Support de cours des modules Algorithmique et programmation (Java) de première année

Filières EURINSA et AMERINSA de l'INSA Lyon

Table des matières

1 Préface à l'attention des enseignants ou des étudiants sachant déjà programmer

En EURINSA et AMERINSA première année, les choix pédagogiques suivants ont été faits :

— 1 seul return par méthode (en 1A uniquement). Les codes rendus avec plusieurs return dans une même méthode seront comptés comme contenant une erreur.
— `switch` et `do/while` non présentés. Usage accepté.
— `break` : interdit en 1A (sauf dans les switch)
— Les opérateurs arithmétique courts (+=, -=, /=, ...) ne sont pas présentés, sauf : ++
— Seuls 4 types primitifs sont au programme : int, double, boolean, char (jamais de float)
— jamais rien en static à part le main (en 1A)
— L'encapsulation est stricte : Tous les attributs sont obligatoirement en `private` et les méthodes peuvent être en `public` ou `private`. Les autres cas sont considérés comme des erreurs.
— Les accolades sont toujours présentes, même si une seule instruction (en 1A uniquement et seulement pour les enseignants. Les étudiants ont le droit d'alléger s'ils savent)
— Choix pertinent des boucles (modifier le compteur d'un for pour faire un break est symptomatique d'un mauvais choix)
— Initialisation des attributs, prioritairement dans le constructeur et non pas sur la déclaration. Cette consigne est surtout pour les enseignants. Les étudiants, eux, peuvent faire comme ils veulent.

Ce support de cours couvre l'intégralité du programme de première année EURINSA/AMERINSA, mais intègre également des éléments hors programme à titre informatif. Si des choses restent peu claire après lecture des explications, n'hésitez pas à prévenir :

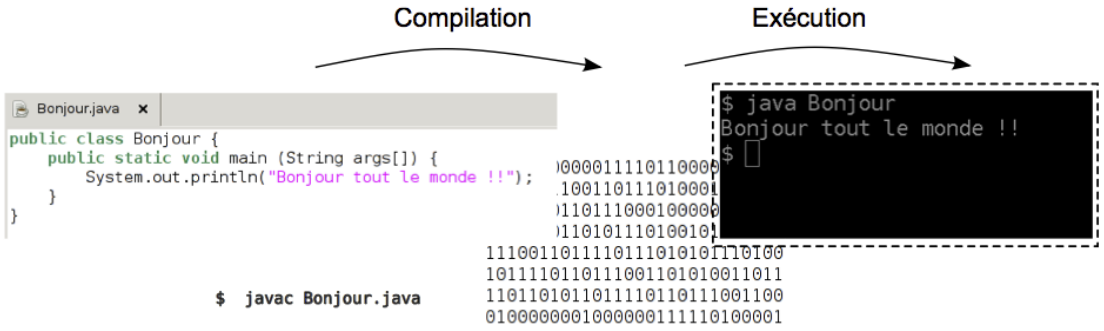
nicolas.stouls@insa-lyon.fr

2 Notions élémentaires

2.1 Qu'est-ce qu'un programme ?

L'objectif de la programmation est de créer des logiciels ou programmes. Ceux-ci sont constitués d'un ensemble de traitements qui permettent de transformer des données numériques (les entrées) en d'autres données numériques (les sorties). Les données de sortie peuvent être affichées sous une forme graphique (avec des fenêtres comme le fond les programmes tels que Word et Excel) ou plus simplement affichées dans une console* [1] sous forme de texte.

Que se passe-t-il pour l'ordinateur lorsqu'on exécute un programme ? Il va lire le fichier exécutable du programme comme une suite de 0 et de 1 (codage binaire) et exécuter l'une après l'autre les instructions ainsi codées. Cette suite de 0 et de 1 est appelée langage machine et est directement exécutable par le micro-processeur de l'ordinateur. Or il est très difficile pour un humain de programmer directement en langage machine, c'est pourquoi on utilise un langage de programmation écrit en langage textuel dans un fichier source (fichier .java). Le fichier source est ensuite compilé* en langage binaire (fichier .class) puis exécuté pour réaliser les traitements :



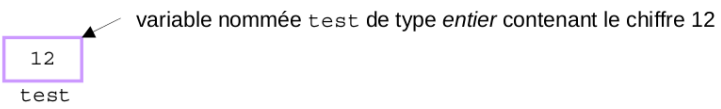
Pour compiler un fichier .java en binaire, on utilise la commande `javac` (avec le 'c' de compilateur), tandis que l'on utilise la commande `java` pour exécuter un programme Java.

Commande	Rôle	action	Exemple
javac	compilation	Traduit un fichier .java en fichier .class	javac MaClasse.java
java	exécution	Exécute un fichier .class	java MaClasse

Pour mémoire, certains outils comme `Geany` appellent ces commandes à votre place.

2.2 Les variables

Les variables constituent l'aspect le plus important de la programmation, puisqu'elles permettent de stocker dans un emplacement mémoire les données et de les transformer en utilisant des opérateurs*. On peut se représenter une variable comme une étiquette associée à une unique boîte dans laquelle est rangée une valeur d'un certain type (entier ou réel, par exemple) :



Avant de pouvoir utiliser une variable, il est nécessaire de la déclarer*, c'est-à-dire d'associer la variable à un emplacement de la mémoire et de spécifier son type. C'est en fonction du type de la variable que la taille de l'emplacement mémoire (en octet, soit 8 bits) et le codage binaire de la valeur seront déterminés. La déclaration se fait par une instruction de la forme :

1. Le symbole * indique que le mot est défini dans la section Glossaire.

```
typeVariable nomVariable;
```

La variable `nomVariable` est de type `typeVariable`. Il lui est associé un emplacement mémoire qui contient une valeur qu'il faut initialiser*. Cela se fait en utilisant l'opérateur d'affectation "=" :

```
nomVariable = uneValeur;
```

L'affectation* écrit dans l'emplacement mémoire associé à `nomVariable` la valeur `uneValeur`. On dit alors que `nomVariable` *"prend la valeur"* `uneValeur`.

`uneValeur` doit être compatible avec le type de `nomVariable`. Par exemple si `typeVariable` est un entier, `uneValeur` doit être une valeur entière. On peut faire ici le parallèle entre le type d'une variable et l'unité de mesure d'une grandeur physique qui spécifie la nature de la grandeur manipulée.

On peut également faire la déclaration et l'initialisation d'une variable en une seule instruction :

```
typeVariable nomVariable = uneValeur;
```

Dans la suite, nous allons détailler les quatre types dits "primitifs"*, qui sont directement accessibles en mémoire (contrairement aux types non primitifs que nous verrons dans la section 3), c'est-à-dire que la valeur affectée à la variable est stockée dans l'emplacement mémoire associée à celle-ci. Pour chacun de ces types, nous verrons également les opérateurs que l'on peut appliquer sur des variables de ce type pour transformer leur valeur.

2.2.1 Les types primitifs

On distingue 4 catégories de types primitifs (entier, réel, booléens, caractères). L'intervalle de valeurs représentables pour chacun des types peut varier en fonction de l'espace mémoire qu'ils occupent.

Les entiers En Java, tous les types permettant de représenter des entiers sont signés. Ainsi, sur n bits, on peut coder les entiers de $-(2^{n-1})$ à $2^{n-1}-1$. Les valeurs négatives sont encodées en complément à 2. Pour un rappel sur cet encodage, nous vous renvoyons au cours de numération disponible sur la page du CIPC.

Les différents types d'entiers sont les suivants :

Nom	Taille	Intervalle représentable
`byte`	1 octet	$[-128\dots127]$ ou $[-2^7\dots2^7-1]$
`short`	2 octets	$[-32768\dots32767]$ $[-2^{15}\dots2^{15}-1]$
`int`	4 octets	$[-2^{31}\dots2^{31}-1]$
`long`	8 octets	$[-2^{63}\dots2^{63}-1]$

On peut appliquer des opérateurs arithmétiques (+,-,*,/) à deux variables ou deux expressions de type entier (la composition étant possible comme en mathématiques). Le résultat de cette opération est également du type entier (ce qui permet la composition).

Lorsque les deux opérandes sont de type entier, l'opérateur / calcule la division entière et l'opérateur % calcule le reste de cette division.

Par exemple [2] :

```
int valA = 7;
int valB = 2;
int valC = valA / valB;      // valC contient la valeur 3
int valD = valA % valB;      // valD contient la valeur 1
```

2. // indique que la suite de la ligne est un commentaire.

LES RÉELS La description du codage des réels est décrite dans le cours de numération sur la page du CIPC. En Java il existe deux types de représentation pour les nombres réels : simple et double précision (respectivement les types `float` et `double`).

Nom	Taille	Intervalle représentable
`float`	4 octets	$[-3.4 \cdot 10^{38}, \cdots, -1.4 \cdot 10^{-45}, 0, 1.4 \cdot 10^{-45}, \cdots, 3.4 \cdot 10^{38}]$
`double`	8 octets	$[-1.8 \cdot 10^{308}, \cdots, -4.9 \cdot 10^{-324}, 0, 4.9 \cdot 10^{-324}, \cdots, 1.8 \cdot 10^{308}]$

Lorsqu'au moins une des opérandes est de type réel, l'opérateur / calcule la division réelle.

```
double reelA = 7;
double reelB = 2;
double division = reelA / reelB; //La variable division contient la valeur 3.5
```

LES BOOLÉENS Un type de variable très utile en informatique est le type booléen, qui prend deux valeurs VRAI ou FAUX.

Nom	Taille	Intervalle représentable
`boolean`	1 octet	[true,false]

On peut appliquer des opérateurs logiques (**et, ou, non**) à des variables ou des expressions booléennes. Le résultat de cette opération est également du type booléen.

- && désigne l'opérateur **et** logique
- || désigne l'opérateur **ou** logique
- ! désigne l'opérateur **non** logique (qui transforme une valeur `true` en valeur `false` et inversement)

a	b	a **et** b
Faux	Faux	Faux
Faux	Vrai	Faux
Vrai	Faux	Faux
Vrai	Vrai	Vrai

a	b	a **ou** b
Faux	Faux	Faux
Faux	Vrai	Vrai
Vrai	Faux	Vrai
Vrai	Vrai	Vrai

```
boolean boolA = TRUE;
boolean boolB = FALSE;
boolean nonA = !boolA;          // nonA vaut FALSE
boolean AetB = boolA && boolB; // AetB vaut FALSE
```

LES CARACTÈRES Le type caractère peut correspondre à n'importe quel symbole du clavier (lettre en majuscule ou minuscule, chiffre, ponctuation et symboles).

Nom	Taille	Intervalle représentable
`char`	2 octets	[a...z,A...Z,0...9, :, ;, . ? !...]

Pour distinguer la valeur correspondant au caractère a de la variable dénommée `a`, on utilise l'apostrophe pour la première.

```
char carac = 'a'; // La variable carac contient la lettre a
int a = 3;            // La variable a contient la valeur 3
```

Comme toute donnée numérique, un caractère est encodé sous forme d'une suite de 0 et de 1 que l'on peut interpréter comme un entier non signé. Dans certains contextes, il est utile de manipuler directement ce code. On convertit alors la variable de type `char` en une variable de type `int` comme expliqué ci-dessous.

2.2.2 Conversion entre types primitifs (*type casting*)

La conversion (également appelée *transtypage* ou *type casting*) d'un type primitif en un autre se fait de la manière suivante :

```
typeA variableA = (typeA) valeurB;
```

Si `variableA` et `valeurB` ne sont pas du même type, alors c'est la valeur `valeurB` convertie en `typeA` qui est affectée à `variableA`.

Voici l'action réalisée par les principales conversions :

Entier vers Réel	la même valeur codée en réel
Réel vers Entier	la partie entière du réel
Entier vers caractère	le caractère dont le code ASCII est l'entier
Caractère vers Entier	le code numérique correspondant au caractère

```
int i;
double x = 2;
i = (int) (x * 42.3); // i vaut 84
```

2.2.3 Opérateurs de comparaison

Les opérateurs de comparaison permettent de comparer deux variables d'un même type primitif (entier, flottant, booléen et caractère) et renvoient une valeur booléenne (*vrai* ou *faux*) :

==	comparaison d'égalité
!=	différence
<	inférieur strict
<=	inférieur ou égal (s'écrit comme il se prononce)
>	supérieur strict
>=	supérieur ou égal (s'écrit comme il se prononce)

Attention ! L'opérateur = est l'affectation et l'opérateur == est à la comparaison d'égalité, c'est-à-dire le signe = utilisé en mathématiques ! !

Nous vous recommandons de parenthéser complètement les expressions composées :

```
double a = 8;
boolean estDansIntervalle = ((a >= 0) && (a <= 10));
                          // vaut true ssi a appartient à [0,10]
```

2.3 Le programme

De manière générale, la structure d'un programme simple est toujours la même. Cette structure de base doit être apprise par cœur, car elle constitue le squelette du programme. Il est conseillé, lors de la création d'un programme, de commencer par écrire cette structure. En effet, une fois cette structure créée, le programme est fonctionnel : il peut être compilé* et exécuté*. Bien entendu à ce stade, le programme ne fait strictement rien puisqu'il n'y a aucune instruction, seulement des commentaires.

```
public class Exemple{ //Exemple est le nom du programme
                      // écrit dans le fichier Exemple.java
    public static void main (String[] args){
        //bloc d'instructions du programme
        //exécuté lors du lancement du programme
    }
}
```

Déclare et initialise deux variables celsius et fahrenheit, fahrenheit étant calculée à partir de celsius, pour ensuite les afficher à l'écran.

```
public class ConversionCelsiusVersFahrenheit{
    public static void main (String[] args){
        double celsius = 12.0;
        double fahrenheit = ((9.0/5.0) * celsius) + 32.0;
        System.out.print(celsius);
        System.out.print(" degrés Celsius convertit en Fahrenheit vaut ");
        System.out.println(fahrenheit);
    }
}
```

L'instruction `System.out.print` permet d'afficher la valeur d'une variable de type primitif ou un texte délimité par des guillemets.

