serait souhaitable de pouvoir bénéficier d'un moyen de créer des instances dont les attributs sont initialisés à la création et de telle sorte à ce que le nombre d'instances soient maintenu à jour.

Ce moyen existe au travers des constructeurs.

Les constructeurs d'une classe font partie de sa définition et portent nécessairement le même nom que la classe (majuscule et minuscule comprises). Ils permettent de créer des instances initialisées dès leur création. Une classe peut avoir plusieurs constructeurs pour peu qu'ils diffèrent par le nombre et les types de leurs arguments.

Si on considère la classe *Personne* représentée dans la figure 1.10 il serait intéressant de pouvoir en créer des instances dont le nom, le prénom et la date de naissance sont connues à la création. Il pourrait également être intéressant de créer des *Personne* dont on ne connaît que le nom et le prénom ; l'instance pourrait indiquer l'absence de date de naissance en donnant une valeur particulière (0 par exemple) à l'attribut correspondant. Les deux constructeurs suivants doivent être définis pour obtenir ce résultat :

```
Personne(String prenom, String nom, String nationalite,
          int anneeDeNaissance)
Personne(String prenom, String nom)
```

### 1.5.1 Un premier constructeur

Dans sa définition, un constructeur ne se charge que d'initialiser l'instance qu'il crée. Cette opération consiste en l'initialisation des valeurs des attributs de l'instance et de l'éventuelle mise en place des relations qui doivent exister entre l'instance créée et son environnement.

Si on considère par exemple le constructeur qui crée une *Personne* étant donnés, un nom, un prénom, une nationalité et une année de naissance. Il pourrait s'écrire comme suit :

---

**Exemple 1.5** Constructeur de *Personne* à 4 paramètres

---

```

1 /**
2  * Constructeur permettant de créer des instances
3  * de Personne dont on connaît le nom, le prénom
4  * et l'année de naissance
5 */
6 Personne(String prenom,
7           String nom,
8           String nationalite,
9           int anneeDeNaissance){
10    // Initialisation de l'attribut nom avec la
11    // valeur de l'argument nom
12    this.nom=nom;
13    // Initialisation de l'attribut prenom avec la
14    // valeur de l'argument prenom
15    this.prenom=prenom;
16    // Initialisation de l'attribut nationalite
17    // avec la valeur de l'argument nationalite
18    this.nationalite=nationalite;
19    // Initialisation de l'attribut anneeDeNaissance
20    // avec la valeur de l'argument anneeDeNaissance
21    this.anneeDeNaissance=anneeDeNaissance;
22    // Incrémentation du nombre d'instance
23    // de la classe
24    Personne.incrementeNbPersonnes();
25 }
```

---

Ce constructeur admet 4 arguments permettant d'initialiser les quatre variables *nom*, *prenom*, *anneeDeNaissance* et *nationalite* et se charge d'incrémenter le compteur d'instances de la classe. Dans l'exemple ci-dessus les paramètres du constructeur et les attributs de la classe portent les mêmes noms. Ceci n'est pas nécessairement le cas mais impose d'utiliser le mot-clé *this* lors de l'initialisation des attributs de l'instance. En effet, lorsque, dans le constructeur on fait référence à *prenom* (par exemple), il s'agit du paramètre *prenom*.

### 1.5.1.1 Un mot sur la portée des variables

Lorsqu'on fait référence à une variable en utilisant son nom, la variable est recherchée dans le code de la façon suivante :

1. On recherche la variable, en remontant dans le code, dans le bloc d'instruction courant (un bloc d'instructions est délimité par des accolades '{' et '}').
2. Si la variable n'est pas trouvée dans le bloc courant on la recherche dans les blocs englobants jusqu'à atteindre le bloc de définition d'une méthode ou d'un constructeur.
3. Si la variable n'a pas été trouvée dans la définition de la méthode ou du constructeur dans lequel elle est utilisée, elle est recherchée dans les paramètres de la méthode ou du constructeur.

4. Si la variable n'a pas été trouvée dans les paramètres de la méthode ou du constructeur dans lequel elle est utilisée, elle est recherchée dans les attributs de la classe.
  5. Si la variable n'a pas été trouvée dans les attributs de la classe c'est qu'elle n'est pas définie.

Dans le constructeur à quatre paramètres de la classe *Personne* présenté ci-dessus, en ligne 14, lorsqu'il est fait référence à la variable *prenom* il s'agit du paramètre *prenom* du constructeur. En effet :

1. aucune variable nommée *prenom* n'est définie dans le bloc courant (le bloc de définition du constructeur, compris entre l'accolade ouvrante de la ligne 8 jusqu'à l'accolade fermante de la ligne 24),
  2. un paramètre du constructeur se nomme *prenom*. C'est donc bien du paramètre *prenom* du constructeur qu'il s'agit.

Dans le constructeur de la classe *Personne*, si on souhaite initialiser l'attribut *prenom* avec la valeur du paramètre *prenom* du constructeur; il faut donc trouver un moyen de différencier ces deux variables différentes et qui portent pourtant le même nom.

Nous venons de voir que, dans le constructeur de *Personne* à trois arguments, le symbole *prenom* fait référence au paramètre *prenom* du constructeur. L'attribut *prenom* de la *Personne* en cours de construction est, quant à lui, référencé par *this.prenom* (*this* fait ici référence à l'instance en cours de création par le constructeur).

### **1.5.2 Utilisation des constructeurs**

L'opérateur *new*, que nous avons déjà utilisé avec le constructeur par défaut (sans argument), est également utilisé pour créer des instances avec les autres constructeurs de la classe. Ainsi, pour créer la *Personne* qui représente Nelson Mandela né en 1918 et de nationalité sud africaine et stocker l'instance dans la variable *personne* on procédera comme suit :

Ainsi, avec ce nouveau constructeur à 4 paramètres l'exemple présenté en 1.4.1.7 devient :

```
Personne personne = new Personne("Jean",
                                  "de la Fontaine",
                                  "francais", 2005);
```

et le nouveau constructeur de la classe *Personne* se charge à la fois de la construction de l'instance et de l'initialisation de ses attributs.

### 1.5.3 Un deuxième constructeur

Regardons maintenant le constructeur à deux paramètres qui crée une *Personne* dont seuls les nom et prénom sont donnés et qui donne à l'année de naissance la valeur 0 pour indiquer l'absence de valeur et donne pour valeur à la nationalité la nationalité par défaut stockée dans la constante de classe *NATIONALITE\_PAR\_DEFATU*. Ce constructeur pourrait avoir la définition suivante :

---

#### Exemple 1.6 Constructeur à deux paramètres

---

```
/**  
 * Constructeur permettant de créer des instances de  
 * Personne dont on connaît le nom et le prénom.  
 * Les instances créées ont une année de naissance  
 * initialisée à 0 pour indiquer l'absence de valeur  
 * de cette information.  
 */  
Personne(String prenom, String nom){  
    setNom(nom);  
    setPrenom(prenom);  
    setAnneeDeNaissance(0);  
    /*  
     * NATIONALITE_PAR_DEFATU étant une constante de  
     * classe, c'est à la classe qu'on doit demander sa  
     * valeur.  
     */  
    setNationalite(Personne.NATIONALITE_PAR_DEFATU);  
    Personne.incrementeNbPersonnes();  
}
```

---

### 1.5.4 Factorisation du code des constructeurs avec *this(...)*

On peut remarquer que le constructeur à deux paramètres est très semblable au constructeur à quatre paramètres. C'est en fait un cas particulier du constructeur à quatre paramètres. En effet, construire une instance de *Personne* dont on connaît le nom et le prénom revient à construire une instance de *Personne* dont on connaît le nom et le prénom, dont l'année de naissance est 0 et la nationalité est celle étant définie comme la nationalité par défaut.

Les deux appels : `Personne(prenom, nom)` et

`Personne(prenom, nom, Personne.NATIONALITE_PAR_DEFATU, 0)` sont donc équivalents.

Il serait intéressant que cette équivalence puisse se traduire dans le code Java décrivant le constructeur à deux arguments.

Dans le code d'un constructeur quelconque on peut toujours utiliser le constructeur particulier : `this(...)` qui fait référence à un constructeur de la même classe dont le nombre et le type des arguments correspondent à ceux passés à `this(...)`.

Ainsi, le constructeur de *Personne* à deux arguments peut se réécrire de manière plus concise comme suit :

```
/**
 * Constructeur permettant de créer des instances
 * de Personne dont on connaît le nom et le prénom.
 * Les instances créées ont une année de naissance
 * initialisée à 0 pour indiquer l'absence de valeur
 * de cette information et la nationalité par défaut.
 */
Personne(String prenom, String nom){
    /*
     * Appelle un constructeur de la même classe
     * (la classe Personne) comportant 4 paramètres :
     * - les trois premiers de type String
     * - le quatrième de type int
     */
    this(prenom, nom, Personne.NATIONALITE_PAR_DEFAUT, 0);
}
```

La définition ci-dessus peut se comprendre comme : “construire une instance de *Personne* dont on connaît le *nom* et le *prénom* revient à construire une instance de *Personne* dont on connaît le *nom*, le *prénom*, dont l’année de naissance est 0 et dont la nationalité a la valeur par défaut”.

**Remarque importante :** Si *this(...)* est utilisé, ce doit être la première instruction du constructeur. Une conséquence de cette règle est que *this(...)* ne peut être utilisé qu’une seule fois dans un constructeur (s’il était utilisé plusieurs fois, seule sa première utilisation serait la première instruction du constructeur et les autres seraient interdites).

### 1.5.5 À savoir absolument sur les Constructeurs

La règle suivante concernant les constructeurs d’une classe est extrêmement importante et doit être connue **par cœur**. Ce sera très utile pour régler des problèmes de compilation.

**Toute classe Java possède au moins un constructeur :**

1. Ceux définis par le programmeur s’il y en a ;
2. Un constructeur par défaut sans paramètre ajouté par le compilateur sinon.

### 1.5.6 Surcharge (overloading) de méthodes et constructeurs

Lorsqu’une classe propose plusieurs définitions d’une méthode portant le même nom ou d’un constructeur qui diffèrent seulement par les nombres et/ou les types de leurs arguments on parle de **surcharge de méthode ou de constructeurs**. On dit aussi que la méthode ou le constructeur est surchargé. En anglais on parle de “**method overloading**”. Les différents constructeurs que nous avons écrits dans les exemples précédents sont des exemples de surcharge : la classe *Personne* par exemple propose le constructeur à quatre paramètres de l’exemple 1.5 et celui à deux arguments de l’exemple 1.6. Ces deux constructeurs portent le même nom et diffèrent par le nombre de leurs arguments.

En Java, la surcharge peut s'appliquer de la même manière à toute méthode que ce soit une méthode d'instance ou une méthode de classe.

---

### 1.5.7 Quelques exercices

Les exercices suivants ont pour objectifs :

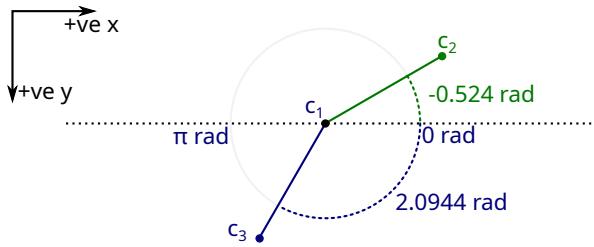
1. de vous familiariser avec l'écriture de constructeurs,
2. de vous faire entrevoir des problèmes couramment rencontrés avec les constructeurs,
3. de récapituler les concepts vus depuis le début de ce document à travers une implémentation complète de quelques classes.

**Exercice 1 :** Ajouter les constructeurs présentés dans les exemples 1.5 et 1.6 à la classe *Personne*. Dans la classe *PersonneEssai* construire de nouvelles instances de *Personne* en utilisant les deux nouveaux constructeurs et afficher les objets créés.

Compiler les deux fichiers *Personne.java* et *PersonneEssai.java* et exécuter la classe *PersonneEssai*.

#### Exercice 2 : Premiers pas en géométrie : des coordonnées

1. Définir la classe *Coordonnees* représentant les coordonnées des points du plan.  
La classe *Coordonnees* devra fournir : des méthodes permettant de
  - déplacer ses instances vers des coordonnées données :  
*deplacerVers(double x, double y)*
  - déplacer ses instances selon un vecteur directionnel donné :  
*deplacerDe(double deltaX, double deltaY)*.On n'oubliera pas d'écrire des constructeurs dans la classe *Coordonnees*. Les deux constructeurs qu'elle devra nécessairement proposer sont :
  - un constructeur permettant de construire une instance de *Coordonnees* dont les abscisses et ordonnées sont connues ;
  - un constructeur par défaut permettant de créer une instance de *Coordonnees* par défaut, dont les abscisses et ordonnées seront fixées à 0.Outre, les méthodes et constructeurs ci-dessus, la classe *Coordonnees* comprendra :
  - une méthode permettant de calculer la distance entre deux *Coordonnees* :  
*distanceVers(Coordeunes coord)*;
  - une méthode permettant de calculer l'angle entre l'axe de abscisses et le segment défini par deux *Coordonnees* : *angleVers(Coordonnees coord)*.  
Si on considère les points (instances de *Coordonnees*)  $c_1$  et  $c_2$  tels que représentés sur la figure 1.11, l'appel :  $c_1.angleVers(c_2)$  devra retourner la valeur de l'angle  $\alpha = -0.524\text{rad}$ .
2. Écrire une classe *CoordonneesEssai* permettant de s'assurer que la classe *Coordonnees* est correctement implémentée. Dans sa méthode principale la classe *CoordonneesEssai* créera des instances de *Coordonnees* en utilisant les différents constructeurs de la classe et vérifiera, en affichant leurs attributs, qu'ils ont été correctement construits. La méthode devra aussi vérifier que les deux méthodes de déplacement fonctionnent telles que souhaitées.

FIGURE 1.11 – La méthode *angleVers*.

### Exercice 3 : Des rectangles

1. Définir la classe *Rectangle* représentant les formes géométriques de même nom. Un *Rectangle* est caractérisé par ses dimensions (sa largeur et sa hauteur) et un point désignant sa position (on a pour habitude, en informatique, de choisir le point situé sous le coin supérieur gauche d'un rectangle comme étant sa position). D'autre part, la classe *Rectangle* devra fournir, outre les accesseurs et mutateurs, des méthodes permettant de calculer l'aire et le périmètre de ses instances ainsi que des méthodes permettant de déplacer un *Rectangle*. La classe *Rectangle* proposera également des méthodes permettant d'accéder (en lecture et en écriture) à l'abscisse et à l'ordonnée de la position d'un *Rectangle*.

Enfin, la classe devra proposer les constructeurs suivants :

- un constructeur permettant de créer une instance de *Rectangle* dont on connaît la position, la largeur et la hauteur;
- un constructeur permettant de créer une instance de *Rectangle* dont on connaît la largeur et la hauteur et qui sera créée avec une position par défaut fixée à (0,0);
- un constructeur permettant de créer une instance de *Rectangle* dont on connaît la position et dont la largeur et la hauteur sont initialisées avec des valeurs par défaut;
- un constructeur permettant de créer une instance de *Rectangle* par défaut (largeur et hauteur par défaut et position par défaut).

2. Écrire une classe *RectangleEssai* permettant de s'assurer que la classe *Rectangle* est correctement implémentée. Dans sa méthode principale la classe *RectangleEssai* créera des instances de *Rectangle* en utilisant les différents constructeurs de la classe et vérifiera, en affichant leurs attributs, qu'ils ont été correctement construits. La méthode devra aussi vérifier que les autres méthodes de la classe fonctionnent telles que souhaité.

## 1.6 Paquetages (*packages*)

Nous avons, jusqu'à maintenant, écrit un certain nombre de classes : certaines représentant des personnes, d'autres des rectangles, certaines exécutant des tests ou des applications simples.

Pour l'instant rien ne permet de différencier les différents types de classes que nous manipulons, à part leurs noms. Tous les fichiers concernant ces classes sont stockés dans un même répertoire qui risque de devenir difficilement exploitable lorsqu'il contiendra plus de classes.

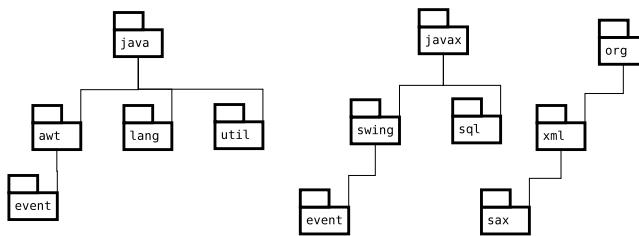


FIGURE 1.12 – Quelques paquetages de Java

Le langage Java propose la notion de paquetages (*packages*) pour organiser les différentes classes qui seront manipulées par un programme. Les paquetages (*packages*) de Java permettent de “ranger” ensemble des classes semblables. D’autre part, les paquetages (*package*) de Java peuvent être organisés hiérarchiquement. La notion de paquetage telle qu’elle est proposée par Java permet donc de proposer une organisation hiérarchique des classes d’un programme. On peut ainsi faire une analogie entre les notions de répertoires (dossiers) et de fichiers et celles de paquetage et de classe. En effet, tout comme un répertoire peut contenir des fichiers et des répertoires, un paquetage peut contenir des classes et d’autres paquetages. Le paquetage peut de ce fait être assimilé à un répertoire, tandis que les fichiers s’apparenteraient aux classes.

Une classe Java appartient toujours à un paquetage et cette appartenance doit être indiquée dans le code source de la classe. Dans les classes que nous avons créées pour le moment, il n’a pas été question de paquetage dans les définitions : elle sont donc “rangées” dans un paquetage particulier : *le paquetage par défaut*.

### 1.6.1 Paquetages et classes de l’API

Comme toutes les classes du langage, les classes fournies dans l’API sont organisées hiérarchiquement à l’aide de paquetages. La figure 1.12 montre un extrait de la structure de paquetages de l’API de Java. Comme on peut le constater dans ce schéma, il existe trois paquetages racine qui définissent ainsi trois arbres :

- `java` et ses sous paquetages : contiennent les classes de l’API qui étaient livrées en standard avec le JDK dès les premières versions du langage ;
- `javax` et ses sous paquetages : contiennent des classes qui initialement étaient des extensions de Java (*java extensions*) et devaient être téléchargées séparément des autres classes standard. Les classes de ce paquetage sont livrées avec le JDK depuis la version 4 de Java ;
- `org` et ses sous paquetages : contiennent les classes fournies par des tierces et qui sont fournies par défaut avec le JDK.

#### 1.6.1.1 Nommage des paquetages

Java propose une stratégie de nommage pour les paquetages :

- s’il s’agit d’un paquetage “racine”, son nom est simplement le nom du paquetage : par exemple les paquetages `java`, `javax` et `org`
- s’il s’agit d’un paquetage nœud ou feuille, son nom contiendra tous les paquetages dans la hiérarchie depuis la racine : par exemple les paquetages `java.awt.event`, `java.lang`, `javax.swing` ou encore `org.xml.sax`

Les classes de l'API de Java sont toutes “rangées” dans des paquetages. Les noms des paquetages (moyennant un certain “décodage”) permettent de connaître le type des classes qu'ils contiennent et peut de ce fait nous aider à trouver des classes dans l'API.