2.4 Les structures de contrôle

Le principe d'un programme est de modifier le contenu des variables à l'aide des instructions élémentaires que nous venons de voir (affectation et opérateurs). Or, nous pouvons vouloir que ces instructions ne soient réalisées que dans certains cas, ou bien nous pouvons vouloir répéter l'exécution de ces instructions. Ce sont les structures de contrôle qui permettent de spécifier si l'exécution d'un traitement est conditionnée ou bien si elle se fait de manière répétée.

2.4.1 Bloc d'instructions

Les accolades {} permettent de délimiter un bloc d'instructions, c'est-à-dire un ensemble d'instructions qui vont être exécutées les unes à la suite des autres. Un bloc d'instructions peut, par exemple, être exécuté que lorsqu'une condition est vérifiée ou bien il peut être exécuté plusieurs fois de suite. Ce sont les structures de contrôle conditionnelles et itératives qui permettent d'exprimer cela. Les variables déclarées* dans un bloc sont accessibles à l'intérieur de ce bloc uniquement.

Deux variables de même nom peuvent être déclarées dans deux blocs distincts. Ce sont deux variables différentes associées à deux emplacements mémoires différents. Ainsi, leurs valeurs sont généralement différentes. Confondre l'une avec l'autre peut être une source d'erreur de programmation.

2.4.2 Structures conditionnelles

Les structures de contrôle conditionnelles permettent de spécifier à quelles conditions un bloc d'instructions va être exécuté. Cette condition est exprimée par une expression logique.

La structure alternative Le premier type de conditionnelle s'écrit comme suit :

```
if (condition) {// équivalent à (condition == true)
    // bloc d'instructions exécutées si condition est vraie
} else {
    // bloc d'instructions exécutées si condition est fausse
}
```

Cette structure de contrôle exprime une alternative. Or, il est possible de vouloir qu'un bloc soit exécuté sous une certaine condition et que sinon, aucune instruction ne soit exécutée. Dans ce cas, la clause `else` et son bloc sont supprimés. Les parenthèses autour de `condition`, qui est variable ou une expression à valeur booléenne, sont obligatoires.

Affiche un message si la température est supérieure à 50.

```
public class QuelleUnite{
    public static void main (String[] args){
        int temperature = 36;
        if(temperature > 50) {
            System.out.println("La température est probablement en Fahrenheit");
        }
    }
}
```

LA STRUCTURE CHOIX MULTIPLES Le second type de conditionnelle permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. Sa syntaxe est la suivante :

```
switch (variable) {
    case valeur1 :
        // Liste d'instructions exécutées si (variable == valeur1)
        break;
    case valeur2 :
        // Liste d'instructions exécutées si (variable == valeur2)
        break;
    ...
    case valeurN :
        // Liste d'instructions exécutées si (variable == valeurN)
        break;

    default:
        // Liste d'instructions exécutées sinon
}
```

Le mot clé `default` précède la liste d'instructions qui sont exécutées lorsque `variable` a une valeur différentes de `valeur1,..,valeurN`. Le mot clé `break` indique que la liste d'instructions est terminée.

2.4.3 STRUCTURES ITÉRATIVES

Il existe 3 formes de structure itérative, chacune a un cadre d'utilisation bien spécifique que nous allons voir.

L'ITÉRATION RÉPÉTÉE *n* FOIS La première forme itérative est la boucle `for`. Elle permet de répéter un bloc d'instructions un nombre de fois fixé. Dans sa syntaxe, il faut déclarer et initialiser la variable qui sert de compteur de tours de boucle, indiquer la condition sur le compteur pour laquelle la boucle s'arrête et enfin donner l'instruction qui incrémente* ou décrémente* le compteur :

```
for (int compteur = 0 ; compteur < n ; compteur = compteur + 1) {
   // bloc instructions répétées n fois
}
```

ou

```
for (int compteur = n ; compteur > 0 ; compteur = compteur - 1) {
   // bloc instructions répétées n fois
}
```

Affiche la conversion en Fahrenheit des degrés Celsius de 0 à 39.

```
public class ConversionCelsiusVersFahrenheit{
    public static void main (String[] args){
        for(int celsius = 0; celsius < 40; celsius = celsius + 1) {
            double fahrenheit = ((9.0/5.0) * celsius) + 32.0;
            System.out.print(celsius);
            System.out.print(" degres Celsius convertit en Fahrenheit vaut ");
            System.out.println(fahrenheit);
        }
    }
}
```

La boucle `for` s'utilise lorsque l'on connaît *a priori* le nombre de répétitions à effectuer.

L'ITÉRATION RÉPÉTÉE TANT QU'UNE CONDITION EST VRAIE La seconde forme d'itérative est la boucle `while`. Elle exécute le bloc d'instructions tant que la `condition` est vraie. Le bloc peut ne jamais être exécuté. La syntaxe est la suivante :

```
while (condition) { // équivalent à (condition == true)
   // bloc d'instructions répétées tant que condition est vraie.
   // condition doit être modifiée dans ce bloc
}
```

Cette structure exécute le bloc d'instructions *tant que* la condition est réalisée (*while* en anglais).

Il est important de toujours s'assurer que la condition deviendra fausse lors d'une itération de la structure itérative. Dans le cas contraire, l'exécution du programme ne s'arrêtera jamais.

Affiche la conversion en Fahrenheit des degrés Celsius tant que la conversion est inférieur à 100.

```
public class ConversionCelsiusVersFahrenheit{
    public static void main (String[] args){
        int celsius = 0;
        double fahrenheit = ((9.0/5.0) * celsius) + 32.0;
        while(fahrenheit < 100) {
            System.out.print(celsius);
            System.out.print(" degres Celsius convertit en Fahrenheit vaut ");
            System.out.println(fahrenheit);
            celsius = celsius + 1;
            fahrenheit = ((9.0/5.0) * celsius) + 32.0;
        }
    }
}
```

La boucle `while` s'utilise lorsque le nombre d'itérations n'est pas connu a priori mais peut s'exprimer au moyen d'une expression à valeur booléenne qui devient fausse lorsque la répétition doit s'arrêter.

L'ITÉRATION EXÉCUTÉE AU MOINS UNE FOIS La troisième forme d'itérative est la boucle "`do while`". C'est une variante de la boucle `while`, où la condition d'arrêt est testée après que les instructions ont été exécutées :

```
do {
    // bloc d'instructions exécutées
    // condition doit être modifiée dans ce bloc
} while (condition); // si condition est vraie,
                     // le bloc est exécuté à nouveau
```

Ne pas oublier le ; après la condition d'arrêt. Le bloc d'instructions est exécuté au moins une fois.

Affiche la conversion en Fahrenheit des degrés Celsius jusqu'à ce que le degré Fahrenheit soit supérieur ou égale à 100.

```
public class ConversionCelsiusVersFahrenheit{
    public static void main (String[] args){
        double fahrenheit;
        int celsius = 0;
        do {
            fahrenheit = ((9.0/5.0) * celsius) + 32.0;
            System.out.print(celsius);
            System.out.print(" degres Celsius convertit en Fahrenheit vaut ");
            System.out.println(fahrenheit);
            celsius = celsius + 1;
        } while (fahrenheit < 100);
    }
}
```

2.4.4 EXEMPLE D'ALGORITHME SIMPLE À CONNAÎTRE

Le calcul de la valeur d'une série est un exemple de calcul où vous ne devriez pas avoir besoin de réfléchir.

Par exemple, considérons la série $\sum_{i=0}^{N} \left(\frac{1}{2}\right)^i$. Elle contient une somme sur un intervalle donné : ce sera un `for` sur cet intervalle.

Pour ce qui est du terme général $\left(\frac{1}{2}\right)^i$, il sera dans le `for`. Mais il va nécessiter de mémoriser le

résultat intermédiaire. Par exemple, si je devais calculer à la main $\sum_{i=0}^{4}\left(\frac{1}{2}\right)^i$, je poserai :

$$\left(\frac{1}{2}\right)^0+\left(\frac{1}{2}\right)^1+\left(\frac{1}{2}\right)^2+\left(\frac{1}{2}\right)^3+\left(\frac{1}{2}\right)^4$$

Ensuite, pour calculer la valeur finale, je procéderais par étapes :

$$\begin{aligned}&=1+\left(\frac{1}{2}\right)^1+\left(\frac{1}{2}\right)^2+\left(\frac{1}{2}\right)^3+\left(\frac{1}{2}\right)^4\\&=\frac{3}{2}+\left(\frac{1}{2}\right)^2+\left(\frac{1}{2}\right)^3+\left(\frac{1}{2}\right)^4\\&=\frac{7}{4}+\left(\frac{1}{2}\right)^3+\left(\frac{1}{2}\right)^4\\&=\frac{15}{8}+\left(\frac{1}{2}\right)^4\\&=\frac{31}{16}\end{aligned}$$

Le programme va faire pareil. Le terme le plus à gauche de le calcul manuel doit être stocké dans une variable (on l'appelle un accumulateur) qui contient à tout moment la somme des i premiers termes. En fin de boucle, elle contient donc la somme des termes.

Concrètement, voici un exemple d'implémentation de cette série :

```
public class ExempleSerie{
    public static void main (String[] args){
        final int N = 4;
        double tmp = 0; // Accumulateur initialisé avec l'élément neutre de l'addition

        for(int i=0 ; i<=N ; i++) {
            tmp= tmp + Math.pow(1/2.0,i);
        }

        System.out.println("Valeur de cette série pour N valant "+N+" : "+tmp);
    }
}
```

L'un des points importants à retenir est que lorsque l'on fait un calcul via une boucle, il est très souvent nécessaire d'introduire une variable en plus qui devra contenir le résultat intermédiaire (et le résultat final à la fin de l'exécution de la boucle). Comme cette variable (cet accumulateur) doit conserver sa valeur d'une itération à l'autre et que l'on veut connaitre sa valeur après la sortie de la boucle, elle doit être déclarée avant la boucle (et initialisée avec une valeur neutre pour ne pas fausser le premier calcul).

2.5 Utiliser des méthodes

2.5.1 Principes

En mathématique, on connait la notion de fonction : une expression algébrique à laquelle on associe un nom, et généralement un ou des paramètres.

En Java, la notion de **méthode** est similaire [3] : c'est un bloc d'instructions auquel on associe un nom et éventuellement un ou des paramètres. Une méthode peut être exécutées par un simple appel* de son nom depuis le bloc du programme principal (méthode `main`) ou depuis une autre méthode.

La création d'une méthode est abordée en section 3.2.4.

En Java, il existe de nombreuses méthodes prédéfinies. La plus connue est sans doute la méthode suivante qui permet d'afficher une chaîne de caractères à l'écran :

```
System.out.println("la chaîne de caractères à afficher");
```

D'autres exemples de méthodes que vous pouvez utiliser sont celles de la librairie `Math` : `sqrt, cos, sin, abs, etc..`

De même que pour les fonctions en mathématiques, pour appeler une méthode, on utilise son nom suivi de la liste de ses paramètres effectifs (séparés par une virgule) entre parenthèses :

```
nomMethode(parametre_1,... , parametre_n);
```

3. Nous verrons dans la section sur l'orienté objet la notion d'attribut, qui justifie qu'une méthode n'est pas seulement une fonction.

Si cette méthode renvoie un résultat, il est possible de l'affecter à une variable de type compatible. Les différentes méthodes du systèmes sont très nombreuses et parfois homonymes. Il est souvent nécessaire de préciser que fourni la méthode. Par exemple, pour appeler la méthode de racine carrée (`sqrt`), fournie par la classe `Math` du système on peut faire :

```
double racine = Math.random(5.2);
```

2.5.2 Exemple à connaitre : la génération de valeurs aléatoires

Lorsque l'on a besoin d'avoir un comportement différent d'une exécution à l'autre, alors on peut avoir besoin de faire un tirage aléatoire. L'unique méthode à connaître pour faire cela est la méthode `random` fournie par la classe `Math`. Cette méthode renvoie toujours un flottant double précision (un réel) compris dans l'intervalle $[0..1[$. Charge à vous d'adapter cette spécification à vos besoins.

Exemples les plus courants :

— choisir un entier dans $[n..m]$
— choisir un caractère dans $[c_1..c_2]$
— choisir un booléen $[true..false]$

Vous devez savoir faire ces tirages aléatoire sans avoir à réfléchir longuement.

Par exemple, pour faire un tirage dans l'intervalle entier $[n..m]$ on fera :

```
int tirage = (int)(Math.random()*(m-n+1)+n);
```

Où $m-n+1$ correspond au nombre d'éléments voulus dans l'intervalle d'arrivée et n est la borne minimum. Exemple d'un tirage dans l'intervalle $[4..8]$, qui commence à 4 et qui contient 5 éléments $(8-4+1)$:

```
int tirage = (int)(Math.random()*5+4);
```

Si l'on sait faire un tel tirage, et que l'on a compris la conversion de type (`int` vers `char`), alors le tirage d'une lettre minuscule pourra s'écrire :

```
char lettre = (char)(Math.random()*('z'-'a'+1)+'a');
```

Pour ce qui est du tirage d'un boolean, c'est beaucoup plus facile, puisque l'on veut simplement considérer 2 cas. Il suffira donc d'écrire :

```
boolean bool = Math.random()<0.5;
```

2.6 Transmettre des informations à un programme

Il est possible qu'un programme attende des informations de l'utilisateur pour s'exécuter. Il y a principalement deux manières de faire :

— Paramètre ligne de commande
— Lecture clavier

2.6.1 Exécution paramétrée du programme (Paramètre ligne de commande)

Dans la définition :

```
public static void main(String[] args) { ... }
```

le paramètre `args` représente les paramètres fournis sur la ligne d'appel.

Ainsi, lorsqu'un utilisateur appel un programme avec des paramètres, comme par exemple :

```
$ java MonPorgramme 42 toto
```

alors le paramètre `args` du `main` est un tableau [4] constitué de 2 cases, contenant les chaines de caractères "42" et "toto". Si l'on veut convertir "42" en 42, alors on fera une conversion String vers int, comme décrit en section 3.4 :

```
public static void main(String[] args) {
    int val = Integer.parseInt(args[0]);
    System.out.println("Le premier paramètre est la valeur entière "+val);
}
```

2.6.2 Lecture d'informations au clavier

Il est possible, à l'exécution, de demander à l'utilisateur de saisir une information à clavier, ce qui permet de réaliser une exécution interactive.

De même que le flux de sortie standard (la console) est `System.out`, le flux d'entrée standard (le clavier) est `System.in`.

On utilisera le type `Scanner`, disponible dans la bibliothèque `java.util` pour faciliter la lecture des informations sur ce flux d'entrée. Ses primitives `nextInt()`, `nextDouble()` ou `nextLine()`, par exemple, permettent de lire au clavier un entier, un flottant ou une chaine de caractères.

Exemple :

```
import java.util.Scanner;
public class .... {

    public void maMethode() {
        //Déclaration d'un scanner, outil qui sert à récupérer ce que l'utilisateur tape au
            clavier
        Scanner clavier = new Scanner(System.in);

        System.out.println("Saisissez un entier :");
        int n = clavier.nextInt();
        System.out.println("Vous avez donné : "+n);

        System.out.println("Saisissez un nombre réel, avec une partie décimale différente de
            0 :");
        double x = clavier.nextDouble();
        System.out.println("Vous avez donné : "+x);

        //Prendre une autre instance de Scanner pour utiliser nextLine()
        Scanner secondClavier = new Scanner(System.in);
        System.out.println("Saisissez une phrase : ");
        String s = secondClavier.nextLine();
        System.out.println("Vous avez donné : "+s);
    }
}
```

Attention ! Si votre programme lit une alternance de chaines de caractères et de valeurs numériques, alors Il est recommandé d'utiliser un `Scanner` différent pour chaque type lu.

Exemple de lecture a 2 scanners :

```
import java.util.Scanner;
public class ClasseDeSaisies {

    public static void main(String[] a) {
        //Déclaration d'un scanner, outil qui sert à récupérer ce que l'utilisateur tape au
            clavier
        Scanner clavierNum = new Scanner(System.in);
        Scanner clavierString = new Scanner(System.in);
```

4. Cette structure de données est décrite en section 4

```
        System.out.println("Combien de saisies nom/valeur voulez vous faire ?");
        int n = clavierNum.nextInt();
        String[] noms = new String[n];
        double[] vals = new double[n];

        for(int i=0 ; i<n ; i++) {
            System.out.println("Saisissez un nom : ");
            noms[i] = clavierString.nextLine();

            System.out.println("Saisissez une valeur réelle :");
            vals[i] = clavierNum.nextDouble();
        }


        System.out.println();
        System.out.println("Valeurs saisies :");
        for(int i=0 ; i<n ; i++) {
            System.out.println("  * "+noms[i]+" : "+vals[i]);
        }
    }
}
```

3 PROGRAMMATION ORIENTÉE OBJETS

L'objectif de la programmation orientée objet est de pouvoir stocker un ensemble de données et d'actions sémantiquement liées dans une unique variable. On définira donc la notion d'objet comme un reflet des objets de la vie réelle. Ces types sont *non-primitifs*.

Exemple :

Nous souhaitons réaliser un programme permettant d'écrire des courriers de manière automatique aux abonnés d'une bibliothèque. Pour chaque abonné, nous voulons mémoriser son nom, prénom, le nombre de volumes empruntés ainsi que le nombre de jours écoulés depuis l'emprunt.

Pour atteindre cet objectif, nous devrons manipuler des centaines d'abonnés, chacun décrit par 4 valeurs (nom, prénom, nb livres et durée emprunt). Ce type de traitement est fastidieux en n'utilisant que des types primitifs et des tableaux (Section 4).

Dans un tel cas, il serait plus simple de définir la notion d'abonné (avec ses 4 caractéristiques), puis de gérer un tableau d'abonnés. Le type `Abonne` sera un moule permettant de créer autant d'abonnés différents que l'on veut, mais avec toujours la même structure.

3.1 UTILISATION D'UNE VARIABLE OBJET

Considérons qu'il existe déjà un type `Abonne`. Un exemple de définition de ce type est présent en section 3.2. Voici un petit exemple, montrant une utilisation de ce type. On crée ici deux instances `a1` et `a2`, matérialisant 2 abonnés dont on veut savoir s'ils ont du retard sur leurs retours de livres :

```
public class UtiliseAbonne {
    public static void main(String[] a) {
        // Déclaration, instanciation et initialisation de 2 abonnés
        Abonne a1 = new Abonne("Le rouge", "John", 4, 2);
        Abonne a2 = new Abonne("Boulix", "Jojo", 3, 42);

        // Affichage, pour chacun, de sa description et s'il est en retard.
        System.out.println(a1.toString());
        boolean res = a1.estEnRetard();
        System.out.println("   Est il en retard ?" + res);

        System.out.println(a2.toString());
        res = a2.estEnRetard();
        System.out.println("   Est il en retard ?" + res);
    }
}
```

Une fois que l'on est d'accord sur le fait que les variables `a1` et `a2` sont de type `Abonne` et qu'elles contiennent chacune un nom, un prénom, un nombre de livres empruntés et un nombre de jours depuis emprunt, alors on peut se concentrer sur les éléments de langage. Il y a principalement 2 choses dans

cet exemple : (i) comment créer un nouvel abonné et (ii) comment interagir avec un abonné. Ces deux éléments sont respectivement discutés dans les 2 sections suivantes : instanciation et appel d'une méthode.

Le type d'un objet est défini par une classe. Une classe est un type.

Pour utiliser un objet, il est nécessaire que la classe définissant son type fasse partie de la bibliothèque standard Java ou bien qu'elle soit dans le même dossier que la classe utilisatrice.

INSTANCIATION Pour utiliser un objet, il est nécessaire de commencer par l'*instancier** (Le créer). Instancier un objet consiste à réserver un espace mémoire pour le stocker. Comme pour les tableaux, cela se fait avec le mot clef `new`.

```
//déclaration et instanciation d'une variable objet
TypeObjet maVariable = new TypeObjet(...);
```

Le mot clef `new` est suivi du nom du type, suivi de parenthèses, pouvant contenir des valeurs. Cela correspond à un appel au **constructeur** de ce type (Section 3.2.3). Le constructeur a pour rôle d'initialiser l'objet créé, notamment avec les valeurs transmises. Pour appeler un constructeur, il est nécessaire de savoir quels paramètres sont attendus. Si nous ne sommes pas les auteurs de la classe utilisée, alors il faudra généralement lire sa documentation (la javadoc).

Si nous voulons créer un abonné en fournissant son nom, son prénom, le nombre de livres empruntés et le nombre de jour de l'emprunt, alors une syntaxe de création d'un tel abonné serait idéalement :

```
Abonne a1 = new Abonne("Le rouge", "John", 4, 2);
```

Pour que cela soit possible, il nous faudra simplement définir un constructeur de cette forme dans la classe `Abonne`. Cela est fait en section 3.2.3.

APPEL D'UNE MÉTHODE Une fois un objet instancié, il est possible d'interagir avec lui. Tout objet fourni des actions, appelées méthodes (parfois appelées par abus de langage *fonctions* ou *procédures*).

Pour appeler une méthode fournie par un objet, on utilisera la syntaxe suivante, appelée *notation pointée* :

```
maVariableObjet.maMethode(...);
```

Cela revient à dire que l'on appelle la méthode `maMethode` fournie par l'objet `maVariableObjet`. Dire qu'une variable fournie une méthode est un abus de langage signifiant qu'une variable est d'un certain type et que ce type décrit cette méthode.

Par exemple, pour savoir si un abonné a du retard sur le rendu de ses livres, on peut écrire :

```
boolean res = a1.estEnRetard();
System.out.println(a1.getNom()+" est il/elle en retard sur le rendu des livres ?" + res);
```

Un tel appel exécute cette méthode dans le contexte de l'objet. Ainsi, suivant l'objet à gauche du point, le résultat ne sera pas le même. Concrètement, suivant que l'on appelle `a1.estEnRetard()` ou `a2.estEnRetard()`, alors la réponse sera relative à l'abonné `a1` ou l'abonné `a2`. En revanche, la méthode `estEnRetard()` n'aura été écrite qu'une seule fois, dans la classe `Abonne`

Notez que si les variables ont des noms communs comme identifiants et que les méthodes ont des verbes comme identifiants, alors cette notation pointée se lit comme une phrase où le sujet est l'objet, le verbe est la méthode et les compléments sont les paramètres.

Par exemple :

```
boolean res = a1.estEnRetard();
```

Pourra se lire :
Je stocke dans res le fait que a1 est, ou n'est pas, en retard.

3.2 Création d'un nouveau type / d'une nouvelle classe

Une classe est un type. Créer une nouvelle classe, c'est créer un nouveau type. Pour que ce type soit utile, il est nécessaire qu'il fournisse au moins des données (appelées `attributs`), ou des actions (appelées `méthodes`). Si une classe contient des attributs, alors il est fortement recommandé qu'elle contienne au moins un `constructeur`, dont le rôle est d'initialiser les `attributs`.

Voici un exemple complet de la classe `Abonne` utilisée dans les exemples de la section précédente. Nous l'utiliserons maintenant pour illustrer cette section :

```
/** Cette classe permet de matérialiser la notion d'abonné. Un abonné est défini par
 *  4 propriétés : nom, prénom, nombre de livres empruntés et durée de cet emprunt.
 */
public class Abonne {
    //les attributs de la classe Abonne
    private String nom;
    private String prenom;
    private int nbLivres;
    private int nbJours;

    // Un constructeur pour initialiser les variables
    public Abonne(String leNom, String lePrenom, int nbL, int nbJ) {
        nom =  leNom;
        prenom = lePrenom;
        nbLivres = nbL;
        nbJours = nbJ;
    }

    /** Renvoie vrai ssi l'abonné est en retard */
    public boolean estEnRetard() {
        boolean retard = false;
        if(nbJoursEmprunts > 21 && nbLivresEmpruntes > 0) {
            retard = true;
        } else {
            retard = false;
        }
        return retard;
    }

    /** Renvoie une description textuelle de l'objet */
    public String toString() {
        String res = "Abonné "+ nom+" "+prenom;
        res = res + " " + nbLivres + " livres empruntés, depuis "+nbJours+ " jours.";
        return res;
    }
}
```

3.2.1 Les classes

Les classes permettent de définir des types d'objets. Elles contiennent la description des objets, c'est-à-dire qu'elles définissent les attributs et méthodes communs aux objets d'un certain type.

On pourra prendre la métaphore du moule et des objets : Une classe est un moule à partir duquel on pourra créer plusieurs objets ayant des caractéristiques similaires. Par exemple, une classe définissant ce qu'est un abonné pourra être utilisée pour créer plusieurs abonnés, ayant chacun un nom différent, un prénom différent, ...

Les objets créés à partir d'une classe sont appelés des *instances* de cette classe. Ainsi, un abonné particulier est une instance de la classe `Abonne`.

Vous connaissez déjà la syntaxe générale de déclaration d'une classe, vous n'aviez simplement pas conscience qu'il pouvait y avoir plus d'éléments dedans :

```
public class NomType {
    // Le corps de la classe vient ici, avec :
    // * les attributs
    // * les constructeurs
```

```
    // * les méthodes
}
```

3.2.2 Les attributs

Un attribut est une variable globale à toute la classe. Il matérialise une propriété/une information que tout objet de ce type doit avoir. Dans l'exemple de l'abonné, 4 attributs sont déclarés :

```
public class Abonne {
    //les attributs de la classe Abonne
    private String nom;
    private String prenom;
    private int nbLivres;
    private int nbJours;

    // ...
}
```

Cela signifie que chaque abonné aura un nom, un prénom, un nombre de livres et un nombre de jours. Mais ces éléments ne sont pas partagés entre les abonnés. Chaque abonné a les siens.

Par convention, les attributs sont toujours déclarés en `private`. Cela signifie qu'ils ne sont pas accessibles depuis l'extérieur de cette classe. Cette convention est appelée *encapsulation*⋆.