Prenons par exemple le paquetage `java.awt` :

- la dénomination `awt` correspond à *Abstract Window Toolkit*. Il s'agit de la boîte à outils dédiée à la construction des interfaces utilisateurs d'applications graphiques interactives. On trouvera dans ce paquetage les classes permettant de mettre en œuvre ce type d'application et d'implémenter des éléments graphiques tels que des boutons, des fenêtres, des menus, etc.
- *SQL* (Simple Query Language) est un langage d'interrogation de bases de données. Ainsi, le paquetage `javax.sql` contient les classes permettant d'interroger des bases de données depuis un programme écrit en Java,
- comme son nom l'indique, le paquetage `java.util` contient des classes “utilitaires” pouvant servir dans tout type d'application. C'est, par exemple, dans ce paquetage qu'on trouvera les classes Java permettant d'implémenter des structures de données standard : des piles, des files, des listes, des dictionnaires.

#### 1.6.1.2 Nommage complet des classes et importation de classes

Les classes que nous avons créées jusqu'à présent étaient toutes des classes du paquetage par défaut et leurs noms se réduisaient à celui choisi dans la déclaration de la classe : `Personne`, `PersonneEssai`, `Rectangle`, `UserId` ...

Lorsqu'une classe fait partie d'un paquetage, son nom complet contiendra le nom du paquetage en préfixe. Par exemple, si on considère la classe `LinkedList` du paquetage `java.util`, son nom complet est en fait : `java.util.LinkedList`.

Si une classe *A* fait référence à une autre classe *B* en utilisant son nom court. Le compilateur interprète cette référence à “la classe *B*” comme une référence à “la classe *B* qui se trouve dans le paquetage de la classe *A*”. Si les classes *A* et *B* se trouvent dans le même paquetage une telle interprétation ne sera pas problématique. En revanche, si les deux classes sont dans des paquetages différents, la référence à la classe *B* exprimée dans la classe *A* provoquera une erreur de compilation. Il faut dans ce cas utiliser le nom complet de la classe *B* pour y faire référence depuis la classe *A*.

Dans la classe `PersonneEssai` par exemple, les références faites à la classe `Personne` ne posent pas de problème au compilateur. En effet, les deux classes sont dans le même paquetage : le paquetage par défaut.

En revanche, si dans la classe `PersonneEssai` on souhaite faire référence à une classe d'un autre paquetage, par exemple la classe `ArrayList` du paquetage `java.util` contenue dans l'API, il faudra utiliser son nom long, dans le cas présent : `java.util.ArrayList`.

Les hiérarchies de paquetages pouvant être profondes et un nom long pouvant être constitué d'une longue succession de noms de paquetages, cette écriture peut vite devenir lourde. Pour cette raison, le langage Java propose une instruction d'importation qui permet d'utiliser le nom court d'une classe y compris dans les cas où il s'agit d'une classe d'un autre paquetage. Cette instruction, qui doit être écrite entre la déclaration du paquetage de la classe et avant le début de la définition de la classe, s'utilise de la façon suivante :

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

et peut être interprétée comme une indication au compilateur que lorsqu'il sera fait mention de la classe `ArrayList` en utilisant son nom court, il s'agira plus précisément de

la classe `java.util.ArrayList`.

**Remarque importante :** le paquetage `java.lang` est particulier dans la mesure où il contient des classes indispensables au bon fonctionnement de Java. Par exemple, c'est dans ce paquetage qu'est définie la classe `String` qui représente les chaînes de caractères. Les classes du paquetage `java.lang` étant fréquemment utilisées, pour éviter d'alourdir inutilement le code, leur importation n'est pas nécessaire. Ce sont les seules classes dans ce cas.

Le mécanisme de nommage de Java permet la cohabitation de plusieurs classes de mêmes noms courts appartenant à des paquetages différents. Des classes de mêmes noms et appartenant à des paquetages différents sont fournies dans l'API de Java. Par exemple, il existe trois classes différentes dont le nom court est `InputStream` pour représenter des flux d'entrée :

- la classe `java.io.InputStream` du paquetage `java.io` contenant les classes de l'API chargées des entrées/sorties ;
- la classe `org.omg.CORBA.portable.InputStream` du paquetage `org.omg.CORBA.portable` contenant les classes permettant de gérer des applications s'appuyant sur l'architecture logicielle CORBA ;
- la classe `org.omg.CORBA_2_3.portable.InputStream` du paquetage `org.omg.CORBA_2_3.portable` contenant les classes permettant de gérer des applications s'appuyant sur la version 2.3 de l'architecture logicielle CORBA.

Si on souhaite utiliser plusieurs classes de même nom dans un programme il sera nécessaire d'utiliser le nom complet des classes pour éviter toute ambiguïté.

### 1.6.2 Définition du paquetage d'une classe

Tout comme les classes de l'API sont organisées en paquetages, il est possible (et même très vivement recommandé) de définir des paquetages pour contenir les classes développées dans le cadre d'un programme Java.

L'inclusion d'une classe à un paquetage se fait à plusieurs niveaux différents. En effet, il est nécessaire pour cela de :

1. ranger le fichier de définition de la classe dans un répertoire dont le nom est défini par le nom du paquetage ;
2. déclarer, dans le code source de la classe, son appartenance au paquetage ;
3. s'assurer que les importations nécessaires sont faites.

#### 1.6.2.1 Paquetages et noms de répertoires

Nous avons vu comment les paquetages de Java étaient organisés hiérarchiquement. Parallèlement à cette structure arborescente des paquetages il existe une structure arborescente de répertoires respectant la même hiérarchie.

Il existe une convention qui régit les noms des paquetages des différents niveaux de la hiérarchie.

Un nom de paquetage ne doit contenir que des chiffres et des lettres minuscules (de '0' à '9' et de 'a' à 'z').

Le paquetage de plus haut niveau (la racine de l'arborescence) a pour nom : `com`, `org`, `net`, `edu` ou un code pays à deux lettres (`fr` par exemple).

On a pour habitude de donner au paquetage de niveau inférieur le nom de l'organisation dans laquelle il est développé : dans notre cas ça pourra être `eseo`.

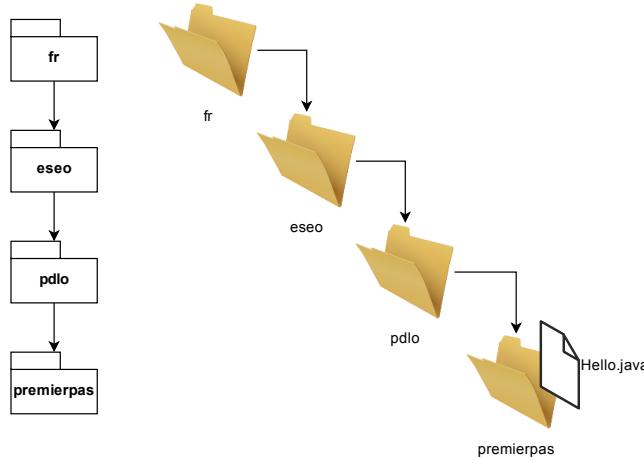


FIGURE 1.13 – Deux hiérarchies parallèles

Pour les paquetages des niveaux hiérarchiques suivants, on a pour habitude de donner le nom de l’application puis des noms reflétant la sémantique des classes contenues.

Par exemple, si nous considérons la classe *Hello* de l’exemple 1.1 ; on pourrait vouloir l’intégrer à un paquetage *fr:eseo:pdlo:premierspas* . En effet, nous sommes en France, à l’ESEO et cette classe, écrite dans le cadre du PDLO, constituait notre premier pas en Java. C’est donc dans un répertoire *fr/eseo/pdlo/premierspas* que nous devrions “ranger” le fichier *Hello.java* tel que l’indique la figure 1.13. Dans cette figure, le répertoire *java* est le répertoire de travail dans lequel l’”application” *Hello* est stockée.

#### 1.6.2.2 Déclaration de l’appartenance d’une classe à un paquetage

Outre le stockage dans le répertoire adéquat, l’appartenance d’une classe à un paquetage doit être indiqué dans le code source de la classe. L’instruction suivante :

```
package fr.eseo.pdlo.premierspas;
```

doit être ajoutée en première ligne du fichier *Hello.java*. Cette instruction ne sera prise en compte dans le programme qu’après la compilation de la classe.

#### 1.6.2.3 Compilation et exécution d’une classe dans un paquetage

La structure de répertoires introduite lors de la définition des paquetages modifie légèrement le mécanisme de compilation. Il est maintenant nécessaire de se placer dans le répertoire de travail (*java* dans notre exemple) pour compiler le fichier *Hello.java*. Une fois dans le répertoire *java* la ligne de commande suivante permettra de compiler le fichier *Hello.java* :

```
javac fr/eseo/cpoo/premierspas/Hello.java
```

C’est logique car lorsqu’on se place dans le répertoire de travail *java*, le fichier à compiler est *fr/eseo/pdlo/premierspas/Hello.java* .

Comme pour la compilation, l'exécution de la classe *Hello* doit maintenant se faire depuis le répertoire de travail *Java* et en donnant le nom complet de la classe grâce à la ligne de commande suivante :

```
java fr.eseo.pdlo.premierspas.Hello
```

Encore une fois, les choses sont très logiques car le nom de la classe *Hello* du paquetage

*fr.eseo.pdlo.premierspas* est  
*fr.eseo.pdlo.premierspas.Hello*  
et c'est bien cette classe qu'on souhaite exécuter.

**Remarque :** il est possible que la profondeur de la structure hiérarchique vous semble inquiétante vis-à-vis de la longueur des commandes qui devront être tapées pour accéder aux différents répertoires, compiler les fichiers ou exécuter les classes. N'oubliez pas que les langages de commandes proposées par les différents systèmes d'exploitation proposent un mécanisme d'auto-complétion lorsqu'on appuie sur la touche tabulation.



### 1.6.3 Quelques exercices

Les exercices suivants ont pour objectifs :

1. d'introduire la modularité possible grâce aux paquetages,
2. d'assimiler les aspects pratiques permettant de définir des paquetages et d'y intégrer des classes.

#### Exercice 1 :

1. Faire les modifications décrites dans la section 1.6.2 pour intégrer la classe *Hello* au paquetage *fr.eseo.pdlo.premierspas*

Ne pas oublier de compiler le fichier *Hello.java* une fois que les modifications sont faites et d'exécuter la classe afin de s'assurer qu'elle fonctionne encore.

2. Procéder de la même manière pour placer la classe *Bonjour* dans le paquetage *fr.eseo.pdlo.premierspas*

#### Exercice 2 :

1. Faire les modifications nécessaires à la création des paquetages suivants :
  - *fr.eseo.pdlo.exercices.utilisateurs* pour les classes *Personne* et *UserId*
  - *fr.eseo.pdlo.projet.geom* pour les classes *Coordonnees* et *Rectangle*
  - *fr.eseo.pdlo.essais.exercices.utilisateurs* pour la classe *PersonneEssai*
  - *fr.eseo.pdlo.essais.projet.geom* pour les classes *CoordonneesEssai* et *RectangleEssai*
2. Compiler toutes les classes et exécuter les classes d'essai.

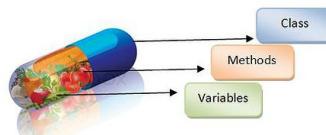


FIGURE 1.14 – Encapsulation des méthodes et des attributs dans une classe

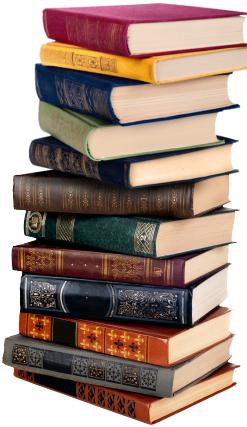


FIGURE 1.15 – Une pile ... de livres

3. Dans l'exercice précédent vous avez dû remarquer qu'une fois la mise en paquetages terminée, vous ne pouviez plus compiler votre code, ni, bien sûr l'exécuter. C'est normal ! Notez bien les messages d'erreurs qui vous sont rapportés par le compilateur, ne perdez pas votre temps à tenter de résoudre le problème et **lisez la section suivante**. Vous y trouverez la **solution au problème**.

## 1.7 Encapsulation et visibilité

Un concept important en programmation orientée objets est l'*encapsulation*. Ce principe consiste d'une part à "encapsuler" le comportement et la structure d'une famille d'objets au sein d'une classe (figure 1.14). Par ailleurs, l'*encapsulation* permet de dissocier les méthodes et attributs d'une classe servant à son implémentation de ceux proposant des services à l'extérieur.

### 1.7.1 Un exemple d'encapsulation : une pile de livres

L'exemple couramment utilisé pour illustrer le principe d'encapsulation est celui de la pile. Vue de l'extérieur, une pile est un empilement d'objets sur lequel on peut :

- ajouter un nouvel objet au sommet de la pile : empiler l'objet
- prendre l'objet au sommet de l'empilement : dépiler l'objet.

En outre, une pile doit savoir si elle est vide ou non et doit pouvoir être créée.

Si nous devions implémenter la classe des piles de livres en Java, elle devrait pouvoir être vue de l'extérieur telle que le montre la figure 1.16.

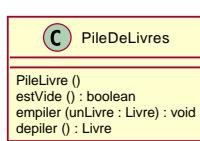


FIGURE 1.16 – pour l’extérieur    FIGURE 1.17 – pour l’implémenteur

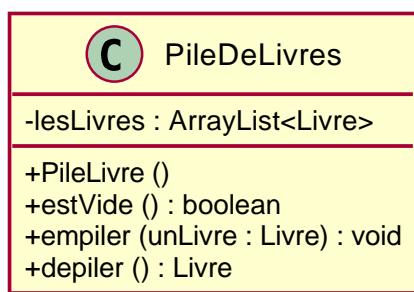
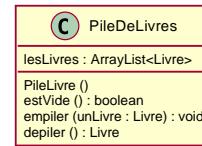


FIGURE 1.18 – pour tout le monde

Néanmoins, pour mettre en œuvre ses trois méthodes et son constructeur il est nécessaire de disposer d’un certain nombre d’outils (des méthodes, des attributs voire d’autres constructeurs plus spécifiques). Dans le cas simple de la *PileDeLivres* on aurait par exemple besoin d’une structure de données dynamique (une *ArrayList* de *Livre* par exemple) pour stocker les différents *Livre*. Ainsi, du point de vue de l’implémenteur de la classe, *PileDeLivres* est vue telle que figurée dans la figure 1.17.

Il serait donc intéressant de pouvoir rendre plus ou moins visible les attributs, méthodes et constructeurs d’une classe. Pour la classe *PileDeLivres* on pourrait rendre les trois méthodes et le constructeur publiques et visibles de l’extérieur, tandis qu’on pourrait restreindre l’accès à la variable *lesLivres* aux *PileDeLivres* seules. Tout comme les modificateurs de visibilité d’UML nous permettent d’indiquer cette information sur le diagramme de classe avec les caractères '+', '-' et '#' comme le montre la figure 1.18 (voir la section D.1 dans l’annexe), Java propose trois mots-clés permettant d’indiquer la visibilité des différents éléments d’un programme Java : les attributs, les méthodes et les constructeurs mais aussi les classes.

Ainsi, la classe *PileDeLivres* décrite par le diagramme de classe 1.18 serait représentée par le code Java de l’exemple 1.2.

---

**Exemple 1.7** Une PileDeLivre en Java

---

Listing 1.2 – Une pile de livre

```
import java.util.ArrayList;
/**
 * La classe PileDeLivres implémente une pile pouvant
 * recevoir des instances de la classe Livre.
 * La classe propose les fonctionnalités standard
 * d'une pile, permettant de :
 * <ol>
 *   <li>créer une pile
 *   <li>savoir si une pile est vide
 *   <li>empiler un nouveau livre au sommet de la pile
 *   <li>dépiler le livre en sommet de pile
 * </ol>
 * @author Olivier Camp
 * @version 1.0
 */
public class PileDeLivres {
    /*
     * La structure dynamique pour stocker l'empilement
     * de Livres
     * C'est une ArrayList de Livre
     * Cet attribut ne sert qu'à l'implémentation
     * il doit être caché. Il est donc "private"
     */
    private ArrayList<Livre> lesLivres;

    /*
     * Créer une PileDeLivre revient à
     * créer la structure qui
     * contiendra les Livres
     */
    public PileDeLivres (){
        ...
    }

    /*
     * La PileDeLivre est vide si la structure qui
     * contient les Livre à une taille nulle
     * Cette méthode doit être visible de tous.
     * Elle est donc "public"
     * @return un booléen indiquant si la pile est vide
     * ou non
     */
    public boolean estVide(){
        ...
    }
}
```

```

/*
 * Pour empiler un livre il suffit de l'ajouter à
 * la structure de stockage de l'empilement
 * Cette méthode doit être visible de tous
 * Elle est donc "public"
 * @param le livre qu'on souhaite empiler
 */
public void empiler(Livre unLivre){
    ...
}

/*
 * Si la pile n'est pas vide il suffit
 * de supprimer le dernier Livre
 * et de le retourner pour le dépiler
 * Si la pile est vide la méthode provoquera
 * une erreur.
 * Cette méthode doit être visible de tous
 * Elle est donc "public"
 * @return le livre qui se trouvait au sommet
 *         de la pile
 */
public unLivre depiler(){
    ...
}
}

```

### 1.7.2 Les modificateurs de visibilité de Java

Les modificateurs de visibilité de Java sont au nombre de trois : *public*, *private* et *protected*. Chacun d'eux peut être utilisé devant la déclaration d'une méthode, d'un constructeur ou d'un attribut. Le modificateur *public* peut également être utilisé devant une déclaration de classe. La sémantique de ces mots-clefs peut être différente selon l'élément devant lequel ils se trouvent.

#### 1.7.2.1 Absence de modificateur

**Pour une classe** Si la déclaration d'une classe n'est précédée d'aucun modificateur (c'est le cas de toutes les classes que nous avons vues jusqu'à présent) elle a la *visibilité par défaut*; c'est à dire qu'elle n'est visible que par les autres classes de son paquetage.

**Pour une méthode, un constructeur ou un attribut** Si la déclaration d'une méthode, d'un constructeur ou d'un attribut n'est précédée d'aucun modificateur de visibilité ils ont une *visibilité par défaut* c'est-à-dire qu'ils ne sont visibles que par les classes du paquetage de la classe dans laquelle ils sont déclarés.

### 1.7.2.2 Le modificateur *public*

**Pour une classe** Si la déclaration d'une classe est précédée du modificateur *public*, la classe est alors visible et peut être utilisée depuis n'importe quel paquetage.

```
public class MaClasse {
    // MaClasse est visible et utilisable depuis
    // n'importe où
    ...
}
```

**Pour une méthode, un constructeur ou un attribut** Si la déclaration d'une méthode ou d'un attribut d'une classe A est précédée du modificateur *public*, alors la méthode, le constructeur ou l'attribut est utilisable depuis toute classe qui se trouve dans le même paquetage que la classe A.

```
class MaClasseDefault {
    /*
     * MaClasseDefault n'est visible que depuis les
     * autres classes du paquetage
     * Elle a une visibilité "default"
     */

    public int unAttributPublic;
    /*
     * unAttributPublic est "public" et déclaré dans
     * une classe "default"
     * unAttributPublic n'est visible que depuis
     * les classes du même paquetage
     */
    ...

    public int uneMethodePublic(){
        /*
         * uneMethodePublic est "public" et déclarée dans
         * une classe "default"
         * uneMethodePublic n'est visible que depuis
         * les classes du même paquetage
         */
        ...
    }
}
```

Dans le cas où la déclaration de la classe A est elle aussi précédée du modificateur *public* alors la méthode ou l'attribut sera visible et utilisable depuis n'importe quel paquetage.

```
public class MaClassePublic {  
    /*  
     * MaClassePublic est visible et utilisable  
     * depuis n'importe où  
     * Elle a une visibilité "public"  
     */  
  
    public int autreAttributPublic;  
    /*  
     * autreAttributPublic est "public" et déclarée  
     * dans une classe "public"  
     * autreAttributPublic est donc visible  
     * depuis n'importe où  
     */  
    ...  
  
    public int autreMethodePublic(){  
        /*  
         * autreMethodePublic est "public" et déclaré  
         * dans une classe "public"  
         * autreMethodePublic est donc visible  
         * depuis n'importe où.  
         */  
        ...  
    }  
}
```

### 1.7.2.3 Le modificateur *private*

**Pour une méthode, un constructeur ou un attribut** Si la déclaration d'une méthode, d'un constructeur ou d'un attribut d'une classe *A* est précédée du modificateur *private*, alors la méthode ou l'attribut n'est utilisable que dans la classe *A* quelque soit la visibilité de la classe *A*.

```
public class MaClasse {  
  
    private int unAttributPrivate;  
    /*  
     * unAttributPrivate est "private",  
     * il n'est donc visible que dans la classe MaClasse  
     */  
    ...  
  
    private int uneMethodePrivate(){  
        /*  
         * uneMethodePrivate est "private"  
         * elle n'est donc visible et utilisable que  
         * dans MaClasse.  
         */  
        ...  
    }  
}
```

#### 1.7.2.4 Le modificateur *protected*

**Pour une méthode, un constructeur ou un attribut** Si la déclaration d'une méthode, d'un constructeur ou d'un attribut d'une classe *A* est précédée du modificateur *protected*, alors la méthode, le constructeur ou l'attribut est utilisable depuis toute classe qui se trouve dans le même paquetage que la classe *A*.

```
class MaClasseDefault {  
    /*  
     * MaClasseDefault n'est visible que depuis  
     * les autres classes du paquetage  
     * Elle a une visibilité "default"  
     */  
  
    protected int unAttributProtected;  
    /*  
     * unAttributProtected est "protected"  
     * et déclaré dans une classe "default"  
     * unAttributProtected n'est visible que  
     * depuis les classes du paquetage  
     */  
    ...  
  
    protected int uneMethodeProtected(){  
        /*  
         * uneMethodeProtected est "protected"  
         * et déclarée dans une classe "default"  
         * uneMethodeProtected n'est visible  
         * que depuis les classes du paquetage  
         */  
        ...  
    }  
}
```

Dans le cas où la déclaration de la classe A est précédée du modificateur *public* alors la méthode ou l'attribut sera visible et utilisable depuis les classes du paquetages, mais aussi depuis toutes les classes qui héritent de A.

```

public class MaClassePublic {
    // MaClassePublic est visible depuis n'importe où
    // Elle a une visibilité "public"

    protected int unAttributProtected;
    /*
     * unAttributProtected est "protected" et déclaré
     * dans une classe "public"
     * unAttributProtected est visible
     * depuis les classes du paquetage et
     * depuis les sous classes de MaClassePublic
     */
    ...

    protected int uneMethodeProtected(){
        /*
         * uneMethodeProtected est "protected" et déclarée
         * dans une classe "public"
         * uneMethodeProtected n'est visible que
         * depuis les classes du paquetage et
         * depuis les sous classes de MaClassePublic
         */
        ...
    }
}

```

**En résumé** Le tableau 1.3 présente une vue globale des effets des différents modificateurs de visibilité (*public*, *protected*, *private*) ou de leur absence lorsqu'ils sont appliqués aux différents éléments d'un programme Java (classe, attribut, méthode ou constructeur).

	classes	attributs/méthodes/constructeurs	
		d'une classe publique	d'une classe default
<i>aucun modificateur</i>	paquetage	paquetage	paquetage
<i>public</i>	totale	totale	paquetage
<i>protected</i>		paquetage, sous classes	paquetage
<i>private</i>		classe	classe

TABLE 1.3 – Visibilités associées aux modificateurs

### 1.7.3 Des bonnes pratiques

Il est recommandé lorsqu'on écrit une classe de cacher son implémentation (avec le modificateur *private*) et de n'exposer (avec le modificateur *public*) que les services qu'elle doit rendre ; c'est-à-dire les méthodes que l'on souhaite rendre accessibles aux utilisateurs de la classe. Ainsi, une différence claire est faite entre ce qui est utilisable pour l'implémentation de la classe (toutes les méthodes) et ce qui est accessible lorsqu'on utilise la classe. Si, pour son bon fonctionnement, il est nécessaire que la classe propose

un accès en lecture ou en écriture à ses attributs elle le fera en exposant ses accesseurs et/ou ses mutateurs.

Ce principe est très bien illustré par la voiture : elle propose cinq fonctionnalités (des *méthodes*) à son utilisateur (le conducteur) à travers le volant, les pédales de frein, d'embrayage et d'accélérateur et à travers le levier de changement de vitesse. En revanche, l'implémentation permettant de mettre en œuvre ces fonctionnalités est cachée sous le capot de la voiture : c'est le domaine de l'implémenteur de la voiture (le mécanicien).

Une bonne pratique consiste à donner la visibilité minimum à chaque élément, on pourra toujours donner une visibilité plus grande si le besoin s'en fait sentir.

#### 1.7.4 Quelques explications sur la méthode *main*

Comme nous l'avons vu plus tôt, la méthode principale d'une classe doit avoir la signature suivante :

```
public static void main(String [] args)
```

Nous savons maintenant que :

- la méthode *main* étant *public* elle est donc visible depuis tout paquetage ;
- la méthode *main* étant *static* c'est une méthode de classe.

Nous sommes donc maintenant en mesure d'expliquer ce qui se passe lorsqu'une classe principale est exécutée par la machine virtuelle Java.

Supposons que la classe *UneClasse* soit une classe exécutable, elle définit donc une méthode de classe *main*. On exécutera cette classe en la soumettant à la machine virtuelle et on pourra lui transmettre des arguments en ligne de commande de la façon suivante :

```
java UneClasse argument1 argument2
```

L'exécution de la classe se déroulera alors de la façon suivante, telle qu'illustrée par la figure 1.19 :

1. lancement de la machine virtuelle Java
2. chargement de la classe *MaClasse* depuis le fichier *MaClasse.class*
3. construction d'un tableau contenant les arguments *arg1* et *arg2*
4. passage du tableau en paramètre de la méthode de classe *main*
5. envoi du message *main({arg1,arg2})* à la classe *MaClasse*.

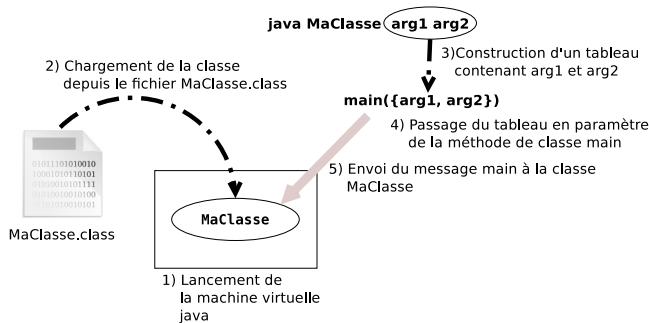


FIGURE 1.19 – Exécution d'une classe

### 1.7.5 Quelques exercices

Les exercices suivants ont pour objectifs :

1. de mieux comprendre l'utilisation des modificateurs de visibilité,
2. d'expérimenter l'encapsulation sur l'exemple simple de pile vu ci-dessus,
3. d'expérimenter l'encapsulation sur un exemple plus complexe traitant des nombres ... complexes.

#### Exercice 1 :

1. Remettre en état de fonctionnement les classes des paquetages *fr.eseo.pdlo.\**\* en associant les bons modificateurs de visibilité aux classes et à leurs méthodes, constructeurs et attributs.

Vérifier que les modifications effectuées permettent à tous les fichiers source d'être compilés sans erreur et aux classes résultantes de s'exécuter correctement.

#### Exercice 2 :

1. Un *Triangle* est composé de 3 côtés et possède un type<sup>4</sup> défini par une énumération (*enum*).
2. Écrire l'énumération *fr.eseo.pdlo.exercices.triangle.TriangleType* qui permet de définir si un triangle est soit de type *scalène*, soit *isocèle*, soit *équilatéral* ou n'est pas un triangle par impossibilité des longueurs renseignées pour les 3 côtés (exemple : (1,1,5)). Complétez le code suivant :

```
package ...

public enum TriangleType {
    SCALENE ("Scalène"),
    ISOCELE ("Isocèle"),
    EQUILATERAL ("Équilatéral"),
    NON_TRIANGLE ("Non triangle");

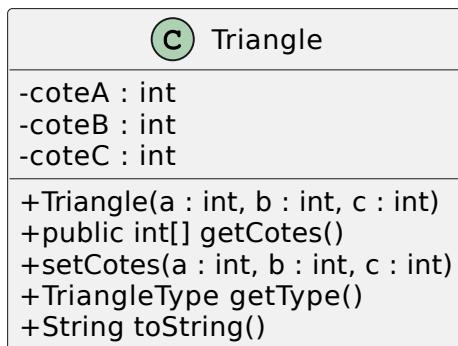
    private String typeTriangle ...

    private TriangleType (String typeTriangle) {
        ...
    }

    @Override
    public String toString () {
        ...
    }
}
```

3. Écrire la classe *fr.eseo.pdlo.exercices.triangle.Triangle* définie par le diagramme UML de la figure 1.20. Notez bien que la méthode *getType()* renvoie bien une valeur de l'énumération. La méthode *toString()* renvoie un message de type "Triangle : Isocèle" si le triangle est isocèle ou "Ceci n'est pas un triangle." si le triangle ne peut pas être un triangle avec les longueurs renseignées pour chaque côté.

4. [https://fr.wikipedia.org/wiki/Triangle#Cas\\_particuliers](https://fr.wikipedia.org/wiki/Triangle#Cas_particuliers)

FIGURE 1.20 – La classe *Triangle*

## 4. Écrire la classe exécutable

`fr.eseo.pdlo.essais.exercices.triangle.triangleEssai` qui permet d'avoir un aperçu du bon fonctionnement de la classe *Triangle*.

**Exercice 3 :**

1. En vous inspirant de la classe *PileDeLivres* vue dans la section 1.7 écrire la classe `fr.eseo.pdlo.exercices.pile.PileDeRectangles`. Une *PileDeRectangles* pouvant contenir un nombre quelconque de *Rectangle* on utilisera un tableau dynamique (par exemple de type `ArrayList<Rectangle>` pour les stocker (voir la section 1.4.1.8 pour des explications sur l'instanciation de ce type de classes).

## 2. Écrire la classe exécutable

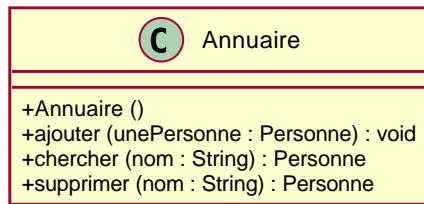
`fr.eseo.pdlo.essais.exercices.pile.PileDeRectanglesEssai` qui permet d'avoir un aperçu du bon fonctionnement de la classe *PileDeRectangles*.

Dans sa méthode principale, la classe d'essai devra créer et empiler 5 rectangles, puis les dépiler et afficher leurs caractéristiques à l'écran.

Afin de “reconnaître” les rectangles affichés on aura pris soin de leurs donner des positions et des dimensions reconnaissables. Par exemple, le premier rectangle empilé aura comme position le point (1,1) et pour dimensions (100x10) tandis que le cinquième rectangle créé sera positionné au point (5,5) et aura les dimensions (500,50).

**Exercice 4 :**

1. Écrire la classe `fr.eseo.pdlo.exercices.utilisateurs.Annuaire` permettant de stocker des *Personne* et d'y accéder par leurs noms. La classe devra proposer les méthodes publiques décrites dans le diagramme UML de la figure 1.21.

FIGURE 1.21 – La classe *Annuaire*

2. Écrire la classe *fr.eseo.pdlo.essais.exercice.utilisateurs.AnnuaireEssai* permettant d'avoir un aperçu du bon fonctionnement de la classe *Annuaire*. La classe d'essai devra :
- créer un *Annuaire*,
  - ajouter 5 *Personne*,
  - supprimer 1 *Personne*,
  - ajouter 1 *Personne*,
  - rechercher 1 *Personne* (présente dans l'annuaire) et afficher son identité,
  - rechercher 1 *Personne* (absente de l'annuaire) et afficher son identité.

**Exercice 5 :**

1. Écrire la classe *fr.eseo.pdlo.exercices.complexe.cartesien.Complexe* pour représenter les nombres complexes en Java. Dans un premier temps on considérera qu'un complexe est formé d'une *partie réelle* et d'une *partie imaginaire*.

On souhaiterait que les objets de la classe *Complexe* :

- puissent retourner leurs parties réelles et imaginaires ;
- puissent changer leurs parties réelles et imaginaires ;
- puissent retourner leurs modules et arguments ;
- puissent changer leurs modules (en maintenant leurs arguments constants) ;
- puissent changer leurs arguments (en maintenant leurs modules constants) ;
- puissent s'additionner à d'autres nombres complexes ;
- puissent se multiplier avec d'autres nombres complexes ;
- puisse retourner la chaîne de caractères

$$a + bi$$

correspondant à sa forme algébrique

- puissent afficher des informations les concernant (forme algébrique, parties imaginaires et réelles, modules et arguments).

La classe *Complexe* à écrire est représentée par le diagramme UML de la figure 1.22.

FIGURE 1.22 – La classe *Complexe*

On supposera que le constructeur par défaut de la classe retourne un complexe dont les parties réelle et imaginaire sont nulles.

## 2. Écrire la classe

*fr.eseo.pdlo.essais.exercices.complexe.ComplexeCartesienEssai* qui permet de vérifier que la classe *Complexe* du paquetage *fr.eseo.pdlo.exercices.complexe.ComplexeCartesienEssai* fonctionne correctement.

3. Modifier le constructeur de la classe *Complexe* pour qu'il permette aussi de créer un *Complexe* dont on ne connaît que le module et l'argument (sa représentation polaire). Pour pouvoir différencier les deux cas polaire ou cartésien, il faut rajouter un argument booléen au constructeur. Lorsqu'il aura pour valeur *true* on considérera que les deux autres arguments du constructeur sont le module et l'argument du complexe à créer. Lorsque sa valeur sera *false* les deux autres arguments du constructeur seront les parties réelle et imaginaire du nombre complexe à créer. Le nouveau constructeur aura donc pour signature :

```

/*
 * Si estPolaire==true
 * arg1 est le module et arg2 est l'argument
 * du complexe à créer
 * sinon
 * arg1 est la partie réelle et
 * arg2 est la partie imaginaire
 * du complexe à créer
 */
public Complexe(boolean estPolaire, double arg1, double arg2)

```

On n'oubliera pas de rajouter un argument aux appels de constructeur pouvant exister dans la classe

*fr.eseo.pdlo.exercices.complexe.cartesien.Complexe*.

4.Modifier la classe *fr.eseo.pdlo.essais.exercices.complexe.ComplexeCartesienEssai* pour qu'elle tienne compte de ce nouveau constructeur.

5. Écrire la classe `fr.eseo.pdlo.exercices.complexe.polaire.Complexe` qui représente les nombres complexes en représentation polaire (avec leurs modules et leurs arguments). Le diagramme de classe UML correspondant à cette nouvelle classe *Complexe* sera celui de la figure 1.23. On constate que les méthodes publiques de cette classe sont les mêmes que celles proposées par la classe représentée dans la figure 1.22. La seule différence réside dans l'implémentation des deux classes et dans le comportement de la méthode *affiche()* qui dans le cas d'un complexe en représentation polaire l'affichera sous la forme : *module exp(i argument)*.

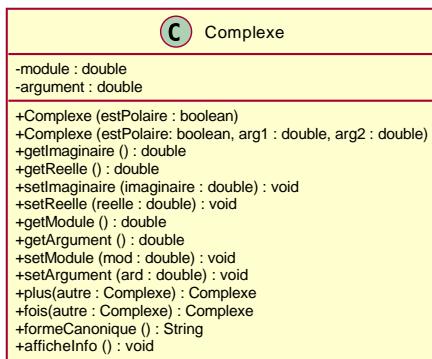


FIGURE 1.23 – Une autre version de *Complexe*

6. Écrire la classe  
`fr.eseo.pdlo.essais.exercices.complexe.ComplexePolaireEssai` qui permet de vérifier que la classe *Complexe* du paquetage  
`fr.eseo.pdlo.exercices.complexe.polaire` fonctionne correctement.
7. Écrire la classe `fr.eseo.pdlo.essais.exercices.complexe.ComplexeEssai` qui permet de vérifier que les deux classes *Complexe* écrites ci-dessus ont les mêmes comportements. On sera amené à utiliser deux classes de mêmes noms courts dans la classe  
`fr.eseo.pdlo.essais.exercices.complexe.ComplexeEssai`.

## 1.8 L'héritage

L'héritage étant un des concepts fondamentaux de la programmation orientée objets, il est naturellement proposé par Java et permet de déclarer qu'une classe est sous-classe d'une autre classe.

La relation d'héritage est spécifiée lors de la définition d'une classe. Néanmoins, comme nous avons pu le voir dans les exemples des pages précédentes, il n'est pas toujours nécessaire de spécifier de relation d'héritage. En effet, les classes que nous avons déclarées précédemment, n'en précisent aucune.

Cependant, le fait qu'aucun héritage ne soit déclaré, ne signifie pas que la classe n'hérite d'aucune autre classe. Au contraire, elle indique que la classe hérite, par défaut, d'une classe particulière : la classe *Object* du paquetage `java.lang`, racine de l'arbre d'héritage de toutes les classes Java.

Ainsi, les classes *Personne*, *UserId*, *Rectangle*, *Complexe*, etc. ainsi que les différentes classes qui nous ont permis de vérifier le bon fonctionnement de ces classes

(*PersonneEssai*, *RectangleEssai*, etc.) héritent toutes de la classe *java.lang.Object* comme le montre le diagramme de la figure 1.24 :

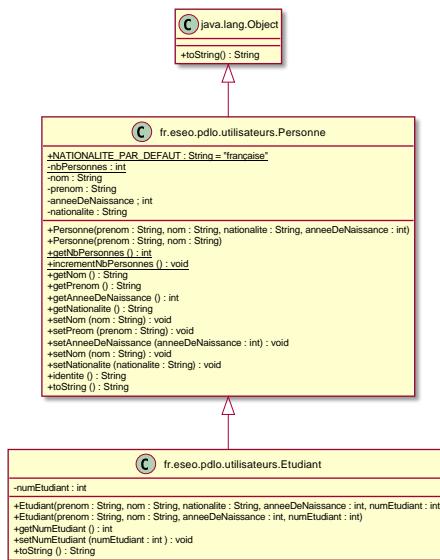


FIGURE 1.24 – La classe *Etudiant* et sa relation d'héritage

Si toutefois on souhaite définir une relation d'héritage autre que l'héritage par défaut, Java propose un mot-clef permettant de le faire : le mot-clef *extends*. Lorsqu'il est utilisé dans la définition d'une classe, ce mot-clef permet de spécifier qu'elle hérite d'une autre classe.