3.2.3 Le constructeur

Toute classe contenant des attributs doit permettre de les initialiser. Elle doit donc fournir au moins un *constructeur*. Son rôle est d'initialiser les attributs. Il est appelé lorsque l'on instancie (crée) un nouvel objet de ce type. Si l'on considère l'exemple suivant d'instanciation d'un abonné :

```
Abonne a1 = new Abonne("Le rouge", "John", 4, 2);
```

alors le mot clef **`new`** suivi du nom de la classe matérialise l'appel au constructeur de la classe `Abonne`, rappelé ci-après :

```
    // Constructeur paramétré
    public Abonne(String leNom, String lePrenom, int nbL, int nbJ) {
        nom =  leNom;
        prenom = lePrenom;
        nbLivres = nbL;
        nbJours = nbJ;
    }
```

On voit la concordance entre la déclaration du constructeur et son appel. La déclaration attend 4 paramètres (`leNom`, `lePrenom`, `nbL` et `nbJ`) qui sont fournis par l'appel (`"Le rouge"`, `"John"`, 4 et 2). Ce constructeur est assez typique de la forme la plus courante d'un constructeur : tous les attributs sont initialisés par des valeurs reçues en paramètre d'appel.

La plupart du temps, on souhaite utiliser le même nom pour les paramètres et les attributs représentant la même chose. Cependant, cela génère un problème d'homonymie (Ex : `nom=nom;`). Pour résoudre ce problème il est possible d'utiliser le mot clef `this`. Dans l'exemple ci-dessous, l'expression `this.nom = nom;` signifie que l'attribut `nom` de l'objet en cours de construction (**`this`**`.nom`) prend la valeur du paramètre formel `nom`.

```
public class Abonne {
    //les attributs de la classe Abonne
    private String nom;
    private String prenom;
    private int nbLivres;
    private int nbJours;

    public Abonne(String nom, String prenom, int nbLivres, int nbJours) {
```

```
        // Constructeur paramétré où tes les paramètres sont homonymes des attributs
        this.nom =  nom;
        this.prenom = prenom;
        this.nbLivres = nbLivres;
        this.nbJours = nbJours;
    }
    //...
}
```

Syntaxiquement, un constructeur se déclare comme une méthode portant le **même nom** que la classe, mais n'ayant **pas de type de retour**.

3.2.4 Les méthodes

Exemples de méthodes Avant d'expliquer ce que sont les méthodes, voici 3 exemples de méthodes qui pourraient être définies dans une classe `Abonne` :

```
public class Abonne {
    //les attributs et constructeurs de la classe Abonne
    // ...

    /** Méthode qui renvoie vrai ssi l'abonné est en retard */
    public boolean estEnRetard() {
        boolean retard = false;
        if(nbJoursEmprunts > 21 && nbLivresEmpruntes > 0) {
            retard = true;
        } else {
            retard = false;
        }
        return retard;
    }

    /** Méthode qui modifie le nombre de jours d'emprunt.
        La valeur reçue en paramètre est le nouveau nombre de
        jour d'emprunt. Si cette valeur est négative, la durée d'emprunt
        est mise à 0. */
    public void setNbJour(int val) {
        nbJoursEmprunts=val;
        if(nbJoursEmprunts<0) {
            nbJoursEmprunts=0;
        }
    }

    /** Renvoie une description textuelle de l'objet */
    public String toString() {
        String res = "Abonné "+ nom+" "+prenom;
        res = res + " " + nbLivres + " livres empruntés, depuis "+nbJours+ " jours.";
        return res;
    }
}
```

Qu'est-ce qu'une méthode ? Une méthode est un bloc d'instructions auquel on associe un nom, à l'image d'une fonction en mathématiques. Dans l'exemple de la classe `Abonne`, 2 méthodes sont définies : `estEnRetard` et `toString`. Une méthode peut être exécutées par un simple appel* de son nom depuis le bloc du programme principal (méthode `main`) ou depuis une autre méthode. L'exemple de la classe `UtiliseAbonne`, présenté dans la section précédente, montre des exemples d'appels des méthodes d'`Abonne`.

Pourquoi des méthodes ? On est amené à créer une méthode principalement dans trois cas de figure :

- Pour permettre d'interagir avec un objet.
- Pour réunir un ensemble d'instructions qui participent à la réalisation d'une même tâche, et leur donner un nom. Cela permet de rendre le programme plus lisible et compréhensible par une autre personne (ou vous-même lors du TP suivant) et de faciliter la mise au point du programme (correction et tests).
- Pour factoriser du code. Lorsque un même groupe d'instructions est présent à plusieurs endroits du programme, cela permet de ne les écrire qu'une seule fois.

Le rôle d'une méthode est de traiter des données. Cela signifie qu'en général, la méthode effectue un traitement à partir des données qui entrent, et renvoie un résultat.

DÉFINITION D'UNE MÉTHODE Une méthode est définie dans une classe (pas dans une autre méthode), selon la syntaxe générale suivante :

```
public TypeRetour nomMethode(Type1 param1,..., TypeN paramN) {
    //bloc d'instructions
    return valeurRetournee;
}
```

La première ligne est appelée la signature et l'intérieur du bloc est le corps de la méthode (Figure 1).

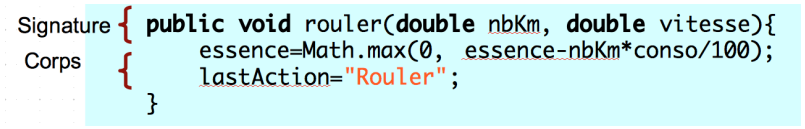


FIGURE 1 – Mise en évidence de ce que l'on appelle la signature et le corps d'une méthode

Afin de simplifier les usages futurs, tant que la notion de visibilité des méthode ne sera pas maitrisée, nous considérerons que toute méthode est déclarée en visibilité `public`. `TypeRetour` est le type de la valeur renvoyée par la méthode (ici, la variable `valeurRetournee`).

Si la méthode ne renvoie aucune valeur, le mot-clé `void` est utilisé à la place du type de la valeur retournée et le mot clef `return` ne doit pas être utilisé.

Il y a souvent confusion entre le fait de **retourner** une valeur (synonyme : renvoyer) et le fait de l'**afficher**. Donc :

Renvoyer/retourner Une méthode qui renvoie un résultat le transmet à son appelant. Typiquement, en mathématiques les fonctions renvoient toutes des résultats. On aura tendance à écrire : `double x = Mat.sin(0);` pour calculer sinus de 0 (en appelant la fonction sin fournie par la classe Math avec le paramètre 0) et stocker son résultat **renvoyé** dans x.

Afficher Une méthode affiche un message sur le terminal. Il est très rare qu'une méthode de calcul affiche son résultat. Si je veux pouvoir calculer $2+sin(x^2)$, il est nécessaire que l'élévation au carré, l'addition le calcul du sinus me **renvoient** tous les trois le résultat au lieu de l'afficher. Il me sera toujours possible, après, d'afficher la valeur reçue avec `System.out.println(..)`

Les paramètres correspondent aux données d'entrée de la méthode. Au niveau de la déclaration, les paramètres sont dits "formels" : ce sont des variables qui représentent chaque donnée d'entrée. On peut faire ici l'analogie avec les fonctions en mathématiques :

Une fonction mathématique $f : x \rightarrow 2 \times x + 1$ s'écrirait en java :

```
public int f(int x) {
    int res = x*2+1;
    return res;
}
```

À noter, que de manière strictement équivalente à l'exemple précédent, la méthode `estEnRetard` aurait pu s'écrire comme suit :

```
public boolean estEnRetard() {
    return nbJoursEmprunts > 21 && nbLivresEmpruntes > 0
}
```

APPEL D'UNE MÉTHODE C'est au moment de l'appel de la méthode que les paramètres formels sont initialisés, c'est-à-dire qu'une valeur leur est affectée. Les paramètres "effectifs"* de l'appel, ceux passés en argument de la méthode au moment de l'appel, sont affectés aux paramètres formels de la méthode (ceux de la définition de la méthode) par position : la valeur du premier paramètre effectif est affectée

au premier paramètre formel, et ainsi de suite. Les paramètres effectifs peuvent être des valeurs ou des variables.

```
public static void main (String[] args) {
    // Déclaration, instanciation et initialisation d'un abonné
    Abonne a = new Abonne("Le rouge", "John", 3, 2);

    // Appel des méthodes
    a.setNbJour(42);
    System.out.println(a.toString());
    boolean res = a.estEnRetard();
    System.out.println("   Est il en retard ?" + res);
}
```

En Java, le passage des paramètres se fait par valeur, c'est-à-dire que la valeur du paramètre effectif est affectée au paramètre formel. Ainsi, si la valeur du paramètre formel est modifiée dans le bloc de la méthode, cette modification est locale à la méthode mais n'est pas repercutée dans le contexte appelant.

3.2.5 Exemple : mise en évidence du passage de paramètres par valeur

Dans l'exemple suivant, la classe `Calculs` est un type ne contenant pas d'information, mais seulement un main et une méthode (addition). En compilant et en exécutant ce code, on va lancer l'exécution de la méthode `main`. Celle-ci va créer une instance de `Calculs`, faire un appel à la méthode `addition` avec 7 et 3 en paramètres, et afficher le résultat obtenu. On voit que la méthode addition modifie la valeur d'un paramètre reçu (entierA), que cette modification est perceptible dans le résultat affiché (11), mais que cette modification n'a pas affecté le paramètre fourni (a) qui vaut toujours 7.

```
public class Calculs {
    // S'il n'y a pas d'attribut, il n'est pas nécessaire de mettre de constructeur.
    public int addition(int entierA, int entierB) {
        entierA = entierA + 1;   //entierA est modifié
        return entierA + entierB;
    }

    public static void main (String[] args) {
        Calculs c = new Calculs();
        int a = 7;
        int b = 3;
        int somme = c.addition(a,b); // la valeur de a est affectée à entierA
        System.out.println("Somme = "+somme); // affiche 11  (7+1+3)
        System.out.println("a = "+a); // affiche 7 (et non pas 8)
    }
}
```

3.2.6 Encore un exemple : températures

L'exemple suivant présente un programme avec deux classes (chacune devant être stockée dans un fichier différent). L'une des deux matérialise une température (`Temperature`), tandis que l'autre (`UtiliseTemperature`) fait des manipulations de températures.

Lors de l'instanciation d'une `Temperature`, le constructeur prend une température en Celsius et stocke la valeur en attribut. Différentes méthodes permettent ensuite de récupérer la température, dans différentes unités.

Deux méthodes complémentaires `fromFahrenheit` et `fromKelvin` permettent de faire une conversion F ver C ou K vers C. Elles sont présentes pour améliorer la lisibilité du constructeur.

Le programme principal `UtiliseTemperature` considère des températures entières, de 0 à 39°C inclues. Une fois sur deux, sa conversion en degré Fahrenheit est affichée, l'autre fois c'est sa conversion en degré Kelvin qui est affichée. Notez le changement de valeur de la variable `calculeFahrenheit` à chaque tour de boucle.

```java
public class Temperature {
    private double tempC ; // température mémorisée en Celsius;

    // 2 constructeurs : par défaut on ne prend qu'une température en Celsius.
    // Le second constructeur laisse le choix
    public Temperature(double tempCelsius) {
        tempC = tempCelsius;
    }

    public Temperature(double temp, String unit) {
        if(unit.equals("F")) {
            tempC = fromFahrenheit(temp);

        } else if(unit.equals("K")) {
            tempC = fromKelvin(temp);

        } else { // Par défaut, c'est en Celsius
            tempC = temp;
        }
    }

    // quelques convertisseurs utilisés pour le constructeur
    public double fromFahrenheit(double f) {
        return (f-32)*5/9.0;
    }

    public double fromKelvin(double k) {
        return k-273.15;
    }


    // Différents getters pour récupérer la température dans l'unité désirée
    public double getCelsius() {
        return tempC;
    }

    public double getFahrenheit() {
        double fahrenheit = ((9.0/5.0) * tempC) + 32.0;
        return fahrenheit;
    }
    public double getKelvin() {
        double kelvin = 273.15 + tempC;
        return kelvin;
    }
}


public class UtiliseTemperature {
    public static void main (String[] args) {
        boolean calculeFahrenheit = true;
        for(int celsius = 0; celsius < 40; celsius = celsius + 1) {
            Temperature t = new Temperature(celsius);
            if(calculeFahrenheit) {
                System.out.println(celsius+"°C = "+t.getFahrenheit()+"F");
            } else {
                System.out.println(celsius+"°C = "+t.getKelvin()+"K");
            }
            calculeFahrenheit = !calculeFahrenheit;
        }
    }
}
```

3.2.7 Ressources complémentaires

— Vidéo de 8' de rappel des bases sur les objets :
`https://videos.insa-lyon.net/videos/?video=MEDIA171005135309172`

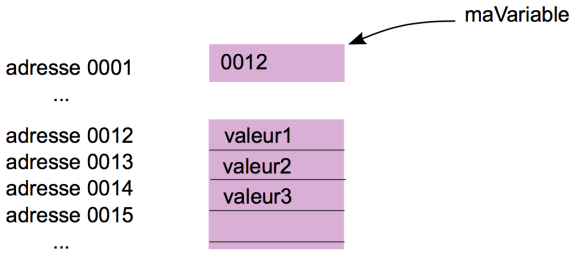
3.3 Connaissances et bonnes pratiques autour de l'orienté objet

3.3.1 Les types non primitifs

Dans cette section nous décrivons les mécanismes sous-jacents à tout objet Java : la façon dont un objet est stocké en mémoire et la manière dont on l'instancie* ; c'est-à-dire la façon dont on réserve l'emplacement mémoire nécessaire à l'objet.

Stockage des objets en mémoire Tout comme les variables de type primitif, une variable de type `objet` est associée à un emplacement mémoire de taille fixe qui contient une unique valeur. Dans cet

emplacement mémoire est stocké une valeur de type `adresse` qui indique l'adresse* de l'emplacement mémoire où sont stockées, de manière contiguë, toutes les valeurs de l'objet. Ainsi, la variable est liée aux données de manière indirecte : elle contient l'adresse à laquelle on peut trouver les données. La variable qui est manipulée est en fait une référence* à l'emplacement mémoire où se trouve l'ensemble des données.