Contrairement à d'autres langages orientés objets, **Java ne propose pas l'héritage multiple** de classes. En Java, une classe ne peut hériter que d'une seule sur-classe.

Par exemple, la définition d'une classe *Etudiant*, sous-classe de la classe *Personne*, se fera de la façon suivante :

```

package fr.eseo.pdlo.exercices.utilisateur;
/**
 * Un Etudiant est une Personne
 * La classe Etudiant hérite donc
 * de la classe Personne
 * @author Olivier Camp
 * @version 1.0
 */
public class Etudiant extends Personne {
    /**
     * Un Etudiant est une Personne
     * qui en plus des variables de
     * la classe Personne possède
     * une variable supplémentaire
     * précisant son numéro d'étudiant
     */
    private int numEtudiant;

    /**
     * La classe Etudiant propose
     * un accesseur pour la variable
     * numEtudiant.
     * @return la valeur courante de
     *         la variable
     */
    public int getNumEtudiant(){
        return numEtudiant;
    }
    /**
     * La classe Etudiant propose
     * un mutateur pour la variable
     * numEtudiant.
     * @param la nouvelle valeur
     *         souhaitée pour numEtudiant
     */
    public void setNumEtudiant(int num){
        numEtudiant=num;
    }
}

```

Lorsqu'une **sous-classe** hérite d'une **sur-classe**, les instances de la **sous-classe** peuvent :

- accéder aux variables d'instance de visibilité *public* et *protected* de la **sur-classe**,
- exécuter les méthodes d'instance de visibilité *public* et *protected* de la **sur-classe**.

Cette seconde propriété s'explique par le mécanisme de recherche de méthode décrit dans la section 1.8.1.

### 1.8.1 Héritage, envoi de messages et recherche de méthode

La réaction d'un objet à un envoi de message est très directement lié à l'arbre d'héritage dans lequel se trouve sa classe. C'est effectivement en remontant l'arbre d'héritage que la méthode correspondante sera recherchée.

Par exemple, si on considère une instance *leMajor* de la classe *Etudiant* représentée dans la figure 1.24 et l'envoi de message suivant :

```
leMajor.setNom("Turing");
```

La recherche de la méthode se fera de la façon suivante :

1. *leMajor* est une instance de la classe *Etudiant*, la méthode *setNom(...)* est tout d'abord recherchée dans cette classe,
2. la méthode *setNom(...)* n'est pas définie dans la classe *Etudiant*, elle est donc recherchée dans sa sur-classe : la classe *Personne*. Elle s'y trouve, c'est donc **la méthode de la classe Personne qui sera exécutée**.

Considérons maintenant l'envoi de message suivant :

```
leMajor.affiche();
```

Comme pour l'exemple précédent :

- La méthode *affiche()* est recherchée dans la classe *Etudiant* et n'y est pas trouvée,
- elle est ensuite recherchée dans la sur-classe (la classe *Personne*) sans succès,
- elle est enfin recherchée dans la classe *Object*, mais en vain (il suffit de consulter la documentation de la classe *java.lang.Object* pour s'en rendre compte). La classe *Object* étant la racine de l'arbre d'héritage et n'ayant pas de sur-classe, la recherche s'arrête et **une erreur est provoquée**.

#### Recherche de méthode :

Lorsqu'un message est reçu par un objet :

1. le receveur recherche la méthode à appliquer dans sa classe,
2. si elle ne s'y trouve pas il remonte l'arbre d'héritage,
3. si il ne la trouve pas dans sa remontée de l'arbre d'héritage, une erreur est déclenchée.

### 1.8.2 Redéfinition (overriding) de méthodes

Lorsqu'une classe propose une nouvelle définition pour une méthode déjà définie dans une de ses classes ancêtres on parle de **redéfinition de méthode**. En anglais on parle de "**method overriding**". C'est le cas pour la méthode *toString()* des classes *java.lang.Object*,

*fr.eseo.pdlo.exercices.utilisateurs.Personne* et *fr.eseo.pdlo.exercices.utilisateurs.Etudiant* décrites dans le diagramme de classe de la figure 1.24.

### 1.8.3 La méthode *toString()* de la classe *java.lang.Object*

La classe *java.lang.Object* définit une dizaine de méthodes. Celles-ci sont donc connues des instances de n'importe quelle classe. En effet, lorsqu'un message portant le nom d'une de ces méthodes est envoyé à une instance quelconque, l'instance la

trouvera nécessairement dans son arbre d'héritage : au pire à sa racine dans la classe `java.lang.Object`.

La méthode `toString()` est l'une des méthodes de la classe `java.lang.Object`. Elle est prévue pour retourner une représentation textuelle d'un objet quelconque qui pourra donc être affichée sur un terminal.

Par défaut, la méthode `toString()` lorsqu'elle est appliquée à un objet quelconque retourne une chaîne de caractères de la forme `NomDeClasse@idObjet`.

```
/*
 * Création d'une Personne
 * et affectation de la variable p
 * avec cette instance
 */
Personne p = new Personne("Turing", "Alan", 1912);
/*
 * Affichage de la représentation textuelle
 * de l'objet p
 */
System.out.println(p.toString());
```

affichera le message suivant :

```
Personne@2a139a55
```

Dans le cas de la classe `Personne` si on souhaite que la méthode `toString()` retourne la chaîne de caractères calculée par la méthode `identite()` il suffit de la définir dans la classe `Personne` comme étant :

```
public String toString(){
    return identite();
}
```

### 1.8.3.1 Appel automatique de la méthode `toString()`

La méthode `toString()` est, en fait, appelée automatiquement lorsqu'on souhaite afficher l'instance d'un objet et les deux demandes d'affichage suivantes ont rigoureusement le même effet :

```
System.out.println(p);
```

et :

```
System.out.println(p.toString());
```

Si on souhaite personnaliser l'affichage des instances d'une classe particulière, il suffit donc de redéfinir la méthode `toString()` de la classe afin qu'elle retourne la chaîne qui correspond au message que l'on souhaite afficher. Par exemple, une fois la méthode `toString()` de la classe `Personne` redéfinie comme ci-dessus, l'affichage d'une instance de `Personne` par un "`System.out.print`" provoquera l'affichage de l'identité de l'instance de `Personne` concernée. Ainsi, le fragment de code :

```
Personne p=new Personne("Malcolm", "X",
                         "américaine", 1925);
System.out.println(p);
```

Résultera en un appel automatique à la méthode *toString()* et affichera le message suivant :

```
Malcolm X - Né(e) : 1925 Nationalité : américaine
```

#### 1.8.4 L'objet *super*

Lorsqu'une instance reçoit un message il est parfois nécessaire de commencer la recherche de méthode à partir de la sur-classe, plutôt que de la classe de l'objet lui-même. Le langage Java propose un moyen pour obtenir ce résultat : l'objet spécial *super*.

Dans le corps d'une méthode ou d'un constructeur, l'objet “spécial” *this* représente l'objet en train d'exécuter la méthode, ou celui en train d'être construit par le constructeur. Cet objet est considéré comme appartenant à la classe qui définit la méthode ou le constructeur en question.

Tout comme l'objet *this*, l'objet *super* représente, lui aussi, l'objet en train d'exécuter la méthode. Cependant, contrairement à *this* il n'appartient pas à la classe dans laquelle est définie la méthode en question, mais à sa sur-classe. Par exemple, s'il est fait référence à *super* dans une méthode de la classe *Etudiant*, cet objet n'est pas perçu comme étant une instance de cette classe, mais plutôt de sa sur-classe : la classe *Personne*.

##### 1.8.4.1 Exemple d'utilisation de l'objet *super*

Supposons qu'on souhaite qu'une instance de la classe *Etudiant* puisse s'afficher sur deux lignes sous la forme suivante :

```
Alan Turing - Né(e) : 1912 Nationalité : britannique
Numéro d'étudiant : 1234
```

Nous avons vu dans la section 1.8.3 que ceci pouvait se faire en donnant une définition adéquate à la méthode *toString()* de la classe *Etudiant*. En regardant attentivement le format d'affichage souhaité pour un *Etudiant* on peut remarquer qu'il est très semblable à celui de la classe *Personne*. En fait, la représentation textuelle d'une instance d'*Etudiant* est la même que celle d'une instance de *Personne*, sauf qu'elle est suivie d'une indication du numéro de l'étudiant.

On pourrait donc se contenter de recopier la définition de la classe *toString()* de *Personne* dans la classe *Etudiant* et d'y faire l'ajout nécessaire, comme suit :

```

/*
 * Une première définition de la méthode
 * toString() pour la classe Etudiant.
 * On peut faire mieux !!
 */
public String toString(){
    return (identite()
        + "\n" // retour à la ligne
        + "Numéro d'étudiant:"+
        + getNumEtudiant());
}

```

Cette définition produirait l'effet escompté, mais elle ne fait pas apparaître clairement la relation qui existe entre la représentation textuelle d'un *Etudiant* et celle d'une *Personne*.

En effet, il serait intéressant que le fait que la représentation textuelle d'un *Etudiant* est celle d'une *Personne* à laquelle est ajoutée l'information concernant son numéro d'étudiant puisse transparaître dans le code Java de la méthode.

Ceci est très simplement réalisé à l'aide du mot-clef *super*, de la façon suivante :

```

/*
 * Une meilleure définition de la méthode
 * toString() pour la classe Etudiant
 */
public String toString(){
    // Pour calculer la représentation textuelle
    // d'un Etudiant,
    // on commence par calculer sa représentation
    // textuelle en le considérant comme une instance
    // de sa sur-classe (la classe Personne)
    return (super.toString()
        + "\n" // on ajoute un retour à la ligne
        // Puis les informations pour le
        // numéro d'étudiant.
        + "Numéro d'étudiant:"+
        + getNumEtudiant());
}

```

Cette deuxième définition fait clairement apparaître le fait que le résultat de la méthode *toString()* de la classe *Etudiant*, dépend de celle de la méthode *toString()* de sa sur-classe.

D'autre part, si la méthode *toString()* de la classe *Personne* venait à être modifiée, celle de la classe *Etudiant* subirait les mêmes changements par héritage.

La section 1.8.5 donne un autre exemple d'utilisation du mot-clef *super* pour accéder à des variables d'instance masquées.

### 1.8.5 Masquage de variables d'instance

Lorsqu'une classe définit une variable d'instance de même nom que celle d'une (ou de plusieurs) de ses classes ancêtres, alors la variable définie dans la classe se trouvant le plus bas dans l'arbre d'héritage masque toutes les autres variables définies par des classes ancêtres. Considérons par exemple le diagramme ci-dessous :

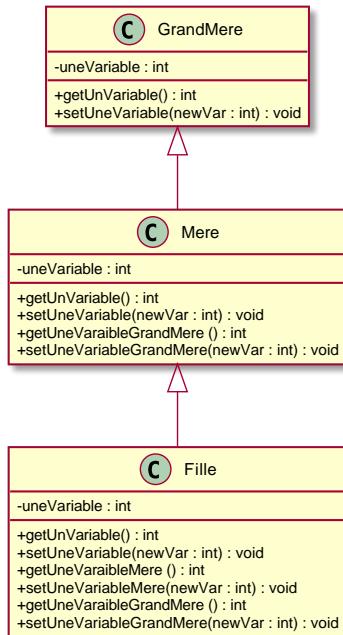


FIGURE 1.25 – Masquage de variables

Compte tenu de ce qui vient d'être vu dans la section 1.8.2 on pourrait penser que la redéfinition des accesseurs et mutateurs de la classe *GrandMère* n'est pas nécessaire. En effet, les accesseurs et mutateurs de la classe *GrandMère* sont hérités par les sous-classes *Mère* et *Fille*.

Cependant, chacune des trois classes (*Fille*, *Mère* et *GrandMère*) définit une variable d'instance de type *int* et de nom *uneVariable*. Ainsi :

- les instances de la classe *GrandMère*, telles que celle représentée dans la figure 1.26, possèdent **une** variable d'instance de type *int* et de nom *uneVariable* :
- celle définie dans la classe *GrandMère*. Les instances de la classe *GrandMère* peuvent accéder à cette variable directement par son nom ;



FIGURE 1.26 – Une instance de la classe GrandMère

- les instances de la classe *Mère*, telles que celle représentée dans la figure 1.27, possèdent **deux** variables d'instance de type *int* et de nom *uneVariable* :
- celle définie dans la classe elle-même (la classe *Mère*). Les instances de la classe *GrandMère* peuvent accéder à cette variable par son nom ;

- et celle héritée de la sur-classe (la classe *GrandMere*). Les instances de la classe *Mere* ne peuvent pas accéder, par son nom, à cette variable de la classe *GrandMere* ;

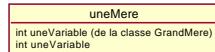


FIGURE 1.27 – Une instance de la classe Mere

- les instances de la classe *Fille*, telles que celle représentée dans la figure 1.28, possèdent **trois** variables d'instance de type *int* et de nom *uneVariable* :
  - celle définie dans la classe elle-même (la classe *Fille*),
  - celle héritée de la sur-classes (la classe *Mere*),
  - et enfin celle héritée de la sur-sur-classe (la classe *GrandMere*).

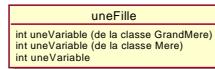


FIGURE 1.28 – Une instance de la classe Fille

De même, chacune des trois classes définit une paire accesseur/mutateur permettant de lire et d'écrire la valeur de **leur** variable *uneVariable*.

En outre, les classes *Mere* et *Fille* définissent des méthodes permettant d'accéder aux variable *uneVariable* héritées de leurs sur-classes les méthodes *get/setUneVariable-Mere(...)* et *get/setUneVariableGrandMere(...)*.

Les classes représentées dans la figure 1.25 peuvent être implémentées en Java par les 3 classes ci-dessous :

1. la classe *GrandMere* définit une variable entière *uneVariable*, son accesseur et son mutateur.
  - Ses instances accèdent à la variable d'instance, au choix, simplement par son nom ou en utilisant “l’objet spécial” *this* :

```


    /**
     * La classe GrandMere possède une variable d'instance:
     * la variable uneVariable
     */
    public class GrandMere {
        /**
         * La variable uneVariable de la
         * classe GrandMere est un entier
         */
        public int uneVariable ;

        /**
         * @return La valeur de la variable uneVariable
         *         définie dans la classe GrandMere
         */
        public int getUneVariable(){
            // On peut accéder à la variable
            // par son nom.
            return uneVariable;
        }

        /**
         * Permet de fixer la valeur de la variable
         * uneVariable définie dans la classe GrandMere
         */
        public void setUneVariable(int newVariable){
            // On peut accéder à la variable
            // avec "this"
            this.uneVariable=newVariable;
        }
    }


```

2. la classe *Mere* définit une variable entière *uneVariable*, son accesseur et son mutateur et hérite de *GrandMere*. Elle propose deux méthodes permettant d'écrire et de lire la variable *uneVariable* définie dans la classe *GrandMere*.
  - Les instances de *Mere* accèdent à la variable d'instance *uneVariable* définie dans la classe directement par son nom ou à l'aide de “l'objet spécial” *this*;
  - elles doivent utiliser “l'objet spécial” *super* pour accéder à la variable d'instance de la sur-classe *GrandMere*, par son nom ou à travers un accesseur-/mutateur.

```


/**
 * La classe Mere hérite de GrandMere.
 * Elle possède une variable d'instance
 * propre : la variable uneVariable.
 * Elle hérite d'une autre variable de même nom
 * de sa sur-classe : la classe Mere
 */
public class Mere extends GrandMere {
    /**
     * La variable uneVariable de la
     * classe Mere est un entier
     */
    public int uneVariable ;

    /**
     * @return La valeur de la variable uneVariable
     *         définie dans la classe Mere
     */
    public int getUneVariable(){
        // On peut accéder à la variable
        // par son nom.
        return uneVariable;
    }
    /**
     * Permet de fixer la valeur de la variable
     * uneVariable définie dans la classe Mere
     */
    public void setUneVariable(int newVariable){
        // On peut accéder à la variable
        // avec "this"
        this.uneVariable=newVariable;
    }
    /**
     * @return La valeur de la variable uneVariable
     *         définie dans la classe GrandMere
     */
    public int getUneVariableGrandMere(){
        // On peut accéder à la variable de GrandMere
        // avec "super"
        return super.uneVariable;
    }
    /**
     * Permet de fixer la valeur de la variable
     * uneVariable définie dans la classe GrandMere
     * @arg La nouvelle valeur de la variable
     */
    public void setUneVariableGrandMere(int newVariable){
        // On peut utiliser le mutateur de GrandMere
        // avec "super"
        super.setUneVariable(newVariable);
    }
}


```

3. la classe *Fille* hérite de *Mere*. Elle définit, elle aussi, une variable entière *uneVariable*, son accesseur et son mutateur. Elle propose des méthodes permettant d'accéder en lecture et en écriture aux variables *uneVariable* définies dans les classes *Mere* et *GrandMere*.
  - Les instances de *Fille* accèdent à la variable d'instance *uneVariable* définie dans la classe directement par son nom (qu'elles peuvent éventuellement préfixer du mot clef *this*) ;
  - elles doivent utiliser “l’objet spécial” *super* pour accéder à la variable d’instance de la sur-classe *Mere*, par son nom ou à travers un accesseur/mutateur ;
  - elles doivent utiliser un transtypage explicite de “l’objet spécial” *this* pour accéder à la variable d’instance de la “sur-sur-classe” *GrandMere*, par son nom.

```


/**
 * La classe Fille possède une variable d'instance
 * propre : la variable uneVariable.
 * Elle hérite de deux autres variables de mêmes noms
 * - l'une de sa sur-classe directe : la classe Mere
 * - l'autre de la classe GrandMere
 */
public class Fille extends Mere {
    /**
     * La variable uneVariable de la
     * classe Fille est un entier
     */
    public int uneVariable ;

    /**
     * @return La valeur de la variable uneVariable
     *         définie dans la classe Fille
     */
    public int getUneVariable(){
        // On peut accéder à la variable
        // par son nom.
        return uneVariable;
    }
    /**
     * Permet de fixer la valeur de la variable
     * uneVariable définie dans la classe Fille
     */
    public void setUneVariable(int newVariable){
        // On peut accéder à la variable
        // avec "this"
        uneVariable=newVariable;
    }
    /**
     * @return La valeur de la variable uneVariable
     *         définie dans la classe Mere
     */
    public int getUneVariableMere(){
        // On doit accéder à la variable de Mere
        // avec "super"
        return super.getUneVariable();
    }
    /**
     * Permet de fixer la valeur de la variable
     * uneVariable définie dans la classe Mere
     * @arg La nouvelle valeur de la variable
     */
    public void setUneVariableMere(int newVariable){
        // On doit accéder à la variable de Mere
        // avec "super"
        super.uneVariable=newVariable;
    }
}


```

```

    /**
     * @return La valeur de la variable uneVariable
     *         définie dans la classe GrandMere
     */
    public int getUneVariableGrandMere(){
        // On accède à la variable de GrandMere
        // à l'aide d'un transtypage explicite.
        return ((GrandMere)this).uneVariable;
    }
    /**
     * Permet de fixer la valeur de la variable
     * uneVariable définie dans la classe GrandMere
     * @arg La nouvelle valeur de la variable
     */
    public void setUneVariableGrandMere(int newVariable){
        // On accède à la variable de GrandMere
        // à l'aide d'un transtypage explicite.
        ((GrandMere)this).uneVariable=newVariable;
    }
}

```

### 1.8.6 Héritage et polymorphisme

La relation d'héritage est parfois appelée la relation *est-un* (ou *is-a* en anglais). Cette dénomination se comprend aisément lorsqu'on regarde l'arbre d'héritage de la figure 1.24. En effet, un *Etudiant* **est-une** *Personne* et une *Personne* **est-un** *Object*.

Ainsi, une instance de la classe *Etudiant* est, bien sûr, un *Etudiant*. C'est aussi une *Personne*, ou encore un *Object*. L'instance peut donc être considérée comme étant un objet **polymorphe** appartenant à plusieurs classes.

Comme nous l'avons vu dans la section 1.8.1, le polymorphisme découlant de l'héritage, aura pour conséquence qu'une instance d'*Etudiant* peut exécuter les méthodes des classes *Etudiant* (bien sûr), mais aussi celles des classes *Personne* et *Object*. Le polymorphisme aura aussi pour conséquence qu'une instance de la classe *Etudiant* pourra être passée en argument à une méthode attendant un *Etudiant* (bien sûr), mais aussi à des méthodes attendant des *Personne* ou des *Object* en argument.

### 1.8.7 Héritage et constructeurs : le constructeur *super()*

Un des avantages de l'héritage est qu'il permet de factoriser le code commun à des classes différentes.

Par exemple, si on considère les classes *Etudiant* et *Personne* représentées dans le diagramme de la figure 1.24, les méthodes qui concernent les *Personne* telles que *get/setNom(...)*, *get/SetPrenom(...)*, *get/setAnneeDeNaissance(...)*, *get/setNationalite(...)* n'ont pas besoin d'être réécrites dans la classe *Etudiant*. En effet, grâce au polymorphisme celles-ci peuvent être utilisées aussi bien par des instances de *Personne* que par des instances de la classe *Etudiant*.

Nous allons voir ici que le polymorphisme permet également de factoriser le code des constructeurs. Par exemple, nous avons vu qu'un *Etudiant* "**est-une**" *Personne*, il paraît donc naturel que pour construire un *Etudiant* il faille procéder d'une façon

similaire à celle pour construire une *Personne* et que les constructeurs de la classe *Etudiant* puissent s'appuyer sur ceux de la classe *Personne*.

Le diagramme de la figure 1.24 indique que la classe *Etudiant* doit proposer deux constructeurs :

- l'un permettant de construire un *Etudiant* étant donné son prénom, son nom, sa nationalité, son année de naissance et son numéro d'étudiant;
- l'autre permettant de construire un *Etudiant* à partir de son prénom, de son nom, de son année de naissance et de son numéro d'étudiant. Dans ce cas on supposera que l'*Etudiant* créé a la nationalité par défaut.

Une première écriture naïve du premier de ces deux constructeurs pourrait être :

```

1  /*
2   * Une première écriture peu optimale
3   * du constructeur à 5 paramètres de la
4   * classe Etudiant
5   */
6  public Etudiant(String prenom, String nom,
7                  String nationalite,
8                  int annéeNaissance,
9                  int numEtudiant){
10    setPrenom(prenom);
11    setNom(nom);
12    setNationalite(nationalite);
13    setAnneeDeNaissance(annéeNaissance);
14    setNumEtudiant(numEtudiant);
15 }
```

En regardant cette définition on peut constater que les lignes 8 à 11 sont rigoureusement identiques au contenu du constructeur à 4 paramètres de la classe *Personne* de l'exemple 1.5. Dans sa définition il serait donc pratique de pouvoir indiquer que pour construire un *Etudiant* étant donnés son prénom, son nom, sa nationalité, son année de naissance et son numéro d'étudiant, il est nécessaire de procéder comme suit :

1. construire une *Personne* (la sur-classe de la classe *Etudiant*) à partir de son prénom, son nom, sa nationalité et son année de naissance,
2. lui attribuer son numéro d'étudiant.

Le “constructeur” *super(...)* permet de procéder de la sorte. Lorsqu'il est utilisé dans un constructeur, il fait référence à un autre constructeur de la sur-classe. Ainsi, le constructeur à 5 paramètres de la classe *Etudiant* peut s'écrire :

```

/*
 * Une écriture du constructeur
 * à 5 paramètres de la classe Etudiant
 * qui s'appuie sur un constructeur de la
 * sur-classe : la classe Personne.
 */
public Etudiant(String prenom, String nom, String nationalite,
                 int annéeNaissance, int numEtudiant){
    // On commence par construire une Personne
    super(prenom, nom, nationalite, annéeNaissance);
    // et on lui ajoute un numéro d'étudiant
    setNumEtudiant(numEtudiant);
}

```

Pour le second constructeur `Etudiant(prenom, nom, année, num)`, on peut constater qu'il est strictement équivalent à :

```
Etudiant(prenom, nom, Personne.NATIONALITE_PAR_DEFAUT, année, num)
```

Nous pouvons donc donner une définition succincte du second constructeur en utilisant `this(...)` vu dans la section 1.5.4 :

```

public Etudiant(String prenom, String nom,
                 int annéeNaissance, int numEtudiant){
    this(prenom, nom, Personne.NATIONALITE_PAR_DEFAUT,
         annéeNaissance, numEtudiant);
}

```

**Important :** lorsqu'il est utilisé dans un constructeur, l'appel à `super(...)` doit être la première ligne.

C'est aussi le cas du "constructeur" `this(...)`. On peut donc déduire qu'un constructeur ne peut utiliser qu'un seul des deux "constructeurs" spéciaux : `super(...)` ou `this(...)`.

### 1.8.7.1 À savoir absolument sur les Constructeurs

La règle suivante concernant les constructeurs d'une classe est extrêmement importante et doit être connue **par cœur**. Comme la règle vue dans la section 1.5.5 elle sera très utile pour régler des problèmes de compilation.

**Un constructeur fait toujours un appel à un constructeur de la sur-classe avec `super(...)` :**

1. C'est soit un appel explicite écrit par l'utilisateur en première ligne du constructeur;
2. Ou bien c'est un appel sans argument à `super()` ajouté par le compilateur sinon.

### 1.8.7.2 Héritages et constructeurs : on récapitule

D'après ce qui a été vu en 1.5.5 :

- **Règle 1 :** toute classe possède au moins un constructeur :
  - ceux définis explicitement dans la classe, s'il y en a;
  - un constructeur par défaut dans le cas contraire ;

La seconde règle que nous avons vu en 1.8.7.1 nous dit que :

- **Règle 2 :** tout constructeur appelle un constructeur de sa sur-classe :
  - un appel à super(...) figurant explicitement en première ligne du constructeur;
  - un appel à super() ajouté implicitement par le compilateur en première ligne du constructeur.

Ces deux règles peuvent être la source d'erreurs de compilation difficiles à corriger.

Considérons la classe suivante :

```
public class SurClasse {
    private int num;
    public SurClasse(int num){
        this.num=num;
    }
}
```

Elle compile sans problème.

Considérons maintenant une de ses sous-classes :

```
public class SousClasse extends SurClasse {
    // On peut difficilement faire plus simple !
}
```

Sa compilation provoque l'erreur suivante. Elle nous indique que le constructeur sans paramètre de la classe *SurClasse* est appelé alors qu'il n'existe pas (seul un constructeur à un argument de type *int* existe).

```
SousClasse.java:1: error: constructor SurClasse
      in class SurClasse cannot be applied to given types;
public class SousClasse extends SurClasse{
    ^
    required: int
    found: no arguments
    reason: actual and formal argument lists differ in length
1 error
```

Il est nécessaire d'avoir une bonne connaissance des deux règles ci-dessus pour corriger cette erreur. La classe *SousClasse* ne définit aucun constructeur : selon la **règle 1**, le compilateur ajoute implicitement le constructeur sans paramètre *SousClasse()*. Ce constructeur implicite ne fait pas explicitement d'appel au constructeur de la sur-classe (en fait il ne fait rien du tout), conformément à la **règle 2**, le compilateur ajoute un appel implicite à *super()* dans le constructeur *SousClasse()*.

Bien que la classe *SousClasse* ne définisse rien explicitement, le compilateur lui a ajouté le constructeur :

```
public SousClasse(){
    super();
}
```

Ceci nous ramène à la **Règle 1**. La classe *SurClasse* définit explicitement le constructeur *SurClasse(int num)*, elle n'a donc pas de constructeur sans paramètre. C'est exactement ce que nous indique l'erreur de compilation : l'appel au constructeur *super()* ajouté implicitement dans le constructeur *SousClasse()*, lui aussi, ajouté implicitement, n'aboutit pas.

Il y a deux solutions pour corriger cette erreur :

1. ajouter un constructeur sans paramètre *SurClasse()* dont le corps peut être vide.  
Ainsi l'appel à *super()* pourra aboutir;

```
public SurClasse(){  
}
```

2. ajouter un appel explicite à un constructeur de *SurClasse* qui existe, dans le constructeur de *SousClasse*.

```
public SousClasse(int num){  
    super(0); // par exemple  
    this.num=num  
}
```

### 1.8.8 Quelques exercices

Les exercices suivants ont pour objectifs :

1. de mieux comprendre les subtilités de l'héritage,
2. d'expérimenter l'héritage sur des exemples concrets,
3. de voir un exemple concret de polymorphisme.

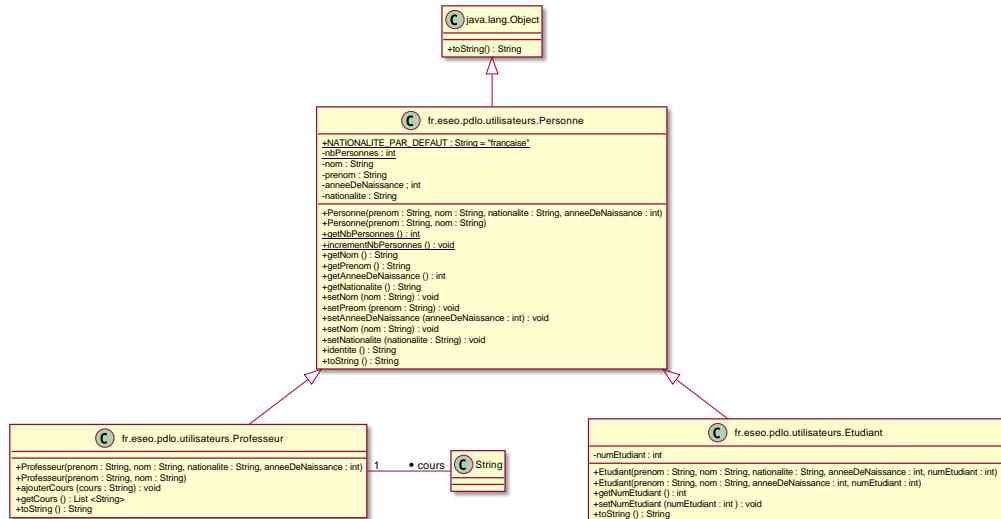


FIGURE 1.29 – Etudiants et professeurs

#### Exercice 1 :

1. Écrire la classe `fr.eseo.pdlo.exercices.utilisateurs.Etudiant` représentée dans le diagramme de la figure 1.29. On n'oubliera pas de définir la méthode `toString()` permettant de retourner une représentation textuelle des instances de la classe.

On prendra bien soin de s'appuyer sur les méthodes et constructeurs de la sur-classe dans les définitions de la classe *Etudiant*.

2. Écrire la classe *fr.eseo.pdlo.essais.exercices.utilisateurs.EtudiantEssai* qui permet de vérifier que la classe *Etudiant* fonctionne correctement.
3. Écrire la classe *fr.eseo.pdlo.exercices.utilisateurs.Professeur* qui représente un enseignant tel qu'il est modélisé dans le diagramme UML de la figure 1.29. On pourra utiliser la classe *java.util.ArrayList* définie dans l'API pour stocker la liste des cours enseignés par un *Professeur* (on verra dans la section 1.10 dédiée aux interfaces, qu'une *java.util.ArrayList* est une implémentation de l'interface *java.util.List* et peut donc être utilisée ici). Lors de la construction d'un *Professeur* on prendra bien soin d'initialiser la structure chargée de stocker ses cours (voir la section 1.4.1.8 pour des détails sur l'instanciation des classes génériques).
4. Écrire la classe *fr.eseo.pdlo.essais.exercices.utilisateurs.ProfesseurEssai* qui permet de vérifier que la classe *Professeur* fonctionne correctement.
5. Dans la classe *Annuaire* vue dans l'exercice 3 de la section 1.7.5 ajouter la méthode :

```
public Personne[] versTableau(){
    ...
}
```

qui retourne un tableau contenant toutes les *Personne* contenues dans l'annuaire.

6. Écrire la classe *AnnuaireTabEssai* permettant de vérifier le bon fonctionnement de cette dernière méthode. Lors de cette vérification :
  - on ajoutera des *Personne*, des *Etudiant* et des *Professeur* dans l'annuaire ;
  - on construira le tableau contenant les différents *Personne/Etudiant/Professeur* ;
  - on affichera la représentation textuelle des différents éléments du tableau.

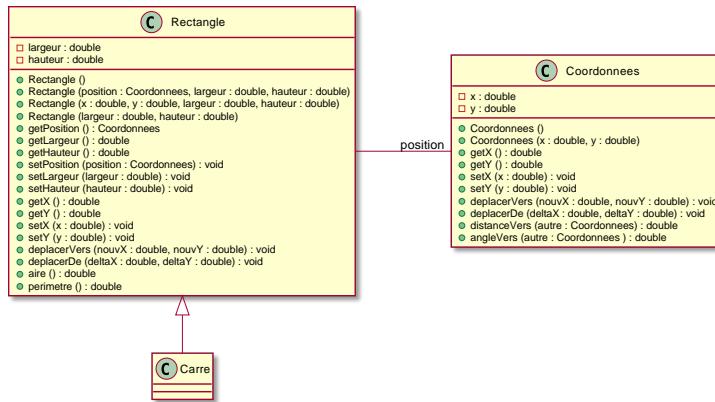


FIGURE 1.30 – Coordonnées et Carrés

### Exercice 2 :

- Écrire la classe `fr.eseo.pdlo.projet.geom.Carre` (sous-classe de `Rectangle`) fourni ssant les mêmes fonctionnalités que la sur-classe `Rectangle` (voir le diagramme de classes de la figure 1.30). On prendra bien soin de faire en sorte qu'un `Carre` ne puisse jamais avoir une largeur et une hauteur différente. Pour ça, on portera une attention particulière aux accesseurs/mutateurs des variables d'instance correspondantes et aux constructeurs de la classe. On fera également en sorte que la classe `Carre` propose des constructeurs semblables à ceux de la sur-classe `Rectangle`.

On en profitera pour ajouter deux constantes de classe dans la classe `Rectangle` (`LARGEUR_PAR_DEFAUT` et `HAUTEUR_PAR_DEFAUT`) qui permettront de spécifier les dimensions par défaut d'un `Rectangle`. On supposera que `LARGEUR_PAR_DEFAUT` est la dimension par défaut pour un `Carre`.

On ajoutera une méthode `toString()` dans la classe `Rectangle` qui retournera une description textuelle de l'objet. Cette description contiendra les informations suivantes sur une seule ligne :

```
[Rectangle] pos : (<x>,<y>) dim : <l> x <h>
périmètre : <p> aire : <a>
```

La méthode `toString()` de la classe `Rectangle` sera définie de sorte à ce qu'elle puisse aussi convenir pour un `Carre`. Elle affichera alors les éléments suivants sur une seule ligne :

```
[Carre] pos : (<x>,<y>) dim : <l> x <h>
périmètre : <p> aire : <a>
```

On utilisera une méthode définie dans la classe `Class` de l'API pour connaître le nom de la classe d'un objet donné.

- Écrire la classe `fr.eseo.pdlo.essais.projet.geom.CarreEssai` pour vous assurer que les instances de `Carre` se comportent comme souhaitées.
- Écrire la classe `fr.eseo.pdlo.projet.geom.Ellipse` sur le modèle de la classe `Rectangle`. Comme pour la classe `Rectangle`, on définira deux constantes de classe dans la classe `Ellipse` (`LARGEUR_PAR_DEFAUT` et `HAUTEUR_PAR_DEFAUT`) qui permettront de spécifier les dimensions par défaut d'une `Ellipse`.
- Écrire la classe `fr.eseo.pdlo.essais.projet.geom.EllipseEssai` pour vous assurer que les instances de `Ellipse` se comportent correctement.
- Écrire la classe `fr.eseo.pdlo.projet.geom.Cercle` sous classe de `fr.eseo.pdlo.projet.geom.Ellipse`. On supposera que `LARGEUR_PAR_DEFAUT` est la dimension par défaut pour un `Cercle`.
- Écrire la classe `fr.eseo.pdlo.essais.projet.geom.CercleEssai` pour vous assurer que les instances de `Cercle` se comportent correctement.

## 1.9 Les classes abstraites

Comme de nombreux langages objets, Java donne la possibilité de définir des classes représentant des concepts abstraits : les classes abstraites.

Considérons les classes *Rectangle*, *Carre*, *Ellipse* et *Cercle* vues précédemment, elles ont toutes un point commun : elles représentent des formes géométriques bien concrètes ; que nous pouvons dessiner, dont nous pouvons calculer l'aire et le périmètre. Or, bien que le concept de forme géométrique soit abstrait ; dans le sens où il n'est pas possible, par exemple, de définir de méthodes permettant de calculer le périmètre ou l'aire d'une forme géométrique, ni de la dessiner, on sait qu'une forme géométrique, quelle qu'elle soit, doit connaître sa position, sa largeur et sa hauteur ou encore, savoir se déplacer. Il serait donc intéressant de pouvoir regrouper ces fonctionnalités communes à toutes les formes géométriques (les rectangles, les carrés, les cercles, les ellipses et d'autres formes géométriques qui pourront être imaginées ultérieurement) dans une classe.

Les classes abstraites proposent une solution à ce problème. Elles servent à factoriser, au sein d'une seule et même entité, une structure et/ou un comportement communs à d'autres classes plus concrètes.

Dans l'exemple évoqué ci-dessus, une forme géométrique pourrait être définie comme étant "quelque chose" ayant une position, une largeur et une hauteur (et bien sûr des accesseurs et des mutateurs permettant de lire et d'écrire ces informations), sachant se déplacer et calculer son aire et son périmètre. Autant, certaines de ces méthodes, telles que les accesseurs, les mutateurs et les méthodes de déplacement sont parfaitement concrètes et se contentent de lire, d'écrire ou de modifier les valeurs de variables d'instance, d'autres comme les méthodes de calcul du périmètre ou de l'aire sont parfaitement abstraites et ne pourraient pas être définies de manière générique pour toutes les formes géométriques existantes. On peut donc distinguer deux types de méthodes :

- celles pour lesquelles on peut donner une définition "générique" : les **méthodes concrètes**,
- celles pour lesquelles on ne peut pas donner de définition "générique" mais dont on peut dire qu'elles devront nécessairement exister : les **méthodes abstraites**.