INSTANCIATION* DES OBJETS L'instanciation d'un objet se fait à l'aide du mot clé `new` qui "réserve" l'emplacement mémoire nécessaire pour stocker toutes les valeurs de l'objet, c'est-à-dire un ensemble de cases mémoire contiguës, et renvoie l'adresse de la première case mémoire.

```
//déclaration et instanciation
TypeObjet maVariable = new TypeObjet;
```

COMPARAISON DE DEUX OBJETS Lorsque l'on compare deux objets, à l'aide des opérateurs == ou !=, ce sont les adresses des objets qui sont comparées. Ainsi, l'opérateur == renvoie la valeur `true` si et seulement si les deux variables font référence au même emplacement mémoire, donc au même objet. On parlera d'*égalité référentielle*. La comparaison d'*égalité sémantique* (ou `structurelle`) est celle liée à l'usage de la méthode `equals`, discutée en section 3.3.8.

```
if(variableA == variableB){
    System.out.println("Les deux variables font référence au même objet.");
}
```

MOT CLEF NULL Si une variable d'un type non primitif n'est pas initialisée, alors elle contient une adresse mémoire particulière, qui n'a pas le droit d'être allouée. Cela permet d'identifier le cas particulier d'un objet non initialisé. Cette adresse est matérialisée par le mot clef `null`. On pourra donc tester si un objet a été initialisé via :

```
// Si non initialisé
if(variable==null) {
    // on instancie.
    variable = new SonType();
}
```

Cette comparaison est faite via « == » et non pas un appel à `.equals(...)`, car c'est l'égalité de l'adresse et de `null` que l'on teste.

3.3.2 MOTS CLEFS PACKAGE ET IMPORT

Lorsqu'un est constitué d'un grand nombre de classes, on organise généralement les fichiers en plusieurs dossiers, formant une arborescence. En Java la notion de dossier s'appelle un paquetage et se déclare avec le mot clef `package`. Si l'on considère qu'une classe `LiaisonPivot` est prévue pour être stockée dans le dossier *donnees/modelemecanique/lisaisons*, alors on dira qu'elle est dans le paquetage :

`donnees.modelemecanique.lisaisons`

et l'entête du fichier java devront être :

```
package donnees.modelemecanique.lisaisons;
public class LiaisonPivot {
```

```
    // ...
```

Par convention, les noms de paquetage s'écrit tout en minuscule, sans barre de soulignement.

Pour utiliser une classe qui se trouve dans un autre paquetage du même programme, il est nécessaire de préciser son chemin d'accès (son paquetage). Ainsi, depuis la fenêtre de conception, on pourra parler du type `LiaisonPivot` en précisant son chemin d'accès :

```
    donnees.modelemecanique.lisaisons.LiaisonPivot lp;
    lp = new donnees.modelemecanique.lisaisons.LiaisonPivot();
```

Afin d'alléger l'écriture des programmes lorsque l'on utilise plusieurs fois une même classes, il est possible de l'importer :

```
    import donnees.modelemecanique.lisaisons.LiaisonPivot;
    public class .... {
        public void maMethode() {
            donnees.modelemecanique.lisaisons.LiaisonPivot lp;
            lp = new donnees.modelemecanique.lisaisons.LiaisonPivot();
        }
    // ...
    }
```

Si l'on sait que l'on va utiliser plusieurs classe d'un même paquetage, on peut choisir d'importer l'ensemble du dossier en une seule action, en remplaçant le nom de la classe par une étoile en fin de la ligne d'import :

```
import donnees.modelemecanique.lisaisons.*; // Import de toutes les liaisons
public class .... {
    // ...
}
```

La *bibliothèque** Java contenant un grand nombre de classes, elle est également structurée en paquetages. L'un d'entre eux, `java.lang` est appelée la bibliothèque standard. Toutes les classe qui y sont présentes n'ont pas besoin d'être importées (Ex : `Math`, `String`, `System`). Les classes des autres paquetages nécessites que l'on précise leur chemin ou qu'on les importe. Par exemple pour faire une lecture clavier on pourra faire :

```
public class .... {
    import java.util.Scanner;

    public void maMethode() {
        Scanner clavier = new Scanner(System.in);
        // ...
    }
```

ou bien :

```
public class .... {
    public void maMethode() {
        // System fait partie de la bibliothèque standard, contrairement à Scanner
        java.util.Scanner clavier = new java.util.Scanner(System.in);
        // ...
    }
```

3.3.3 Visibilité

La visibilité permet de désigner qui a le droit d'utiliser un objet, un attribut ou une méthode. Cela n'a de sens que dans un programme contenant plusieurs classes. En Java il existe 4 niveaux de visibilité,

que l'on peut associé à chaque classe, méthode ou attribut :

privé : l'élément n'est accessible que depuis la même classe

paquetage : l'élément n'est accessible que depuis la même classe ou le même paquetage

protégé : l'élément n'est accessible que depuis la même classe, le même paquetage ou les classes héritées

publique : l'élément est accessible à tous.

Pour définir ces visibilités, il n'existe que 3 mots clefs :
— `private` : visibilité privée
— `protected` : visibilité protégée
— `public` : visibilité publique
Pour définir qu'un élément a une visibilité paquetage, il suffit de ne pas préciser de visibilité.

Maintenant que l'on sait qu'il existe toute ces libertés, il existe des conventions dans la programmation orientée objet sur leur usage. La principale est le principe d'encapsulation (Section 3.3.4).

3.3.4 Principe de l'encapsulation

Le principe de l'encapsulation consiste à dire que l'on veut pouvoir penser en programme globalement, puis répartir les tâches entre plusieurs personnes, faisant chacune des choix d'implémentations. De même ces choix doivent pouvoir évoluer au court des cycles de maintenance du programme sans que cela ait de conséquence sur le reste du programme. Pour ce faire, le principe est de dire qu'aucune donnée (aucun attribut) n'est accessible de l'extérieur d'une classe. Seules des méthodes permettent de récupérer une information. L'utilisateur extérieur n'a pas à savoir quel choix a été fait pour représenter une information en interne. Le rôle des méthodes est alors de gommer ce choix.

Dans l'exemple de manipulation de températures présenté en section 3.2.6 et partiellement rappelé ci-dessous, le choix d'implémentation a été de ne mémoriser qu'une seule version de la température (en °C). Mais on aurait pu, par exemple, choisir de mémoriser 3 valeurs différentes, pour la température dans les 3 unités.

```
public class Temperature {
    // température mémorisée. Choix d'implémentation : en Celsius
    private double tempC ;

    // Constructeur
    public Temperature(double tempCelsius) {
        tempC = tempCelsius;
    }

    // 3 getters : de l'extérieur, on ne sait pas quel choix a été fait en interne
    public double getCelsius() {
        return tempC;
    }
    public double getFahrenheit() {
        double fahrenheit = ((9.0/5.0) * tempC) + 32.0;
        return fahrenheit;
    }
    public double getKelvin() {
        double kelvin = 273.15 + tempC;
        return kelvin;
    }
}
```

Ainsi, sauf cas particuliers, le principe d'encapsulation définit que :

> Tout attribut a une visibilité privée, tandis que les méthodes pourrons avoir accès à l'ensemble des niveaux de visibilité. On pourra définir des méthodes dont le seul rôle est de lire ou écrire une valeur dans un attribut. On appelle *accesseur* l'ensemble de ces méthodes, qui sont principalement de deux types : *getter*, pour lire une valeur, et *setter*, pour écrire une valeur.

Les getters et setters ne sont que des méthodes classique. Simplement, par convention, si l'on veut écrire une méthode permettant de modifier le nombre de jours d'emprunt d'un livre, alors on appellera cette méthode `setNbJourEmprunt` tandis qu'une méthode `getNom` permettra de récupérer l'information du nom.

3.3.5 SURCHARGE

Il est possible de définir, dans une même classe, plusieurs méthodes ayant le même nom, à condition qu'elles n'aient pas les mêmes paramètres. On parlera de surcharge. Par exemple, dans la classe suivante :

```
1  public class Temperature {
2      // température mémorisée. Choix d'implémentation : en Celsius
3      private double tempC ;
4
5      public void setTemp(double d) {
6          tempC = d;
7      }
8
9      // le paramètre t doit terminer par l'unité C ou F sur 1 caractère.
10     // Ex : setTemp("123F") ou setTemp("18C")
11     public void setTemp(String t) {
12         String val = Double.parseDouble(t.substring(0,t.length-1));
13         if(t.charAt(t.length-1) == 'C') {
14             setTemp(val);
15         } else {
16             setTemp((val-32)/1.8);
17         }
18     }
19 }
```

La méthode `setTemp` est surchargée, car elle est définie 2 fois. La première fois pour un paramètre flottant et la seconde fois pour un paramètre sous forme de chaine de caractères. On pourra également dire que ces 2 méthodes sont chainées, car l'une fait appel à l'autre (lignes 14 et 16).

Il est classique que l'on souhaite surcharger le constructeur d'une classe. Ainsi, en fonction des paramètres effectifs (nombre de paramètres ou types des paramètres) passés au constructeur au moment de l'instanciation, l'un ou l'autre des constructeurs sera appelé. Par exemple, l'instruction

```
new Abonne("Dupond", "Luc",4,0)
```

déclenchera l'exécution du premier constructeur de l'exemple ci-dessous (constructeur paramétré), tandis que l'instruction `new Abonne()` déclenchera l'exécution du second constructeur (constructeur par défaut).

```
public class Abonne {
    //les attributs de la classe Abonne
    private String nom;
    private String prenom;
    private int nbLivres;
    private int nbJours;

    // Constructeur paramétré où tes les paramètres sont homonymes des attributs
    public Abonne(String nom, String prenom, int nbLivres, int nbJours) {
        this.nom =  nom;
        this.prenom = prenom;
        this.nbLivres = nbLivres;
        this.nbJours = nbJours;
    }

    // Constructeur par défaut
    public Abonne() {
        this("", "", 0, 0); // appel chainé vers le constructeur paramétré
    }
}
```

Quand plusieurs constructeurs existent dans une même classe, il est recommandé de les chainer via `this(...)`, comme cela est fait dans le constructeur par défaut de l'exemple précédant. Cela est expliqué dans la section 3.3.6.

3.3.6 MOT CLEF THIS

Ce mot clef fait référence à l'objet courant. Il peut être utilisé de 2 manières : en tant que référence vers l'objet courant ou pour appeler un constructeur de l'objet courant.

L'usage en tant que référence le plus courant est pour lever les problèmes d'homonymie. Considérons le code suivant :

```
public class MaClasse {
    private double val ;

    public void setTemp(double val) {
        val=val; // Problème. On écrira plutôt : this.val=val;
    }
}
```

On voit que l'on a un problème pour parler en même temps d'un paramètre et d'un attribut qui porteraient le même nom. Si on utilise le nom de la variable, alors on parle de sa définition la plus locale. Donc du paramètre. À l'inverse, pour parler de l'attribut, on parlera de quelque chose qui fait partie de l'objet courant. Donc `this.val` sera un moyen de parler, explicitement, de l'attribut.

Le second usage possible de this est l'appel à un constructeur de l'objet courant. Cela n'est possible que depuis un autre constructeur du même objet, ce qui nécessite donc qu'il soit surchargé.

Pour résumer, voici les 2 usages du this :

— `this(...)` : seulement depuis un constructeur. Permet d'appeler un autre constructeur de la même classe (Donc le constructeur a été surchargé).

— `this.monAttribut` : Depuis n'importe quelle méthode d'une classe. En cas d'homonymie, cela permet de distinguer un attribut d'une variable locale (ou d'un paramètre).

Exemple :

```
public class MaClasse {
    private int i;

    MaClasse(int i) {
        this.i=i; // Résolution de l'homonymie (paramètre VS attribut)
    }
    MaClasse() {
        this(0);  // Appel de l'autre constructeur, avec la valeur
                  // par défaut : 0
    }
}
```

3.3.7 Méthode pré-définie toString

Tout objet contient la méthode :

```
public String toString()
```

Cette méthode a pour rôle de renvoyer une description textuelle de l'objet.

Cette méthode est parfois appelée implicitement par le système. Par exemple, si l'on fait la somme entre une chaine de caractère et un Objet, alors c'est la description textuelle de l'objet qui va être concaténée avec la chaine initiale. Le système appellera indirectement `toString`. Exemple :

```
// Ces deux lignes sont équivalentes :
String s = "La description est : "+monObjet; // appel implicite de toString par le système
String s = "La description est : "+monObjet.toString(); // appel explicite
```

À noter que si vous créez votre propre classe, mais que vous ne fournissez pas cette méthode, alors le système en fournit une à votre place. Mais cela ne fait généralement pas ce que vous voudriez.

Classiquement, une méthode `toString` devrait renvoyer une description de l'objet, incluant le contenu de ses différents attributs. Exemple typique :

```
public class MaClasse {
    private int attributA;
    private int attributB;
    private String attributC;
```

```
    public String toString(){
        return "MaClasse : "+attributA+"  /  "+attributB+"  /  "+attributC;
    }
}
```

Mais n'hésitez pas à écrire une phrase ayant du sens. Cela facilitera vos relectures de tests :

```
public class Voiture {
    private String marque;
    private int kilometrage;

    public String toString(){
        return "La voiture de marque "+marque+ " a "+kilometrage+"km au compteur.";
    }
}
```

Règle :

> Dans toute classe servant à stocker des données que vous écrivez, vous **devez** fournir une méthode `toString`, même si cela ne vous est pas directement demandé.