Puisqu'on ne peut pas donner de définition à une méthode abstraite, seule sa signature (visibilité, type de retour, nom de la méthode, types et noms des paramètres) apparaîtra dans la définition de la classe.

Les classes abstraites ont pour vocation d'être sous-classées, la définition concrète d'une méthode abstraite devra être donnée dans les sous classes de la classe abstraite dans laquelle elle est définie.

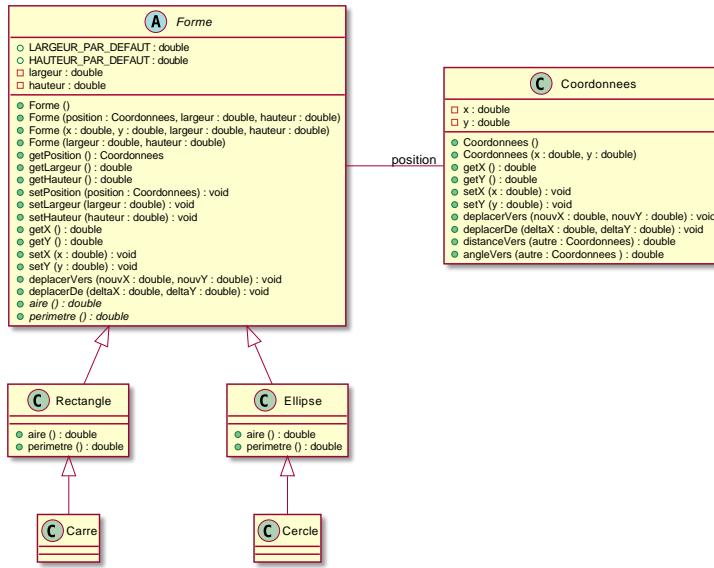


FIGURE 1.31 – Forme

### 1.9.1 Définition d'un classe abstraite

La classe *Forme* présentée ci-dessus est chargée de factoriser ce qui est commun aux différentes sortes de formes géométriques. Comme le montre le diagramme de classes de la figure 1.31 elle peut être définie en tant que sur-classe des classes qui représentent les différentes formes géométriques (*Rectangle*, *Carre*, *Ellipse* et *Cercle*).

On peut remarquer que dans la classe abstraite *Forme*, une grande majorité des méthodes sont concrètes. Seules les méthodes *aire()* et *perimetre()* doivent être abstraites ; pour cette raison leurs noms apparaissent en lettres italiques dans le diagramme (tout comme le nom de la classe *Forme* elle-même qui est également abstraite). Seules ces deux méthodes ne peuvent pas être définies de façon générique pour les différentes formes géométriques rectangle, carré, ellipse et cercle.

Le mot clef de Java *abstract* permet de spécifier qu'une classe ou qu'une méthode est abstraite. Le code ci-dessous donne la définition de la classe abstraite *Forme* :

```

public abstract class Forme {
    /*
     * La classe abstraite Forme définit des constantes
     * de classe publiques. On y accédera de la façon
     * suivante :
     * Forme.LARGEUR_PAR_DEFAUT et Forme.HAUTEUR_PAR_DEFAUT
     */
    public static final double LARGEUR_PAR_DEFAUT = 100;
    public static final double HAUTEUR_PAR_DEFAUT = 150;
    /* Les variables d'instances privées de la classe
     * abstraite
     */
    private double largeur;
    private double hauteur;
    private Coordonnees position;
}

```

```
/*
 * Les constructeurs de la classe abstraite Forme.
 * Ils ne peuvent pas être utilisés pour construire
 * des Formes (une classe abstraite ne pouvant pas
 * avoir d'instance) mais pourront l'être par ses
 * sous classes.
 */
public Forme(){
    this(new Coordonnees(),LARGEUR_PAR_DEFAUT,
         HAUTEUR_PAR_DEFAUT);
}

public Forme(double x, double y, double largeur, double hauteur){
    this(new Coordonnees(x,y),largeur,hauteur);
}

public Forme(Coordonnees position, double largeur, double hauteur){
    setPosition(position);
    setLargeur(largeur);
    setHauteur(hauteur);
}

public Forme(double largeur, double hauteur){
    this(new Coordonnees(), largeur, hauteur);
}

public Forme(Coordonnees position){
    this(position, LARGEUR_PAR_DEFAUT, HAUTEUR_PAR_DEFAUT);
}

/*
 * Les méthodes d'instances de la classe abstraite Forme
 * seront héritées par les sous-classes de Forme.
 * Ces dernières pourront donc les utiliser.
 */
public Coordonnees getPosition(){
    return position;
}

public void setPosition(Coordonnees position){
    this.position=position;
}
... // Définition des autres méthodes d'instances concrètes
```

```

/*
 * Déclaration des méthodes abstraites de la classe.
 * Seules les signatures des méthodes abstraites
 * doivent être spécifiées.
 */
public abstract double aire();

public abstract double perimetre();
}

```

Les deux méthodes *aire()* et *perimetre()* étant abstraites, les sous-classes de la classe *Forme* doivent nécessairement fournir des définitions concrètes de ces méthodes afin d'être des classes concrètes. C'est le cas, par exemple, de la classe *Rectangle* dont la définition (conformément au diagramme de classes de la figure 1.31) pourrait être :

```

public class Rectangle extends Forme {

    public Rectangle(Coordonnees position,
                      double largeur, double hauteur){
        super(position, largeur, hauteur);
    }

    public Rectangle(){
        this(new Coordonnees(), Forme.LARGEUR_PAR_DEFAUT,
             Forme.HAUTEUR_PAR_DEFAUT);
    }

    public Rectangle(double x, double y, double largeur, double hauteur){
        this(new Coordonnees(x,y),largeur,hauteur);
    }

    public Rectangle(double largeur, double hauteur){
        this(new Coordonnees(), largeur, hauteur);
    }

    public Rectangle(Coordonnees position){
        this(position, Forme.LARGEUR_PAR_DEFAUT,
              Forme.HAUTEUR_PAR_DEFAUT);
    }

    /*
     * Implémentation des méthodes abstraites de la
     * classe Forme
     */
    public double aire(){
        return getLargeur()*getHauteur();
    }

    public double perimetre(){
        return 2*(getLargeur()+getHauteur());
    }
}

```

**Important :** Les sous-classes concrètes d'une classe abstraite **doivent** donner des définitions concrètes à toutes les méthodes abstraites de leur sur-classe. Si elles ne le font pas elles doivent être déclarées en tant que classes abstraites.

### 1.9.2 Constructeurs et classes abstraites

Parmi toutes les classes, seules les classes abstraites ne peuvent pas avoir d'instance. Néanmoins, comme on peut le voir dans la définition de la classe *Forme* ci-dessus, rien n'empêche de définir des constructeurs dans une classe abstraite. Ceux-ci ne serviront pas directement à la classe abstraite mais pourront être utilisés par ses sous-classes pour simplifier l'écriture de leurs constructeurs. C'est par exemple ce qui est fait dans les constructeurs de la classe *Rectangle* ci dessus. En effet, le premier constructeur de la classe *Rectangle* s'appuie sur un constructeur de la sur-classe pour créer une instance de *Rectangle* (ce qui est matérialisé par l'appel au constructeur *super(...)*). Tous les autres constructeurs de *Rectangle* sont vus comme des cas particuliers du premier constructeur et font appel à lui à l'aide du constructeur *this(...)*.

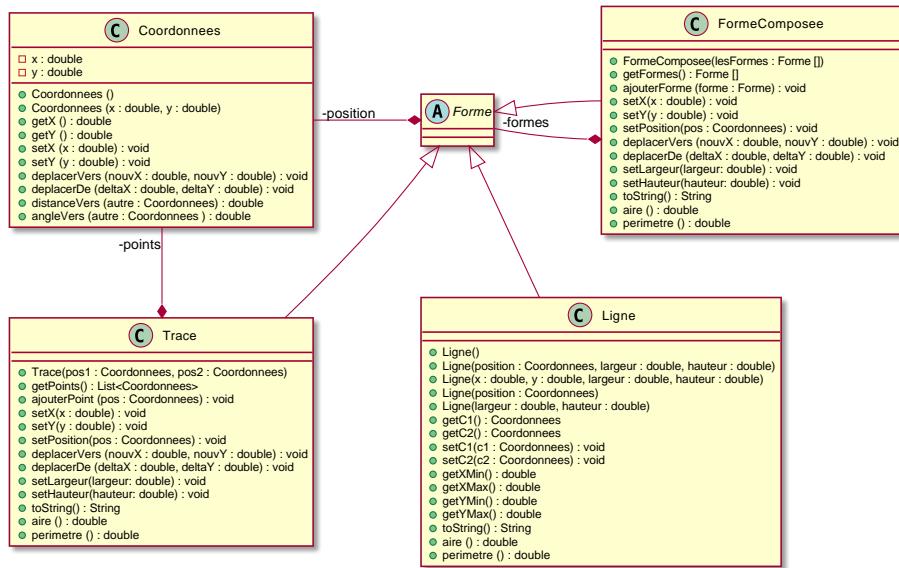
### 1.9.3 Quelques exercices

Les exercices suivants ont pour objectifs :

1. de mieux comprendre le concept de classe et de méthode abstraites,
2. de mettre en œuvre les notions de classe et de méthode abstraites sur un exemple concret.

#### Exercice 1 :

1. Définir la classe abstraite *fr.eseo.pdlo.projet.geom.Forme* décrite dans le diagramme de classes de la figure 1.31.
2. Dans le paquetage *fr.eseo.pdlo.projet.geom* , modifier les définitions des classes *Rectangle*, *Carre*, *Ellipse* et *Cercle* afin qu'elles soient des sous-classes de la classe abstraite *Forme* comme c'est illustré par le diagramme de la figure 1.31.
3. Reprendre les classes de test des classes *Rectangle*, *Carre*, *Ellipse* et *Cercle* et vérifier qu'elles fonctionnent toujours avec la nouvelle hiérarchie de classes. Il ne devrait pas être nécessaire de modifier quoi que ce soit dans ces classes pour les adapter à la nouvelle hiérarchie.

FIGURE 1.32 – *Ligne*, *Trace* et *FormeComposee***Exercice 2 :**

- Écrire la classe *fr.eseo.pdlo.projet.geom.Ligne* décrite dans le diagramme de classes de la figure 1.32. Une ligne est définie par :
  - sa position : le point qui correspond à l'une des deux extrémités de la ligne (aussi appelé le point *p1*);
  - sa largeur : la distance horizontale entre sa position et l'autre extrémité de la ligne (aussi appelé le point *p2*);
  - sa hauteur : la distance verticale entre sa position et l'autre extrémité de la ligne.

D'après la définition ci-dessus, on peut remarquer que, dans le cas d'une ligne, la largeur et la hauteur peuvent être négatives. Ce n'est pas le cas pour les autres formes vues précédemment.

Le *périmètre* d'une ligne sera considéré comme étant sa longueur, tandis que son *aire* sera nulle.

La représentation textuelle d'une ligne aura la forme suivante :

```
[Ligne] p1 : (<x>,<y>), p2 : (<x>,<y>), longueur : <l>
```

- Écrire la classe *fr.eseo.pdlo.essais.projet.geom.LigneEssai* permettant de vérifier le bon fonctionnement des méthodes de la classe *Ligne*.
- Écrire la classe *fr.eseo.pdlo.projet.geom.Trace* décrite dans le diagramme de classes de la figure 1.32. Un *Trace* représente un tracé à main levée et est constitué d'un ensemble de points. Par exemple, le tracé à main levée à droite de la figure 1.34 peut être représenté par les points d'extrémités des segments de la figure de gauche.

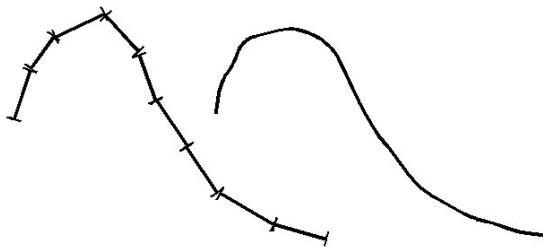


FIGURE 1.33 – Représentation d'un tracé à main levée

Le constructeur à deux paramètres permettra d'initialiser la construction d'un *Trace* en lui donnant les deux extrémités de la première ligne qui le compose.

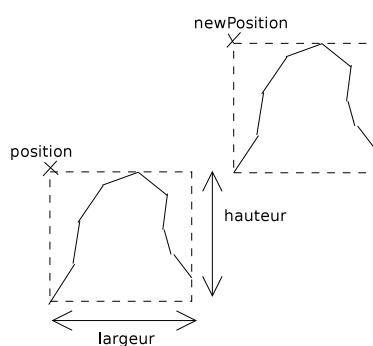
On définira la position d'un *Trace* comme étant le coin supérieur gauche du rectangle englobant le *Trace*. On considérera que les largeur et hauteur d'un *Trace* sont les dimensions du rectangle qui englobe le tracé. Ainsi :

- le changement de la position d'un *Trace* correspondra à un déplacement de son rectangle englobant ;
- le changement de la largeur (ou de la hauteur) d'un *Trace* correspondra à un étirement horizontal (ou vertical) de son rectangle englobant.

On considérera que le *périmètre* d'un *Trace* est la longueur totale du *Trace* et que son *aire* est nulle.

Enfin, la représentation textuelle d'un *Trace* prendra la forme (sur une seule ligne) :

```
[Trace] pos : (<x>,<y>) dim : <l> x <h>
longueur : <p> nbLignes : <n>
```

FIGURE 1.34 – Position, largeur et hauteur d'un *Trace*

4. Écrire la classe *fr.eseo.pdlo.essais.projet.geom.TraceEssai* permettant de vérifier le bon fonctionnement des méthodes de la classe *Trace*.
5. Écrire la classe *FigureComposee* décrite dans le diagramme de classe de la figure 1.32
6. Écrire la classe *fr.eseo.pdlo.essais.projet.geom.FormeComposeeEssai* permettant de vérifier le bon fonctionnement des méthodes de la classe *FormeComposee*.

FIGURE 1.35 – L’interface *Dessinable*

## 1.10 Les interfaces

Tout comme les classes et les tableaux, les interfaces de Java représentent une nouvelle sorte de type référence. Elles fournissent un moyen de garantir le comportement d’objets en spécifiant un ensemble de fonctionnalités (sous la forme de signature de méthodes) qu’ils devront implémenter. Comme son nom l’indique, une interface permet seulement de spécifier une *interface* de programmation pour les applications (une *API - Application Programming Interface*) : toutes ses méthodes sont abstraites et elle ne définit aucune variable ou constante. Tout comme c’est le cas pour les classes abstraites, il n’est pas possible d’instancier une interface ; des classes pourront spécifier qu’elles implémentent (c’est-à-dire qu’elles s’engagent à donner des implémentations pour toutes les méthodes spécifiées par l’interface) une ou plusieurs interfaces mais aucune instance de ces interfaces ne pourra être créée en tant que telles.

D’une certaine façon, on peut considérer qu’une interface permet de spécifier un contrat qu’une classe peut décider d’honorer si elle décide d’implémenter l’interface. Par exemple, l’interface décrite dans le fragment de code suivant spécifie ce que doit implémenter une classe qui souhaiterait être considérée comme *Dessinable*.

```

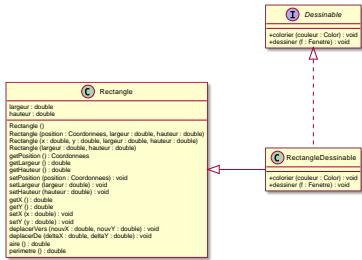
public interface Dessinable { // Un objet est dessinable
    // si on peut
    void colorier(Color c); // le colorier avec
    // une couleur donnée
    void dessiner(Fenetre f); // et le dessiner dans
    // une fenêtre
}

```

Sur un diagramme de classes UML, l’interface *Dessinable* serait représenté par le diagramme de la figure 1.35.

La déclaration d’une interface se fait à l’aide du mot clef *interface* et peut être précédée des mêmes modificateurs de visibilité que ceux pouvant être placés devant la déclaration d’une classe (*public* ou *aucun*). La convention de nommage pour les noms d’interface est identique à celle utilisée pour les noms de classes. Leurs noms doivent donc commencer par une majuscule.

À première vue, on constate qu’une interface ressemble fortement à une classe abstraite. Il existe cependant une différence fondamentale entre ces deux types référence du langage Java : une **classe permet de décrire ce que sont** des objets tandis qu’**une interface permet de spécifier ce que savent faire** des objets. Par exemple, la classe abstraite *Forme* de la section 1.9 décrit une *Forme* comme **étant** quelque chose qui possède une largeur, une hauteur, une position et qui peut répondre à certaines sollicitations transmises sous la forme de messages. En revanche, l’interface *Dessinable* ci-dessus, spécifie qu’un objet est *dessinable* s’il **sait** se *colorier* dans une couleur donnée et **sait**

FIGURE 1.36 – Un *RectangleDessinable*

se *dessiner* dans une fenêtre donnée.

Les interfaces de Java peuvent être vues comme des contrats, auxquels les classes peuvent décider de souscrire en les implémentant.

### 1.10.1 Implémentation d'interfaces

Une classe peut *implémenter* une ou plusieurs interfaces ; dans ce cas la classe doit fournir des implémentations pour toutes les méthodes spécifiées par la ou les interfaces implémentées. Comme illustré dans le code de l'exemple 1.8, le mot clef *implements* de Java permet d'indiquer la relation d'implémentation entre classes et interfaces. Une classe peut, à la fois, implémenter une (ou plusieurs) interfaces et être sous-classe d'une classe.

Le diagramme de classe de la figure 1.36 illustre la relation d'implémentation en UML et montre comment elle peut cohabiter avec la relation d'héritage. Dans cet exemple, la classe **RectangleDessinable** est, à la fois une sous-classe de **Rectangle** et une implémentation de l'interface **Dessinable**.

Si dans sa définition, une classe implémente une ou plusieurs interfaces sans donner la définition de toutes les méthodes de ces interfaces, la classe doit être déclarée abstraite.

### 1.10.2 Héritage d'interfaces

Tout comme les classes, il est possible de spécifier des relations d'héritage entre les interfaces. Avec une telle relation d'héritage entre interfaces, toutes les méthodes spécifiées dans la sur-interface seront héritées par la sous-interface et viendront s'ajouter à ses propres méthodes. Par exemple, dans le diagramme de la figure 1.37, l'interface **InterfaceFille** spécifie directement deux méthodes **méthodeC()** et **méthodeD()** et hérite des méthodes **méthodeA()** et **méthodeB()** spécifiées dans la sur-interface **InterfaceMère**. Ainsi, une classe qui souhaite implémenter l'interface **InterfaceFille**, doit fournir des implémentations pour ces quatre méthodes.

Une différence majeure qui existe entre l'héritage entre classes et l'héritage entre interfaces est que, contrairement à une classe, une interface peut hériter de plusieurs sur-interfaces : l'héritage multiple est donc autorisé en Java mais **uniquement** pour les interfaces.