3.3.8 Méthode pré-définie equals

Comme vu en section 3.3.1, lorsque l'on compare deux objets, à l'aide des opérateurs == ou !=, ce sont les adresses des objets qui sont comparées (*égalité référentielle*).

Pour permettre une comparaison d'égalité *sémantique* (ou `structurelle`), il faut utiliser la méthode `equals` qui existe dans toutes les classes Java :

```
public class MaClasse {
    // ...
    public String equals(MaClasse o) {
        // ...
    }
}
```

Cette signature de méthode est strictement fausse, mais ce "mensonge" est utilisé en première année pour simplifier les explications. En deuxième année, nous voyons le polymorphisme et la vraie signature est présentée :

```
public String equals(Object o)
```

Exemple d'usage mettant en évidence la différence entre les égalités :

```
MaClasse a = new MaClasse();
MaClasse b = new MaClasse();
MaClasse c = b;
System.out.println("a et b sont égaux référentiellement ? "+(a==b));      // false
System.out.println("a et b sont égaux sémantiquement ?  "+a.equals(b)); // true
System.out.println("b et c sont égaux référentiellement ? "+(b==c));      // true
System.out.println("b et c sont égaux sémantiquement ?  "+b.equals(c)); // true
```

La forme classique d'une méthode `equals` est la suivante :

```
public class Vehicule {
    // Attributs
    private String marque ;
    private double consommation;
    private int kilometrage;

    public boolean equals(Vehicule v) {
```

```
        return marque.equals(v.marque) &&
               consommation == v.consommation &&
               kilometrage == v.kilometrage;
    }
}
```

Notons que si la méthode `equals` compare 2 attributs qui sont non-primitifs, alors elle va les comparer en appelant `equals`. Mais ce n'est pas la même méthode, car elle ne vient pas de la même classe. Dans cet exemple, la méthode `equals` de `Vehicule` fait un appel à la méthode `equals` de `String`.

On dit que l'égalité via `equals` est sémantique, car c'est le concepteur qui définit ce qu'est l'égalité. Ainsi, deux voitures pourraient être considérées comme égales si elle ont la même marque, le même modèle et le même kilométrage, même si elles n'ont pas la même immatriculation.

Comme pour `toString`, si on ne définit pas nous même une méthode `equals` dans nos classes, alors le système en génère une pour nous. Le comportement de la méthode générée est celui de l'égalité référentielle (==). Voici un exemple de méthode générée automatiquement par le système dans binaire produit si on n'en définie pas une nous même :

```
public class Vehicule {
    // ...

    public boolean equals(Vehicule v) {
        return this == v;
    }
}
```

3.3.9 Mot clef static

Le mot clé `static` peut être utilisé devant la déclaration d'un attribut ou d'une méthode. Placé devant un attribut, il indique que l'attribut n'appartient pas à une instance particulière de la classe, mais qu'il appartient à la classe elle-même. La zone mémoire associée est donc unique et partagée entre toutes les instances de ce type.

Par exemple, on peut avoir un attribut `nbAbonnes` qui compte le nombre d'`Abonne` qui ont été instanciés :

```
public class Abonne {
    //les attributs de la classe Abonne
    private static int nbAbonnes=0; // attribut static associé à la classe
    private String nom;
    // (...)
    public Abonne(String nom, String prenom, int nbLivres, int nbJours) {
        nbAbonnes = nbAbonnes+1; // Il y a un abonné de plus
        this.nom =  nom;
        // (...)
    }
}
```

Les méthodes statiques sont elles aussi indépendantes des objets. On peut les appeler en écrivant :

```
NomDeLaCLASSE.nomDeLaMethode(Paramètres effectifs)
```

ou simplement

```
nomDUneInstance.nomDeLaMethode(Paramètres effectifs)
```

Si la méthode est appelée à l'intérieure de la classe, il n'est pas nécessaire de préfixer le nom de la méthode. Les méthodes statiques, étant indépendantes de toute instance, n'ont pas accès aux variables ou méthodes non statiques.

Lorsqu'un programme Java est exécuté, il n'existe aucun objet, donc aucun objet auquel appliquer une méthode. C'est pour cela que les concepteurs du langage ont décidé que le corps d'un programme

serait constitué par une méthode statique : `main`.

Par exemple, si l'on souhaite calculer la durée moyenne des emprunts actuels, on créera une méthode `moyenneEmprunts` dont la déclaration est :

```
public class Abonne {
    //les attributs de la classe Abonne
    private static int nbAbonnes=0; // attribut static associé à la classe
    private static int sommeJours=0; // attribut static associé à la classe

    private String nom;
    // (...)

    public Abonne(String nom, String prenom, int nbLivres, int nbJours) {
        nbAbonnes = nbAbonnes+1;
        sommeJours = sommeJours+nbJours;

        this.nom=nom;
        // (...)
    }

    public static double moyenneEmprunts() {
        // Ne pas oublier la conversion en double pour ne pas faire une division entière
        return sommeJours/(double)nbAbonnes;
    }
}
```

3.3.10 Méthodes à paramètres de type non-primitif

En Java, le passage des paramètres à une méthode se fait par valeur. Dans le cas des types non-primitifs, cette valeur est une adresse. Ainsi, une méthode ne pourra pas modifier l'adresse de la variable, en revanche elle pourra modifier les valeurs stockées à cette adresse.

Instanciation des variables de type non-primitif dans une méthode L'instanciation d'un objet à l'aide du mot clé `new` réserve l'emplacement mémoire nécessaire au stockage des valeurs de l'objet et renvoie l'adresse. Ainsi, si cette instruction est effectuée dans une méthode, il faudra penser à renvoyer cette adresse pour pouvoir manipuler l'objet en dehors de la méthode.

Prenons l'exemple suivant où l'on souhaite dupliquer un tableau pour pourvoir le modifier tout en conservant l'original :

```
public int[] copierTableau(int[] tableau) {
    int[] copie = new int[tableau.length];
    for(int i = 0; i < tableau.length; i = i + 1) {
        copie[i] = tableau[i];
    }
    return copie;
}
```

Modification des valeurs d'une variable de type non-primitif dans une méthode Lorsqu'on passe un objet en paramètre d'une méthode, c'est l'adresse de celui-ci qui est affectée au paramètre formel de la méthode. Si la méthode modifie cette valeur (l'adresse), celle-ci ne sera pas modifiée en dehors (le contexte appelant) de la méthode. En revanche, lorsqu'on modifie les valeurs de l'objet, on change directement les valeurs dans les cases mémoires de l'objet et ce changement est alors définitif.

Prenons l'exemple suivant où l'on permute les valeurs des indices i et j du tableau (Les tableaux sont abordés en section 4) :

```
public void permute(int[] tab, int i, int j) {
    int temp = tab[i];
    tab[i] = tab[j];
    tab[j] = temp;
}

public void autreMethode (...) {
    int[] unTableau = {1,2,3,4,5};
    permute(unTableau,0,4);
    //unTableau contient maintenant les valeurs [5,2,3,4,1]
```

```
}
```

3.4 Le type prédéfini String

Il existe de nombreux types non-primitifs prédéfinis en Java. Ils sont répartis dans différents dossiers, que l'on appelle des *packages*. L'ensemble de ces dossiers est appelée la *bibliothèque*⋆ Java. Parmi les types pré-définis, celui le plus utilisé est le type `String`, permettant de stocker des chaînes de caractères.

Une chaîne de caractères représente un texte. C'est une collection linéaire de caractères qu'il n'est pas possible de modifier une fois que l'objet a été instancié (On dira que ce type est *immutable*⋆).

Ces opérations se font selon la syntaxe :

```
//Déclaration, instanciation et initialisation
String ecole = new String("INSA-Lyon"); // instanciation classique
String ecole2 = "INSA-Lyon"; // syntaxe expresse, valable uniquement pour les String
```

Notez la présence de guillemets doubles pour distinguer les chaînes de caractères des éléments du langage de programmation Java (mots du langage, nom de variables, nom de méthodes).

Méthodes principales Les objets de type String sont fournis avec un grand nombre de méthodes permettant de les manipuler facilement. Voici le détail des méthodes principales :

— `public char charAt(int n)` : cette méthode prend en paramètre un entier n et renvoie le $n+1$ ème caractère de la chaîne (le premier caractère est à l'indice 0 et le $n+1$ ème à l'indice n).

```
String ecole = "INSA-Lyon";
char c = ecole.charAt(2); //c contient le caractère 'S'
```

— `public int length()` : cette méthode renvoie la longueur de la chaîne de caractères.

```
String ecole = "INSA-Lyon";
int longueur = ecole.length(); // longueur contient 9
```

Notez les parenthèses de la méthode `length`, parenthèses absentes lorsque l'on appelle cette méthode sur un tableau.

— `public String substring(int debut, int fin)` : cette méthode renvoie un objet de type `String` qui contient la sous-chaîne de caractères commençant à l'indice `debut` et se terminant à l'indice `fin-1`.

```
String ecole = "INSA-Lyon";
String ville = ecole.substring(5, ecole.length());
//équivaut à ecole.substring(5, 9), ville contient "Lyon"
```

— `public equals(Object s)` : cette méthode renvoie une variable de type booléen qui vaut vrai si et seulement si la chaîne de caractères s reçue en paramètre est la même que la chaîne de caractères de l'objet sur lequel on appelle la méthode.

Attention aux optimisations faites par le compilateur Sachant que le type String est immutable, si deux constantes littérales identiques sont connues à la compilation, alors le compilateur n'instanciera qu'une seule chaine :

```
String tmp = "INSA";
String ecoleA = tmp+"-Lyon"; // "INSA-Lyon"
String ecoleB = "INSA-Lyon"; // "INSA-Lyon"
String ecoleC = "INSA-Lyon"; // "INSA-Lyon"
if(ecoleA.equals(ecoleB) == true){
    //cette condition est satisfaite
}
if(ecoleA == ecoleB){
    //cette condition n'est pas satisfaite,
    //les deux objets n'ayant pas la même adresse
}
if(ecoleB.equals(ecoleC) == true){
    //cette condition est satisfaite
}
if(ecoleB == ecoleC){
    //cette condition est satisfaite (optimisation du compilateur)
}
```

OPÉRATIONS SUR LES CHAÎNES ET CONVERSIONS DE TYPES Une opération fréquente consiste à "coller" bout-à-bout deux chaînes de caractères. On appelle cela la *concaténation*. L'opérateur associé est '+' :

```
String ville = "Lyon";
String ecole = "INSA";
String monEcole = ecole + "-" + ville; // monEcole contient "INSA-Lyon"
```

Cet opérateur '+' permet également de concaténer une chaine de caractère avec la représentation textuelle de n'importe quelle variable de type primitif :

```
int i = 42;
boolean b = true;
String question = "Alors ? ";
String resultat = question+b+" "+i; // resultat contient "Alors ? true 42"
```

Il peut être nécessaire de convertir un type primitif en une chaîne de caractères ou réciproquement. La conversion d'un type primitif en une chaîne de caractères se fait soit à l'aide de l'instruction `valueOf`, soit de l'opérateur '+' ;

```
int valeur = 22;
String chaine  = String.valueOf(valeur); // chaine  contient "22"
String chaine2 = ""+valeur;              // chaine2 contient "22"
```

La conversion d'une chaîne de caractères en un type primitif se fait à l'aide des instructions suivante :

```
Integer.parseInt(chaine);     // renvoie une valeur de type int
Double.parseDouble(chaine);   // renvoie une valeur de type double
Boolean.parseBoolean(chaine);// renvoie une valeur de type boolean
chaine.charAt(0);             // renvoie le premier caractère de la chaine
```

Exemple :

```
String chaine = "22" ;
int valeur = Integer.parseInt(chaine); //valeur vaut 22
```

Si la chaine de caractères interprétée ne représente pas un entier ("Bonjour" ou " 42" – avec un espace en trop), alors l'exécution s'arrête avec une exception (une erreur).

3.5 Questions/Réponses autours de l'orienté objet

Quelle différence entre attribut et paramètre ?

Un attribut est une propriété d'un objet, une variable liée à une instance. Syntaxiquement, cette variable se déclare dans une classe. Tandis qu'un paramètre est un argument d'appel d'une méthode, d'une fonction. Donc après l'appel de la méthode, le paramètre n'existe plus, tandis que l'attribut existera tant que l'objet existera.

Pourquoi ne peut on pas utiliser le même nom pour un attribut et un paramètre ?

C'est possible. Simplement, si 2 variables portent le même nom, alors il devient plus difficile pour choisir de laquelle on parle. Si on arrive dans ce cas de figure, alors la solution, en Java, consiste à préfixer le nom de l'attribut par `this.` lorsqu'il est utilisé. Se référer à la section du poly sur l'usage du `this.` pour plus d'exemples.

À quoi sert this ?

Ce mot clef a 2 rôles. Il permet soit d'appeler un constructeur de l'objet courant depuis un autre constructeur (On parle de chainer les constructeurs), soit d'accéder à des attributs de l'objet courant. Dans ce second cas, l'usage de this est facultatif s'il n'y a pas de variable homonyme définie à cet endroit. Plus d'informations peuvent être trouvées dans le poly, en section sur « Le mot clef this ».

Est-il possible d'avoir plus d'une classe dans un même programme Java ?

Un programme est presque toujours composé de plusieurs classes. Par contre, s'il est possible de déclarer plusieurs classes dans un même fichier, cela est déconseillé dans le temps de vos études au PC. Cela implique des contraintes de visibilité que nous préférons ne pas aborder dans le progression du PC.