L'arbre d'héritage, de la figure 1.38, entre les interfaces **Flottant**, **Roulant** et **Amphibie** peut donc tout simplement être implémenté en Java de la manière suivante :

**Exemple 1.8** Implémentation d'une interface

```

/* Un RectangleDessinable "est un" Rectangle (héritage).
 * Un RectangleDessinable s'engage à "respecter le contrat"
 * défini par l'interface Dessinable (implémentation
 * d'interface). */

public class RectangleDessinable extends Rectangle
    implements Dessinable {

    /* La classe RectangleDessinable déclare implémenter
     * l'interface Dessinable.
     * Les méthodes colorier(Color couleur) et
     * dessiner(Fenetre f) qu'elle spécifie doivent
     * donc être implémentées dans la classe
     * RectangleDessinable.
     * La classe RectangleDessinable déclare être une
     * sous classe de Rectangle, elle hérite donc
     * des méthodes et attributs de celle-ci. */

    public void colorier(Color couleur){
        /* Implémentation du mécanisme permettant de
         * fixer la couleur du RectangleDessinable */
        ...
    }

    public void dessiner(Fenetre f){
        /* Implémentation du mécanisme permettant de dessiner
         * le RectangleDessinable
         * dans la fenêtre f */
        ...
    }
}

```

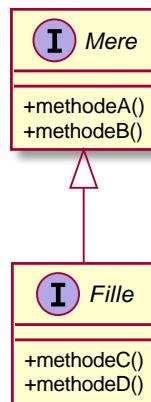


FIGURE 1.37 – Héritage entre interfaces

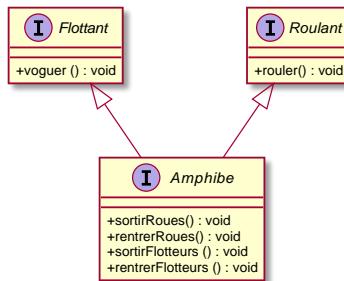


FIGURE 1.38 – Héritage multiple entre interfaces

```

public interface Amphibie extends Flottant, Roulant {
    void sortirRoues();
    void rentrerRoues();
    void sortirFlotteurs();
    void rentrerFlotteurs();
}
  
```

Ainsi, pour implémenter l'interface Amphibie, une classe devra implémenter les méthodes spécifiées dans les interfaces Flottant, Roulant et Amphibie.

### 1.10.3 Utilisation d'une interface

Le nom d'une interface peut être présent dans le code à tout endroit où le nom d'un type est attendu :

- **En tant que type d'une variable** - La variable peut être locale à une méthode, ou une variable d'instance ou de classe. Par exemple, la déclaration suivante :

```
Dessinable objetDessinable;
```

spécifie que la variable `objetDessinable` doit contenir un objet dont la classe implémente l'interface `Dessinable`. De ce fait, l'objet auquel fait référence la variable `objetDessinable` peut répondre aux messages `colorier(...)` et `dessiner(...)`.

- **En tant que paramètre d'une méthode** - Par exemple, la déclaration suivante :

```
void utiliseUnObjetDessinable(..., Dessinable unDessinable){
    ...
}
```

spécifie que la méthode `utiliseUnObjetDessinable(...)` prend en paramètre un objet dont la classe implémente l'interface `Dessinable`. Le paramètre de la méthode comprendra donc les messages `colorier(...)` et `dessiner(...)` et sa classe (ou une de ses sous-classes) définit les méthodes correspondantes.

- **En tant que type de retour d'une méthode** - Par exemple, la déclaration suivante :

```
Dessinable creerObjetDessinable(...);
```

spécifie que la méthode `creerObjetDessinable(...)` retourne un objet dont la classe implémente l'interface `Dessinable`.

### 1.10.4 Un mot sur le framework de collections de Java

Le langage Java propose un ensemble d'interfaces permettant la manipulation (stockage, parcours, recherche...) de collections d'objets. Les interfaces proposées correspondent aux types de données abstraits classiques existants : les listes, les ensembles, les dictionnaires et les files. La plupart sont des sous-interfaces de l'interface `java.util.Collection`<sup>5</sup> qui spécifie que les classes qui l'implémentent doivent définir des méthodes d'ajout, de suppression et de recherche d'instances dans la collection, des méthodes de conversion de la collection en tableau, ainsi qu'une méthode renvoyant la taille de la collection.

Les interfaces du framework de collections de Java sont génériques (voir section 1.4.1.8). Leurs types sont paramétrés par le type des objets qu'elles sont censées contenir.

#### 1.10.4.1 Les listes de l'interface `java.util.List`

L'interface `List`<sup>6</sup> représente les collections ordonnées. Elle spécifie que les classes qui l'implémentent doivent fournir des méthodes permettant de manipuler leurs éléments (les insérer, les supprimer, les rechercher) selon leur position dans la séquence. Les `List` doivent également proposer un mécanisme permettant d'itérer un traitement sur chacun de leurs éléments et d'ajouter un élément en fin de séquence.

Java propose plusieurs classes implémentant l'interface `List` et proposant plusieurs types de listes (des implémentations de l'interface `List`), par exemple :

- `java.util.ArrayList`<sup>7</sup> : des tableaux dynamiques.
- `java.util.LinkedList`<sup>8</sup> : des listes chaînées.
- `java.util.Stack`<sup>9</sup> : des piles.
- `java.util.Vector`<sup>10</sup> : des vecteurs.

#### 1.10.4.2 Les ensembles de l'interface `Set`

L'interface `Set`<sup>11</sup> représente les ensembles. Un ensemble est une collection ne pouvant contenir qu'une seule occurrence de ses éléments.

Java propose plusieurs classes implémentant l'interface `Set` proposant des implementations différentes de l'interface `Set`, par exemple :

- `java.util.HashSet`<sup>12</sup> : utilise une table de hashage pour représenter les ensembles.
- `java.util.TreeSet`<sup>13</sup> : utilise un arbre pour représenter un ensemble.

5. <http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

6. <http://docs.oracle.com/javase/8/docs/api/java/util/List.html>

7. <http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

8. <http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html>

9. <http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html>

10. <http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html>

11. <http://docs.oracle.com/javase/8/docs/api/java/util/Set.html>

12. <http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>

13. <http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>

#### **1.10.4.3 Les files de l'interface Queue**

L'interface Queue représente différents types de files. Les éléments d'une file y sont en général stockés avant de subir un traitement. En règle générale les éléments *enfilés* les premiers seront les premiers à sortir de la file. On parle alors de file FIFO (First In, First Out) ; par opposition aux collections LIFO (Last In, First Out) telles que les piles dans lesquelles les derniers éléments ajoutés sont les premiers à être dépiler.

#### **1.10.4.4 Les dictionnaires de l'interface Map**

Contrairement aux interfaces précédentes, Map n'est pas une sous-interface de Collection. À la manière d'un dictionnaire qui associe un mot à sa définition et permet d'utiliser un mot comme clef pour en trouver sa définition, une Map permet d'associer des clefs aux objets de la collection. On parle d'*association clef/valeur*. Les opérations que doivent fournir les classes qui implémentent l'interface Map sont légèrement différentes de celles de l'interface Collection. Elles doivent permettre l'ajout d'une valeur pour une clef donnée, la recherche de la valeur d'une clef, la suppression d'une clef et de sa valeur. La classe java.util.HashMap<sup>14</sup> est une implémentation couramment utilisée de l'interface Map. Le type java.util.HashMap est paramétré par deux types : celui de la clef et celui de la valeur des objets de la collection.

Une partie entière du tutoriel en ligne<sup>15</sup> traite en détail du framework de collections de Java.

## **1.11 Les exceptions et la gestion d'erreur**

Java propose un mécanisme très puissant pour gérer les situations exceptionnelles qui, dans d'autres langages, pourraient provoquer des erreurs à l'exécution : c'est le mécanisme d'exceptions.

Le mécanisme d'exceptions mis en œuvre dans Java repose sur la classe java.lang.Exception. Lorsqu'une situation exceptionnelle (une *erreur d'exécution* telle que, par exemple, un accès hors des limites d'un tableau, la lecture d'une fichier inexistant, ...) est rencontrée une instance d'une des nombreuses sous-classes de java.lang.Exception est créée et *levée*. Pour gérer cette erreur d'exécution le programmeur doit prévoir un *gestionnaire d'exception* capable de l'attraper —c'est-à-dire de la gérer. Si aucun gestionnaire n'est capable de gérer une exception levée, le programme arrête son exécution et signale une erreur.

### **1.11.1 Les classes d'exceptions**

La classe java.lang.Throwable regroupe toutes les classes servant à signaler un problème à l'exécution. Chacune des classes de l'arbre d'héritage des Throwable, dont un extrait est représenté figure 1.39, permet de représenter un type d'erreur d'exécution bien précis.

Les deux sous-classes de Throwable servent :

- pour la classe java.lang.Exception, à représenter des erreurs d'exécutions qui peuvent parfois être corrigées.

---

14. <http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

15. <https://docs.oracle.com/javase/tutorial/collections/TOC.html>

## 1.11. LES EXCEPTIONS ET LA GESTION D'ERREUR 1. LE LANGAGE JAVA

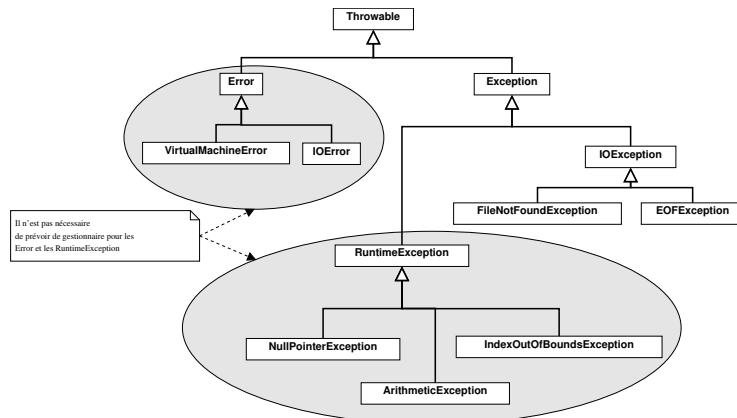


FIGURE 1.39 – Extrait de l'arbre d'héritage des `Throwable`

On distingue comme sous-classe de `java.lang.Exception` la classe `java.lang.RuntimeException` qui représente des exceptions qu'il n'est pas nécessaire de gérer : on trouve parmi ses sous classes des exceptions telles que les `NullPointerException` indiquant la tentative d'envoi d'un message à une référence nulle, `ArrayIndexOutOfBoundsException` indiquant une tentative d'accès en dehors des limites d'un tableau, ou encore `ArithmaticException` pour signaler des problèmes arithmétiques telle qu'une tentative de division par zéro.

- pour la classe `java.lang.Error`, à représenter des erreurs plus sérieuses contre lesquelles on ne peut généralement pas faire grand chose.

### 1.11.1.1 Fonctionnement et gestion des exceptions

Lorsqu'une situation exceptionnelle est rencontrée lors de l'exécution d'un programme Java, la méthode en cours d'exécution crée et *lève* (lance, *throw*) une exception.

Lorsqu'une exception est levée, la méthode en cours d'exécution (ou une de ses méthodes appelantes) peut décider de *rattraper* (*catch*) l'exception grâce à son gestionnaire d'exceptions.

Dans la pratique, lorsqu'une méthode est susceptible de lever une exception, elle doit le signaler. Le mot-clé `throws` du langage Java est réservé à cet effet. Par exemple, si on considère une méthode `lireFichier` (`String nomDeFichier`) susceptible de lever une exception de type `IOException` dans les cas où un problème d'entrée/sortie est rencontré, elle devra être déclarée de la façon suivante :

```
void lireFichier(String nomDeFichier) throws IOException {
    ... // implémentation de la méthode
}
```

---

# **Première partie**

## **Annexes**

## Annexe A

# Nommage et indentation

Pour simplifier la lisibilité de programmes écrit en Java, les concepteurs préconisent d'utiliser une convention pour nommer les différentes parties d'un programme. Dans le cadre de ce cours, et ce sera aussi le cas lorsque vous serez amené à utiliser Java dans d'autres circonstances, vous devrez vous contraindre à suivre cette convention.

### A.1 La notation *Camel Case*

On s'interdit en Java d'utiliser des tirets “-” ou tirets bas (*underscore*) “\_” dans un nom. Si on souhaite utiliser plusieurs mots pour nommer quelque chose on indiquera le changement de mots grâce à un changement de casse (majuscule/minuscule). Ce type de notation est appelé la notation *Camel Case*.

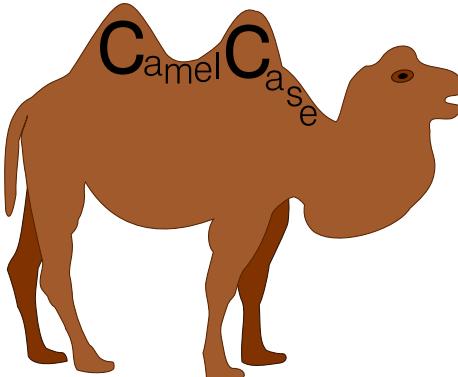


FIGURE A.1 – La notation Camel Case

Cette notation permet d'utiliser des noms très parlants, explicites et lisibles. Des exemples de noms respectant cette convention sont :

- *unNomTrèsLongEtNéanmoinsTrèsExplicite*
- *unNomCourt*
- *unDeuxièmeNomCourt*

Bien que ce soit possible d'utiliser des noms très longs, on privilégiera les noms synthétiques quand ce sera possible, mais on ne s'interdira pas d'utiliser des noms plus longs si c'est nécessaire.

## A.2 Des noms parlants

En règle général on utilise des noms explicites pour nommer les choses (variables, instances, classes ...). On préfère ainsi utiliser le nom *compteur* plutôt que *n* ou *x* pour représenter un compteur. De même, si on doit écrire une méthode qui redimensionne une fenêtre on choisira de l'appeler *redimensionner(...)* ou *resize(...)* plutôt que d'utiliser un nom moins explicite tel que *redim(...)*, *taille(...)* ou un autre nom qui ne traduit pas explicitement la sémantique de l'opération.

Une règle simple de nommage consiste à nommer les méthodes en utilisant des verbes et les variables avec des noms communs.

## A.3 Majuscules et minuscules

La première lettre du nom de quelque chose donne une indication sur la nature de cette chose.

### A.3.1 Nom d'une classe ou d'une interface :

La première lettre du nom d'une classe ou d'une interface doit être **une majuscule** : *UneClasse, Personne, Fenêtre, UneInterface, Dessinable...*

### A.3.2 Nom d'une variable :

La première lettre du nom d'une variable (variable d'instance, variable de classe, variable locale ou paramètre) ou d'une méthode (de classe ou d'instance) doit être **une minuscule** : *uneVariable, uneMéthode, nom, prenom, monArgument...*

### A.3.3 Nom d'une constante de classe :

La **totalité** du nom d'une constante de classe doit être écrite **en majuscule**. Si le nom comprend plusieurs mots, ils doivent être séparés par des tirets bas (*underscore*) “\_” : *UN\_NOM\_DE\_CONSTANTE\_DE\_CLASSE, LONGUEUR\_PAR\_DEFAUT ...*

### A.3.4 Nom d'un paquetage :

Le nom d'un paquetage ne doit contenir que des caractères alphanumériques (lettres ou chiffres) minuscules. Si le nom du paquetage contient plusieurs mots ils ne doivent pas être séparés par des tirets “-” ou des tirets bas (*underscore*) “\_” et ne doit pas utiliser la notation *Camel Case* : *fr:eseo:gpi:figure, fr:eseo:gpi:projet:géométrie dans le plan ...*

## A.4 Indentation d'un programme Java

Il existe plusieurs recommandations d'indentation pour du code Java. Celle que nous utiliserons dans le cadre de ce cours a été pensée par Google et peut être consultée sur <https://google.github.io/styleguide/javaguide.html#s4-formatting>.

# Annexe B

## Java Google style

Cette annexe reprend les règles de formatage préconisées par le “Google Java Style” dont une description complète se trouve sur <http://google.github.io/styleguide/javaguide.html>. Certaines règles de formatage portent sur des aspects du langage qui ne sont pas traité dans ce document. Si vous êtes intéressés par ces aspects plus avancés de Java, de nombreuses documentations et tutoriels se trouvent sur Internet.

**Terminology Note :**

*block-like construct* refers to the body of a class, method or constructor. Note that, by section B.8.3.1, any array initializer may optionally be treated as if it were a block-like construct.

### B.1 Braces

#### B.1.1 Braces are used where optional

Braces are used with *if*, *else*, *for*, *do* and *while* statements, even when the body is empty or contains only a single statement.

#### B.1.2 Nonempty blocks : K & R style

Braces follow the Kernighan and Ritchie style (“*Egyptian brackets*”) for nonempty blocks and block-like constructs :

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace if that brace terminates a statement or the body of a method, constructor or named class. For example, there is no line break after the brace if it is followed by *else* or a comma.

**Example :**

```
return new MyClass() {
    @Override public void method() {
        if (condition()) {
            try {
                something();
            } catch (ProblemException e) {
                recover();
            }
        }
    }
};
```

A few exceptions for enum classes are given in Section B.8.1.

### **B.1.3 Empty blocks : may be concise**

An empty block or block-like construct may be closed immediately after it is opened, with no characters or line break in between ({}), unless it is part of a multi-block statement (one that directly contains multiple blocks : *if/else-if/else* or *try/catch/finally*).

**Example :**

```
void doNothing() {}
```

## **B.2 Block indentation : +2 spaces**

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in Section B.1.2 )

## **B.3 One statement per line**

Each statement is followed by a line-break.

## **B.4 Column limit : 80 or 100**

Projects are free to choose a column limit of either 80 or 100 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in Section B.5.

**Exceptions :**

1. Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference).
2. *package* and *import* statements (see Sections 3.2 Package statement and 3.3 Import statements of the original document at <http://google.github.io/styleguide/javaguide.html>).
3. Command lines in a comment that may be cut-and-pasted into a shell.

## B.5 Line-wrapping

### Terminology Note :

When code that might otherwise legally occupy a single line is divided into multiple lines, typically to avoid overflowing the column limit, this activity is called line-wrapping.

There is no comprehensive, deterministic formula showing exactly how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

**Tip :** Extracting a method or local variable may solve the problem without the need to line-wrap.

### B.5.1 Where to break

The prime directive of line-wrapping is : prefer to break at a higher syntactic level.  
Also :

1. When a line is broken at a non-assignment operator the break comes before the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)
  - This also applies to the following "operator-like" symbols : the dot separator (.), the ampersand in type bounds ( $<T \text{ extends } Foo \& Bar>$ ), and the pipe in catch blocks (*catch (FooException | BarException e)*).
2. When a line is broken at an assignment operator the break typically comes after the symbol, but either way is acceptable.
  - This also applies to the "assignment-operator-like" colon in an enhanced *for* ("foreach") statement.
3. A method or constructor name stays attached to the open parenthesis () that follows it.
4. A comma (,) stays attached to the token that precedes it.

### B.5.2 Indent continuation lines at least +4 spaces

When line-wrapping, each line after the first (each continuation line) is indented at least +4 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +4 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

Section B.6.3 on horizontal alignment addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

## B.6 Whitespace

### B.6.1 Vertical Whitespace

A single blank line appears :

1. Between consecutive members (or initializers) of a class : fields, constructors, methods, nested classes, static initializers, instance initializers.

- **Exception :** A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create logical groupings of fields.
- 2. Within method bodies, as needed to create logical groupings of statements.
- 3. *Optionally* before the first member or after the last member of the class (neither encouraged nor discouraged).
- 4. As required by other sections of this document (such as Section 3.3 Import statements of the original document at <http://google.github.io/styleguide/javaguide.html>).

*Multiple* consecutive blank lines are permitted, but never required (or encouraged).

### B.6.2 Horizontal whitespace

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places **only**.

1. Separating any reserved word, such as *if*, *for* or *catch*, from an open parenthesis () that follows it on that line
2. Separating any reserved word, such as *else* or *catch*, from a closing curly brace {} that precedes it on that line
3. Before any open curly brace {}, with two exceptions :
  - *@SomeAnnotation({a, b})* (no space is used)
  - *String[][] x = {"foo"};* (no space is required between {}, by item 8 below)
4. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols :
  - the ampersand in a conjunctive type bound : *<T extends Foo & Bar>*
  - the pipe for a catch block that handles multiple exceptions : *catch (FooException | BarException e)*
  - the colon ( :) in an enhanced *for ("foreach")* statement
5. After , ; or the closing parenthesis () of a cast
6. On both sides of the double slash (//) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.
7. Between the type and variable of a declaration : *List<String> list*
8. Optional just inside both braces of an array initializer
  - *new int[] {5, 6}* and *new int[] { 5, 6 }* are both valid

Note : This rule never requires or forbids additional space at the start or end of a line, only interior space.

### B.6.3 Horizontal alignment : never required

**Terminology Note :** Horizontal alignment is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is **never required** by Google Style. It is not even required to *Maintain* horizontal alignment in places where it was already used.

Here is an example without alignment, then using alignment :

```
private int x; // this is fine  
private Color color; // this too  
  
private int x; // permitted, but future edits  
private Color color; // may leave it unaligned
```

**Tip :** Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is allowed. More often it prompts the coder (perhaps you) to adjust whitespace on nearby lines as well, possibly triggering a cascading series of reformattings. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.

## B.7 Grouping parentheses : recommended

Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is not reasonable to assume that every reader has the entire Java operator precedence table memorized.

## B.8 Specific constructs

### B.8.1 Enum classes

After each comma that follows an enum constant, a line-break is optional.

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see Section B.8.3.1).

```
private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

Since enum classes are classes, all other rules for formatting classes apply.

### B.8.2 Variable declarations

#### B.8.2.1 One variable per declaration

Every variable declaration (field or local) declares only one variable : declarations such as `int a, b;` are not used.

#### B.8.2.2 Declared when needed, initialized as soon as possible

Local variables are not habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

### B.8.3 Arrays

#### B.8.3.1 Array initializers : can be "block-like"

Any array initializer may optionally be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list) :

```
new int [] {          new int [] {  
    0, 1, 2, 3          0,  
}                      1,  
                    2,  
new int [] {          3  
0, 1,                  }  
2, 3  
}                      new int []  
                      {0, 1, 2, 3}
```

#### B.8.3.2 No C-style array declarations

The square brackets form a part of the type, not the variable :

```
String [] args, not String args []
```

#### B.8.3.3 Switch statements

**Terminology Note :** Inside the braces of a switch block are one or more statement groups. Each statement group consists of one or more switch labels (either *case FOO :* or *default :*), followed by one or more statements.

#### B.8.3.4 Indentation

As with any other block, the contents of a switch block are indented +2.

After a switch label, a newline appears, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

#### B.8.3.5 Fall-through : commented

Within a switch block, each statement group either terminates abruptly (with a break, continue, return or thrown exception), or is marked with a comment to indicate that execution will or might continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically // fall through). This special comment is not required in the last statement group of the switch block.

**Example :**

```

switch (input) {
    case 1:
    case 2:
        prepareOneOrTwo();
        // fall through
    case 3:
        handleOneTwoOrThree();
        break;
    default:
        handleLargeNumber(input);
}

```

#### B.8.3.6 The default case is present

Each switch statement includes a *default* statement group, even if it contains no code.

### B.8.4 Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (Section B.5), so the indentation level is not increased.

**Example :**

```

@Override
@Nullable
public String getNameIfPresent() { ... }

```

**Exception :** A single parameterless annotation may instead appear together with the first line of the signature, for example :

```

@Override public int hashCode() { ... }

```

Annotations applying to a field also appear immediately after the documentation block, but in this case, multiple annotations (possibly parameterized) may be listed on the same line ; for example :

```

@Partial @Mock DataLoader loader;

```

There are no specific rules for formatting parameter and local variable annotations.

### B.8.5 Comments

#### B.8.5.1 Block comment style

Block comments are indented at the same level as the surrounding code. They may be in /\* ... \*/ style or // ... style. For multi-line /\* ... \*/ comments, subsequent lines must start with \* aligned with the \* on the previous line.

```
/*
 * This is                      // And so                  /* Or you can
 * okay.                         // is this.
 * even do this. */
*/
```

Comments are not enclosed in boxes drawn with asterisks or other characters.

**Tip :** When writing multi-line comments, use the /\* ... \*/ style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in //... style comment blocks.

### B.8.6 Modifiers

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification :

```
public protected private abstract static final
transient volatile synchronized native strictfp
```

### B.8.7 Numeric Literals

long-valued integer literals use an uppercase *L* suffix, never lowercase (to avoid confusion with the digit 1). For example, 3000000000L rather than 3000000000l.

## Annexe C

# Les commentaires *javadoc*

### C.1 JAVA DOCUMENTATION COMMENTS

The Java language supports three types of comments :

Comment	Description
<code>/* text */</code>	The compiler ignore everything between /* and */.
<code>// text</code>	The compiler ignores everything from // to the end of the line .
<code>/** documentation */</code>	This is a documentation comment and in general its called doc comment. The JDK javadoc tool uses <i>doc</i> comments when preparing automatically generated documentation.

This tutorial is all about explaining Javadoc. We will see how we can make use of Javadoc for generating useful documentation for our Java code.

#### C.1.1 What is Javadoc ?

Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in

HTML format from Java source code which has required documentation in a predefined format.

Following is a simple example where the comments in the the code are Javadoc comments :

```
/**  
 * The HelloWorld program implements an application  
 * that simply displays "Hello World!" to the standard  
 * output.  
 *  
 * @author Zara Ali  
 * @version 1.0  
 * @since 2014-03-31  
 */  
public class HelloWorld {  
    public static void main(String[] args){  
        // Prints Hello World ! on the standard output  
        System.out.println("HelloWorld!");  
    }  
}
```

You can include required *HTML* tags inside the description part, For example, the example below makes use of *<h1>....</h1>* for heading and *<p>* has been used for creating paragraph break :

```
/**  
 * <h1>Hello, World!</h1>  
 * The HelloWorld program implements an application  
 * that simply displays "Hello World!" to the standard  
 * output.  
 * <p>  
 * Giving proper comments in your program makes it  
 * more user friendly and it is assumed as a high  
 * quality code.  
 *  
 *  
 * @author Zara Ali  
 * @version 1.0  
 * @since 2014-03-31  
 */  
public class HelloWorld {  
    public static void main(String[] args) {  
        /* Prints Hello, World! on standard output.  
         * System.out.println("Hello World!");  
    }  
}
```

### **C.1.2 The javadoc Tags :**

The javadoc tool recognizes the following tags :

## C.1. JAVA DOCUMENTATION CONNEXES. LES COMMENTAIRES JAVADOC

Tag	Description	Syntax
@author	Adds the author of a class.	@author name-text
{ @code }	Displays text in code font without interpreting the text as HTML markup or nested javadoc tags.	{ @code text }
{ @docRoot }	Represents the relative path to the generated document's root directory from any generated page	{ @docRoot root }
@deprecated	Adds a comment indicating that this API should no longer be used.	@deprecateddeprecated-text
@exception	Adds a Throws subheading to the generated documentation, with the class-name and description text.	@exception class-name description
{ @inheritDoc }	Inherits a comment from the immediate superclass.	{ @inheritDoc }
{ @link }	Inserts an in-line link with visible text label that points to the documentation for the specified package, class or member name of a referenced class.	{ @link package.class #memberlabel }
{ @linkplain }	Identical to { @link }, except the link's label is displayed in plain text than code font.	{ @linkplain package.class#member label }
@param	Adds a parameter with the specified parametername followed by the specified description to the "Parameters" section.	@param parameter-name description
@return	Adds a "Returns" section with the description text.	@return description
@see	Adds a "See Also" heading with a link or text entry that points to reference.	@see reference
@serial	Used in the doc comment for a default serializable field.	@serial field-description   include   exclude
@serialData	Documents the data written by the writeObject or writeExternal methods	@serialData datadescription
@serialField	Documents an ObjectStreamField component.	@serialField field-name field-type field-description
@since	Adds a "Since" heading with the specified since-text to the generated documentation.	@since release
@throws	The @throws and @exception tags are synonyms.	@throws class-name description
{ @value }	When { @value } is used in the doc comment of a static field, it displays the value of that constant.	{ @value package.class#field }
@version	Adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used.	@version version-text

### C.1.2.1 Example :

The following program uses several of the important tags available for documentation comments. You can

make use of other tags based on your requirements.

The documentation in the *AddNum* class will produce in HTML file *AddNum.html*. At the same time a master file named *index.html* will also be created.

```

import java.io.*;
/**
 * <h1>Add Two Numbers!</h1>
 * The AddNum program implements an application that
 * simply adds two given integer numbers and Prints
 * the output on the screen.
 * <p>
 * <b>Note:</b> Giving proper comments in your program
 * makes it more user friendly and increases the
 * quality of the code.
 *
 * @author Zara Ali
 * @version 1.0
 * @since 2014-03-31
 */
public class AddNum {
    /**
     * This method is used to add two integers. This is
     * the simplest form of a class method, just to
     * show the usage of various javadoc tags.
     * @param numA First operand of the sum
     * to be computed
     * @param numB Second operand of the sum
     * to be computed
     * @return int This returns the sum
     * of numA and numB.
    */
    public int addNum(int numA, int numB) {
        return numA + numB;
    }

    /**
     * This is the main method which
     * uses the addNum method.
     * @param args Unused.
     * @return Nothing.
     * @exception IOException On input error.
     * @see IOException
    */
    public static void main(String args[])
        throws IOException {
        AddNum obj = new AddNum();
        int sum = obj.addNum(10, 20);
        System.out.println("Sum of 10 and 20 is :" + sum);
    }
}

```

To process the *AddNum.java* with the *javadoc* tool to produce the HTML documentation of the *AddNum* class, simply type the following command in a terminal :

### C.1. JAVA DOCUMENTATION ~~CONNEXES~~. LES COMMENTAIRES JAVADOC

```
javadoc AddNum.java
```

It will display the following messages on the terminal to indicate its progression.

```
Loading source file AddNum.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_51
Building tree for all the packages and classes...
Generating /AddNum.html...
AddNum.java:36: warning - @return tag cannot be used
    in method with void return type.
Generating /package-frame.html...
Generating /package-summary.html...
Generating /package-tree.html...
Generating /constant-values.html...
Building index for all the packages and classes...
Generating /overview-tree.html...
Generating /index-all.html...
Generating /deprecated-list.html...
Building index for all classes...
Generating /allclasses-frame.html...
Generating /allclasses-noframe.html...
Generating /index.html...
Generating /help-doc.html...
1 warning
```

The complete documentation of the Javadoc tool can be found on Oracle's website at <http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/javadoc.html>.

# Annexe D

# De UML à Java

Le passage d'un diagramme de classe UML à un programme Java est, en général, relativement systématique. À tel point que de nombreux éditeurs de diagramme UML proposent une génération automatique du code Java (ou tout au moins du squelette du code : la définition des classes comprenant attributs et entêtes de méthodes) directement à partir d'un diagramme de classe.

Les autres diagrammes UML permettent, eux aussi, de générer du code automatiquement et de compléter les squelettes générés depuis le diagramme de classe.

Dans cette annexe nous nous concentrerons sur le passage du diagramme de classe vers le code Java et n'aborderons pas la génération depuis les autres diagrammes UML.

## D.1 Les classes

Les différents éléments d'une classe qui doivent figurer dans sa description en Java figurent explicitement dans le diagramme de classe UML de la classe en question.

En effet, le diagramme de classe permet d'identifier :

- les variables d'instance,
- les variables de classe,
- les signatures des méthodes d'instance,
- les signatures des méthodes de classe,
- la visibilité des méthodes et des variables,
- l'héritage entre les classes.

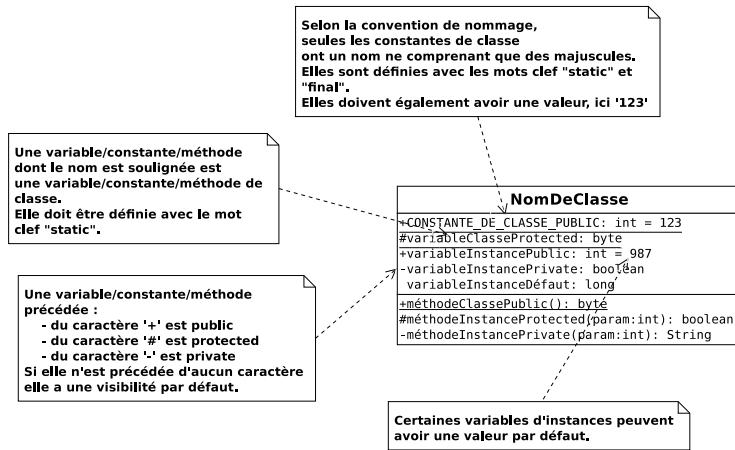


FIGURE D.1 – Une classe UML

Par exemple, le diagramme de classe de la figure D.1 se traduit directement par le code Java suivant :

```
public class NomDeClasse {
    // Constantes et variables de classe (static)
    public static final int CONSTANTE_DE_CLASSE_PUBLIC=123;
    protected static byte variableClasseProtected;

    // Variables d'instance
    // Certaines variables peuvent être initialisées
    public int variableInstancePublic=987;
    private boolean variableInstancePrivate;
    long variableInstanceDéfaut;

    // Méthodes de classe
    public static byte méthodeClassePublic(){};

    // Méthodes d'instance
    protected boolean méthodeInstanceProtected(float param){};
    private String méthodeInstancePrivate(int param){};
}
```

### D.1.1 Classes et méthodes abstraites

La classe du diagramme de la figure D.2 est abstraite et elle définit une méthode abstraite.

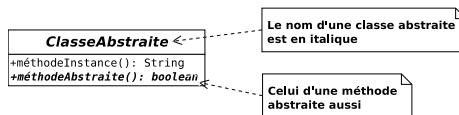


FIGURE D.2 – Une classe abstraite

Le code Java qui la représente serait alors :

```
public abstract class ClasseAbstraite {
    public String méthodeInstance(){};

    public abstract boolean méthodeAbstraite();
}
```

## D.2 Les associations entre classes

### D.3 L'héritage

Le diagramme de classes ci-dessous représente un héritage entre la classe *Personne* et la classe *Etudiant*. La classe *Etudiant* est la sous-classe, tandis que la classe *Personne* est la sur-classe.

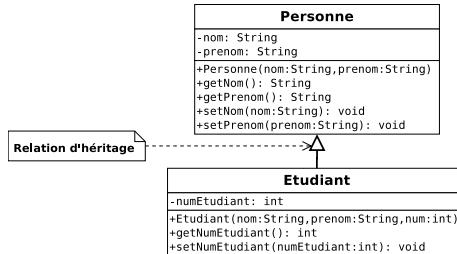


FIGURE D.3 – L'héritage avec UML

Ce diagramme de classe correspond aux deux classes Java *Personne* et *Etudiant* ci-dessous. Tout d'abord la sur-classe, la classe *Personne* :

```
public class Personne {
    private String nom;
    private String prenom;

    public Personne(String nom, String prenom){}
    public String getNom(){}
    public String getPrenom(){}
    public void setNom(String nom){}
    public void setPrenom(String prenom){}
}
```

et la sous-classe, la classe *Etudiant* :

```

public class Etudiant extends Personne {
    private int numEtudiant;

    public Etudiant(String nom, String prenom, int num){}
    public int getNumEtudiant(){}
    public void setNumEtudiant(){}
}

```

## D.4 Les paquetages

Le diagramme de classe suivant représente deux classes en relation l'une avec l'autre à travers une association. Chacune des classes est dans un paquetage différent et donc les classes Java qui les représenteront devront se trouver dans des répertoires différents.

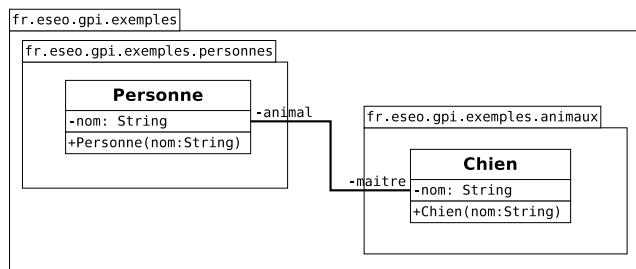


FIGURE D.4 – Des paquetages et des classes

La classe *Personne* est dans le paquetage *fr.eseo.gpi.exemples.personnes*, elle doit donc se trouver dans un répertoire .../fr/eseo/gpi/exemples/personnes son code source :

```

package fr.eseo.gpi.exemples.personnes;

public class Personne {
    private String nom;
    private Chien animal;

    public Personne(String nom){}
}

```

La classe *Chien* est dans le paquetage *fr.eseo.gpi.exemples.animaux*, elle doit donc se trouver dans un répertoire .../fr/eseo/gpi/exemples/animaux son code source est :

```
package fr.eseo.gpi.examples.animaux;

public class Chien {
    private String nom;
    private Personne maitre;

    public Chien(String nom){}
}
```

## D.5 Les interfaces

Le diagramme UML ci-dessous décrit les interfaces *UneInterface* et *AutreInterface* ainsi que la classe *MaClasse* qui les implémente.

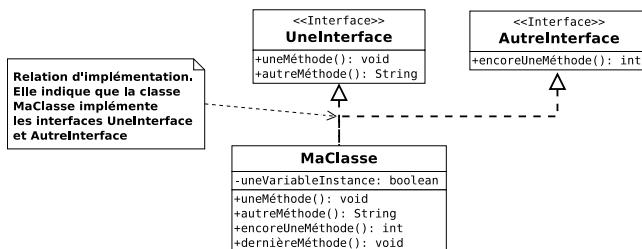


FIGURE D.5 – Une interface et son implémentation

En Java, ce diagramme est représenté par les deux interfaces suivantes :

```
public interface UneInterface {
    public void uneMéthode();
    public String autreMéthode();
}
```

et :

```
public interface AutreInterface {
    public int encoreUneMéthode();
}
```

et par la classe :

```
public class MaClasse implements UneInterface,
                                AutreInterface {
    public void uneMéthode(){}
    public String autreMéthode(){}
    public int encoreUneMéthode(){}
    public void dernièreMéthode(){}
}
```

# Bibliographie

- [1] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 3(3) :147–148, March 1968.
- [2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification (Java SE 8 edition)*. Oracle, java se 8 edition, 2014.
- [3] James Gosling and Henry McGilton. *The Java Language Environment : A White Paper*. Sun Microsystems Computer Company, 1995.