Si on a une classe dont le constructeur attend plusieurs paramètres et que l'on voudrait n'en fournir que certains, alors comment peut on faire ? Par exemple : si le constructeur d'une classe Polar accepte 3 paramètres (titre, auteur, prix), mais que l'on veut créer un livre dont on ne connait pas l'auteur. Comment faire ?

Il est nécessaire de fournir une valeur pour tous les paramètres attendus. Si cela a du sens, alors il peut être possible de donner une valeur neutre. Par exemple, une chaine de caractères ne contenant pas de caractères : "". Ceci étant, si vous êtes l'auteur de la classe utilisée, alors vous avez la possibilité de surcharger le constructeur. C'est à dire d'en écrire plusieurs. Vous pourrez chercher la notion de surcharge dans le poly.

Quelle est la différence entre la méthode equals et le comparateur "==" ?

Si la variable est de type primitif, alors cela compare les valeurs, tandis que si la variable est d'un type non primitif, alors cela compare des adresses mémoire. Si la variable est un objet (donc un type non primitif), alors elle possède une méthode `equals`, qui permettra une comparaison sémantique des objets (basée sur le sens des objets).

Attention, dans le cas particulier des chaines de caractères, le compilateur peut faire des optimisations qui donnent une fausse impression de facilité. Voici un exemple mettant en évidence ce piège :

On peut également rajouter l'exemple suivant :

```
String s1 = "to";
s1=s1+s1;
String s2 = "toto";
String s3 = "toto";

System.out.println("s1 : "+s1);
System.out.println("s2 : "+s2);
System.out.println("s3 : "+s3);

if(s2 == s1) {
    System.out.println("Cas 1 : oui, ils sont égaux");
} else {
    System.out.println("Cas 1 : non, ils sont différents");
}

if(s2.equals(s1)) {
    System.out.println("Cas 2 : oui, ils sont égaux");
} else {
    System.out.println("Cas 2 : non, ils sont différents");
```

```
}

if(s2 == s3) {
    System.out.println("Cas 3 : oui, ils sont égaux");
} else {
    System.out.println("Cas 3 : non, ils sont différents");
}

if(s2.equals(s3)) {
    System.out.println("Cas 4 : oui, ils sont égaux");
} else {
    System.out.println("Cas 4 : non, ils sont différents");
}
```

Vous constaterez en exécutant ce code que s1, s2 et s3 contienne la même chaine de caractères, que les cas 2, 3 et 4 dirons que les chaines sont égales, mais que le cas 1 dira que les chaines sont différentes. Le cas 3 répond en parlant d'égalité par le fait d'une optimisation du compilateur.

Jusqu'à combien de paramètres peut on avoir dans un constructeur (ou une méthode) ?

Beaucoup. Aucune limite n'est officiellement donnée et j'ai arrêté de compté à 100. Mais faire un constructeur ou une méthode attendant plus de 5 ou 6 paramètres, nécessite à repenser au bienfondé de la structure de notre programme.

Qu'est-ce que cela signifie que le constructeur n'a pas de "type de retour " ?

C'est par opposition avec les méthodes. Le constructeur peut être vu comme une méthode particulière, mais portant le nom de la classe et n'ayant pas de type de retour.

Quelle différence entre String et char ?

Une variable de type char contient un et un seul caractère. À l'inverse, une variable de type String est une chaine de caractères. Concrètement, une variable de type String contient un tableau de char.

Quelle est la différence entre retourner et afficher ?

Afficher une valeur la montre sur le terminal pour que l'utilisateur la connaisse. C'est classiquement un appel à `System.out.println()`. Retourner une valeur est le résultat de l'appel d'une méthode. Si on veut faire un appel de la forme `int prix = montre.getPrix();` alors on s'attend à ce que la méthode getPrix() retourne une valeur qui serait le prix. Mais on ne s'attend généralement pas à ce qu'elle affiche quoi que ce soit à l'écran.

Quel est le rôle de la méthode toString() ?

Cette méthode sert à générer une chaine de caractères résumant une description de l'objet courant (l'ensemble des informations présentent dans ses attributs). Cette méthode est souvent utilisée dans le cas des classes ayant pour rôle de stocker des données. Si cette méthode n'est pas définie dans une classe, alors elle existera quand même. Le système en générera une par défaut pour vous. Mais la version par défaut n'est pas très utilisable par l'utilisateur (Ex : `MaClasse@FE45E32` est moins clair que « L'étudiant Benoit a eut la note de 17/20 en informatique. ») Astuce : la méthode `print/println` sait que tout objet contient cette méthode. Donc si on affiche un objet, alors print/println fera un appel implicite à `toString()` pour vous.

Est il nécessaire d'avoir un constructeur dans une classe ? Comment faire sinon ?

S'il n'y a pas d'attribut, il n'y a pas besoin d'un constructeur. En revanche, s'il y a des attributs, il est possible d'initialiser les attributs avec des valeurs constantes directement sur leur ligne de déclaration. Pour initialiser un attribut avec une valeur demandée par l'utilisateur, alors l'usage d'un constructeur est obligatoire.

Est-ce qu'on peut utiliser n'importe quel type comme paramètre dans des classes crées par nous mêmes ?

Oui.

Quelle est la différence entre les paramètres formels et les paramètres effectifs ?

Ces éléments sont définis dans le glossaire du poly.

Une méthode peut elle avoir plusieurs return ?

Oui, une méthode peut en contenir plusieurs. La seule obligation du langage est que tout exécution d'une méthode renvoyant un résultat doit se terminer avec un return. Cependant, en première année, on vous impose un exercice algorithmique simple : vous n'avez PAS le droit d'utiliser plusieurs return

dans une même méthode. Ce n'est pas une bonne pratique de programmation, mais surtout un moyen pédagogique de travailler des problèmes simples et de détecter des gens faussement à l'aise.

Quelle est la fonction précise du mot clef "return" dans une méthode ?

Ce mot clef a 2 actions. Tout d'abord son exécution arrête l'exécution de la méthode de telle sorte que l'exécution se continue dans le code appelant. La seconde action est que la valeur fournie après return est le résultat renvoyé à l'appelant.

Est-ce qu'une méthode peut renvoyer plusieurs éléments ?

Non. Comme en mathématiques, une fonction ne renvoie qu'un seul et unique résultat.

Que signifie le mot clef void ?

void est l'absence de type (et de valeur). Ce n'est utilisé que dans le cas du type de retour d'une méthode. La méthode `println`, par exemple a void comme type de retour, car ça n'aurait pas de sens d'écrire `String s = System.out.println("toto");`

À quoi sert une méthode si elle ne renvoie rien (void) ?

Exemple de méthode très utile mais ne renvoyant rien : `System.out.println()`.

Est-ce qu'il y a d'autres fonctions pour private à part de rendre un attribut exclusif pour une seule classe ?

Oui et non. Private peut également désigné toute méthode, toute classe, ou tout constructeur. Étant donné le sous ensemble de Java que nous abordons au PC, seul les cas attributs et méthode peuvent nous intéresser. Par exemple, si une classe était définie comme privée, alors aucune autre classe ne pourrait manipuler de donnée de ce type. Cependant, une classe privée ne pouvant être déclarée qu'en sous-classe, cela ne sera pas vu au PC. De même, un constructeur privé ne pourrait pas être appelé de l'extérieur. C'est une pratique qui peut prendre du sens pour factoriser plusieurs constructeurs dans un constructeur chainé, qui n'a pas vocation à être utilisé par les utilisateurs de la classe.

Que signifie concrètement le fait qu'un attribut soit déclaré en private ?

Cela signifie qu'il ne sera pas accessible directement depuis une autre classe. Cela rentre dans le cadre d'une bonne pratique de l'orienté objet, que l'on appelle l'encapsulation (Cf poly)

Quel est l'intérêt de définir les attributs en private ?

C'est un principe de programmation qui fait parti des bons usages lorsque l'on programme en orienté objet. On l'appel le principe de l'encapsulation. Pour s'en faire une image, si je range mon bureau et que tout y est posé proprement et que mon voisin a besoin d'une agrafeuse qui s'y trouve. Il peut soit la prendre sans me parler, soit me demander s'il peut la prendre. Je préférerai toujours le second cas, ne serait-ce que pour vérifier qu'il y a bien des agrafes quand je la lui passe, mais aussi pour avoir la possibilité de lui dire non si cela fait trop souvent qu'il me l'emprunte alors qu'il est sensé en avoir une. C'est le même principe. Par ailleurs, ce principe permet une meilleure maintenance et évolutivité des programmes, ainsi qu'une meilleure répartition des tâches lorsque l'on développe à plusieurs.

Que signifie que les attributs ne sont pas accessibles depuis l'extérieur d'une classe ?

Cela veut dire que si la variable v est une voiture ayant un attribut consommation, alors je ne peux pas, depuis une autre classe, parler de v.consommation. Si je veux connaître l'information, il faudra que je le demande gentiment si cela m'est autorisé. Concrètement, si la classe Voiture fournit un getteur (Ex : `getConso()`), alors je pourrai faire un appel à `v.getConso()`. Et dans le cas d'une voiture de marque allemande, il est probable que cette méthode ne me renvoie par la vraie consommation, mais une valeur inférieur.

À quoi vont nous servir les attributs ?

Les attributs sont des données que l'on veut manipuler. Mais au lieu de les stocker « en vrac » sous forme de N variables, on les réunis dans des objets, qui réunissent l'ensemble des caractéristiques d'un même objet. C'est donc « uniquement » un déplacement des informations.

Pourquoi une méthode ne peut-elle pas être définie dans une méthode ?

Une méthode est un moyen d'interaction avec un objet. Il n'y aurait donc pas de sens à enfouir une méthode dans une méthode, ce qui en interdirait l'accès depuis ailleurs. Cependant, au PC, nous n'abordons qu'une partie du langage Java. Dans sa version complète, il est également possible de créer des lambda termes, qui sont de variables de type fonction (et non pas méthode), qui peuvent être crées à la volée dans une méthode, transmises en paramètre d'appel et exécutées le moment voulu.

Que fait le mot clé "new" ?

Il instancie un type non primitif (Tableau ou objet). Il va donc réserver un espace nécessaire pour stocker l'élément créé et faire appel à la séquence de construction (le constructeur dans le cas d'un objet).

Peut on utiliser plusieurs méthodes à la suite ?

Oui. Une méthode peut appeler une méthode qui en appellera une autre, et ainsi de suite. De même, peut être qu'un objet `Cours` manipulera des `Eleve`, qui eux même contiendront la notion de `Cerveau`. Quand `Cours` interagira avec `Eleve`, alors c'est `Eleve` qui interagira avec `Cerveau`.

L'instanciation, c'est créer un espace-mémoire pour stocker un objet. Mais quelle est la différence avec la déclaration ?

La déclaration créer un espace mémoire permettant de stocker une adresse. Au début, cette adresse est null.

Est-il possible d'utiliser une méthode au sein de la même classe ?

Oui. Et elle sera exécutée dans le contexte de l'objet courant. La seule exception est la méthode main, qui est statique. Elle ne pourra pas faire d'appel à des méthodes non statiques. Elle devra donc nécessairement appeler des méthodes sur des objets, avec la notation pointée.

Pourquoi sur les programmes d'objet il est écrit : public static void main (String[] a) et pas public static void main (String [] arg) ?

Ce n'est pas lié à l'objet. C'est juste que le nom du paramètre entrant (a ou arg dans les 2 exemples cités) est libre. C'est la seule partie de la signature du main qui soit laissée libre.

Dans la signature de la méthode main de la classe principale, que signifient les paramètres String[] et args ?

`args` est le nom d'un paramètre et `String[]` est son type (tableau de chaines de caractères). Ce tableau reçu en paramètre correspond à des paramètres que l'on peut donner au programme au moment de son lancement. On appelle cela des paramètres ligne de commande. L'usage de la ligne de commande au PC ayant diminué, nous ne manipulerons probablement pas cette entrée.

Peut-on générer une copie d'un objet ?

Oui et non. C'est au concepteur du type de prévoir une manière de cloner une instance. Si cela n'a pas été prévu, alors il faut le faire « à la main ».

Un objet peut-t-il être attribut d'un autre objet ?

Oui. Une classe est un type comme un autre.

Y aurait-il un moyen mnémotechnique pour ne pas confondre classe objet instance type méthode ?

Non, pas à ma connaissance. C'est du vocabulaire à apprendre progressivement. Un exposé aujourd'hui a essayé de faire une proposition de grille de synonymes.

D'où est importée la classe scanner que l'on utilise dans un programme ?

Elle vient du dossier java/util, qui est dans la bibliothèque java. C'est d'ailleurs pour cela qu'il est nécessaire de l'importer en début de programme.

Qu'elle est l'utilité du mot clef static ?

Java est un langage fortement orienté objet. Définir une méthode ou un attribut en static permet de s'extraire de l'univers de l'orienté objet.

Je n'ai pas compris le passage de paramètres par valeurs. Pourquoi les paramètres donnés dans une méthode restent inchangées alors la méthode les modifie ? (section 2.2.5)

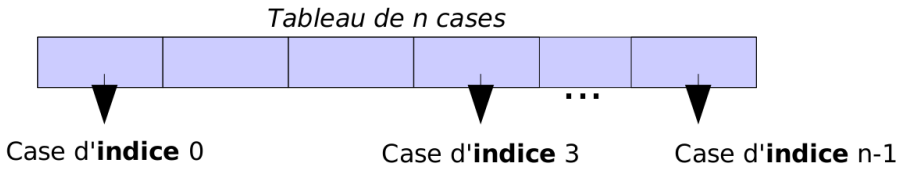
Lors d'un appel de méthode avec un passage de paramètre par valeur, le contenu de la variable va être copié et c'est une copie qui sera transmise à la méthode appelée. Ainsi, la méthode peut modifier la valeur reçue, mais ce ne sera que la copie. À l'inverse, si le paramètre est un type non primitif alors c'est une adresse mémoire qui est copiée dans le paramètre.

4 LES TABLEAUX

Les tableaux permettent de regrouper un ensemble de valeurs de même type. Les tableaux à 1 dimension sont comparables à la notion de vecteur en mathématiques, tandis que les tableaux à 2 dimensions sont comparables à la notion de matrice.

4.1 TABLEAUX À 1 DIMENSION

Un tableau à 1 dimension est une collection linéaire d'éléments de même type. Chaque case d'un tableau est identifiée par un indice et contient une valeur. Les indices d'un tableau commencent à 0. Il s'ensuit, qu'un tableau de n éléments aura des indices compris entre 0 et $n-1$.



Le type contenu dans les cases du tableau est choisi lors de la déclaration du tableau. En revanche, la taille du tableau, ne fait pas parti de son type et sera définie lors de l'instanciation du tableau. La déclaration d'un tableau se fait avec la syntaxe [].

```
int[] tabInt ;   // Déclaration d'un tableau d'entiers
char[] tabChar ; // Déclaration d'un tableau de caractères
```

L'instanciation quant à elle précise la taille à réserver. Elle se fait avec le mot clé `new`.

```
int[] tabInt ;         // Déclaration d'un tableau d'entiers
tabInt = new int[10]; // Instanciation d'un tableau de 10 entiers
```

A noter qu'il est possible, dans le cas de l'initialisation uniquement, de décrire tout le tableau sous la forme d'une liste de valeurs. Cela initialise automatiquement le tableau avec le nombre adéquate de cellules et les valeurs sont stockées dans les différentes cases.

```
int[] tabCinq = {12, 33 ,44,0,50}; //Initialisation expresse
```

Une fois le tableau initialisé, on accède aux éléments du tableau à l'aide de la syntaxe suivante :

```
int i = 0;
int valeur1 = tabCinq[i]; //Renvoie 12, l'élément d'indice 0
int valeur2 = tabCinq[4]; //Renvoie 50, l'élément d'indice 4
```

Une fois initialisé, il est possible à tout moment de connaître la taille d'un tableau (son nombre de cases) à l'aide de la syntaxe suivante :

```
int taille = tabCinq.length; //Renvoie 5, le nombre de cases du tableau tabCinq
```

EXEMPLE D'UN TABLEAU DE VALEURS PRIMITIVES Voici un programme générant les N premières valeurs d'une série ($U_0 = 1/3$ et $U_n = U_{n-1} * 7 - 2$, dont tous les termes devraient valoir $1/3$), puis les affichant :

```
public class SerieFolle {
    public static void main(String[] a) {
        final int N = 25;
        double[] vals = new double[N];

        vals[0] = 1.0/3; // Stockage de U0
```

```
        for(int i=1 ; i<N ; i++) {
            vals[i]=vals[i-1] * 7 - 2; // calcul et stockage de Ui
        }

        System.out.println("Série calculée : ");
        // Affichage de la série
        for(int i=0 ; i<N ; i++) {
            System.out.println("  U"+i+" = "+vals[i]);
        }
    }
}
```

EXEMPLE D'UN TABLEAU D'OBJETS Voici le morceau d'une classe permettant d'ajouter des abonnés à un tableau. Les abonnés étant des objet, la référence `null` permet d'identifier une case vide du tableau.

```
public class EnsembleDesAbonnes {
    //Attributs
    private Abonne[] lesAbonnes;
    private int nbAbonnes;

    //Constructeur
    public EnsembleDesAbonnes(int nbMaxAbonnes) {
        lesAbonnes = new Abonne[nbMaxAbonnes];
        nbAbonnes = 0;
    }

    // Ajoute un abonné. Renvoie vrai ssi l'ajout a fonctionné
    public boolean addAbbonne(Abonne a) {
        res=false;
        // S'il reste de la place
        if(nbAbonnes<lesAbonnes.length) {
            // On cherche la première place libre (elle existe) et on stocke a
            int i=0;
            while(lesAbonnes[i]!=null) {
                i++;
            }
            lesAbonnes[i]=a;
            res=true;
        }
        return res;
    }
}
```

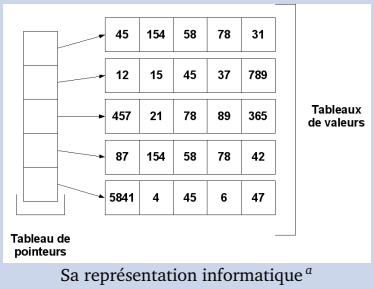
4.2 TABLEAUX À N DIMENSIONS

Un tableau à 2 dimensions est un cas particulier de tableaux à 1 dimension. En effet, c'est simplement un tableau dont chaque case contient un tableau à 1 dimension d'éléments. Par récurrence, il est donc possible de définir un tableau à n dimensions. Du point de vue de la notation, chaque dimension correspond à un couple de crochets supplémentaires au niveau du type et des accès.

```
char[][] T ;          // Déclaration d'un tableau à 2 dimensions de caractères
T = new char[3][5];   // Instanciation d'un tableau contenant 3 tableaux de
                      // 5 caractères chacun. T a donc 3 lignes et 5 colonnes.
```

45	154	58	78	31
12	15	45	37	789
457	21	78	89	365
87	154	58	78	42
5841	4	45	6	47

Une matrice



Sa représentation informatique[a]

```
int[][] T =
    {{  45, 154, 58, 78,  31},
     {  12,  15, 45, 37, 789},
     { 457,  21, 78, 89, 365},
     {  87, 154, 58, 78,  42},
     {5841,   4, 45,  6,  47}};

int i = T[0][1];//i vaut 154
int[] T1 = T[1];//T1 fait
      // référence au tableau
      // {12,15,45,37,789}
```

Le code Java

a. Image provenant de http ://fr.wikipedia.org/wiki/Tableau_(structure_de_données)

Les multiples crochets sont lus de gauche à droite, ce qui correspond à regarder les tableaux depuis le plus extérieur vers plus plus intérieur. Ainsi, T.length renvoie 3, la taille du tableau extérieur, T[0].length renvoie 5 la taille du tableau contenu dans la première case de T. C'est la même convension qu'en mathématiques pour les matrices, d'abord les lignes, puis les colonnes. Pour finir, comme chaque dimension est un tableau, il s'ensuit que chaque dimension est indicée de 0 à $n-1$.

4.3 Exemple d'algorithmes simples à connaître

La majeur partie des calculs que l'on peut avoir à faire sur des données contenues dans un tableau peuvent se résumer à ces simples algorithmes :

— Application d'une fonction sur un tableau (Ex : Calcul de la somme des éléments présents)
— Recherche dans un tableau (Ex : recherche de l'élément le plus petit)
— Trier les données contenues dans un tableau

Les algorithmes correspondants à ces différentes familles de calcul ont été abordés en TD. Nous vous conseillons fortement de les connaître, ce qui vous permettra de plus facilement capitaliser pour résoudre des problèmes plus complexes.

5 Diagramme de classes UML

UML (Unified Modeling Languages) est une famille de langages graphiques permettant de décrire différentes facettes d'un système. Dans le cadre du cours d'algorithmique et programmation, nous n'abordons qu'une petite portion d'un seul de ces langages : le diagramme de classe. Par abus de langage, nous parlerons donc indifféremment d'UML ou de diagramme de classes en le considérant comme des synonymes.

Le diagramme de classes permet de mettre en évidence la structure d'un programme Java. La figure 2 montre l'exemple d'un diagramme de classes contenant 4 classes, et 2 types de liaisons. Cet exemple matérialise un programme de billetterie de train et sera décrit par la suite.

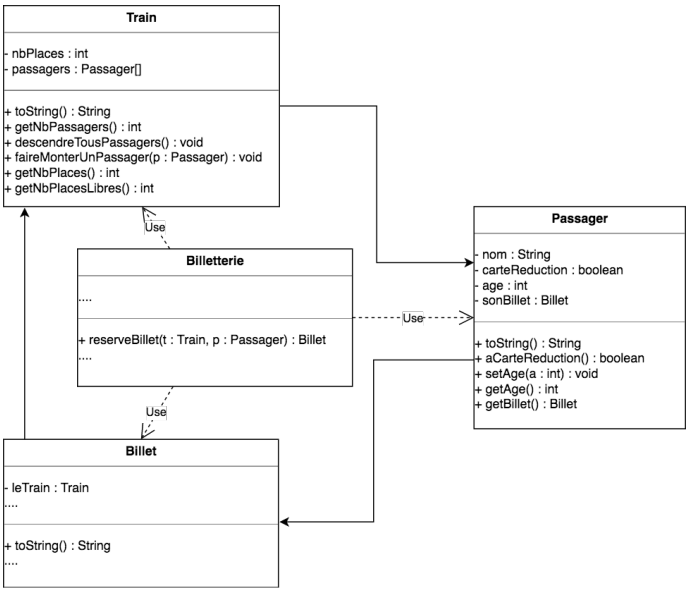


FIGURE 2 – Exemple d'un diagramme de classe

LES CLASSES Chaque classe est représentée par un rectangle constitué de 3 parties alignées verticalement (figure 3). La première partie ne contient que le nom de la classe, la second partie contient la liste des attributs de la classe et la dernière partie contient la liste de ses méthodes.

Ce langage n'étant pas dédié à Java spécifiquement la notation pour les types n'est pas la même qu'en Java. On notera par exemple `x : int` pour désigner un attribut nommé `x` de type entier. De même pour les méthodes, où le type de retour sera mi en fin de signature. Les attributs aussi bien que les méthodes peuvent être décorés d'une visibilité. Il existe 4 visibilités en Java (cf section 3.3.3), et elles sont matérialisées comme suit en début de chaque ligne :

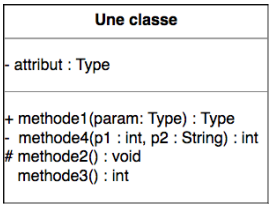


FIGURE 3 – Représentation UML de la structure générale d'une classe

Visibilité	symbole
public	+
package	espace (ou ∼)
protected	#
private	–

Pour reprendre l'exemple de la figure 2, la classe Train est donc constituée de 2 attributs privés (l'entier `nbPlaces` et le tableau d'objets `passagers`), tandis qu'elle contient 6 méthodes.

LES LIAISONS Dans le cadre de ce module, nous n'utilisons que 2 types de liaisons UML :

— **Relation de dépendance (usage) :** Si une classe A manipule des données de type B dans ses méthodes mais qu'elle ne possède pas d'attribut de type B, alors il existe une relation de dépendance de A vers B. Une flèche hachurée ira donc de A vers B avec l'étique "Use" dessus, comme montré dans le premier exemple de la figure 4.

— **Relation structurelle (Associaton) :** Si une classe A possède un ou des attributs de type B, alors il existe une relation structurelle de A vers B. Une flèche pleine ira donc de A vers B, comme montré dans le deuxième exemple de la figure 4.

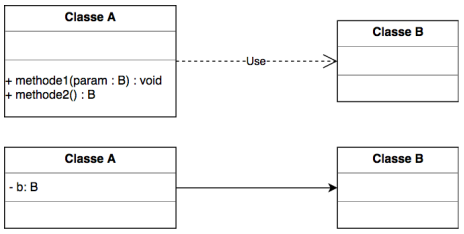


FIGURE 4 – Deux types de liaisons utilisés en première année

POUR ALLER PLUS LOIN Il est à noter que les diagrammes de classe UML sont utilisés dans de nombreux domaines et langages et qu'il existe une large variation autour des syntaxes utilisées. Vous pouvez néanmoins regarder les supports suivants pour avoir des explications plus détaillées et plus larges autour d'UML :

`laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes#L3-3-8-a`
`www.linux-france.org/prj/edu/archinet/DA/fiche-uml-relations/fiche-uml-relations.html`
`uml.free.fr/cours/p15.html`

6 MÉTHODE DE TRAVAIL

6.1 MÉTHODE DE DÉVELOPPEMENT ET DÉBUGGAGE

Lorsque l'on écrit un programme, il ne faut jamais être trop sûr de soit. Nous faisons tous des erreurs d'inattention. Cela peut avoir de lourdes conséquences si le point de départ de ce que l'on a écrit est faux et que non seulement tout notre programme se base dessus, mais qu'en plus on en a fait des copiés collés. Il est donc important toujours prendre me temps de **vérifier** son programme régulièrement.

Il existe différentes méthodes de vérification, apportant plus ou moins de confiance. Par ordre croissant de confiance :

— Compiler : compiler son programme permet de s'assurer qu'il est syntaxiquement correct. Cela ne donne pas un grand niveau de confiance en ce qu'il fait, mais c'est déjà une information.

— Exécuter : exécuter le programme en passant par le `main` et dans le cadre de son exécution normale.
— Tester : écrire du code complémentaire dont le seul objectif est d'évaluer le votre programme fonctionne. Idéalement, c'est une classe à part, contenant un autre `main`, et faisant des appels directs aux méthodes testées. Pour tester une méthode, il faut normalement l'appeler plusieurs fois, dans plusieurs contextes, pour vérifier que tous les comportements "particuliers" sont valides.

Si plusieurs erreurs apparaissent, il faut toujours commencer par corriger la première erreur. Parfois les erreurs suivantes n'existent plus ou sont modifiées par la correction de la première erreur.

Si vous ne comprenez pas pourquoi un code s'exécute d'une manière au lieu d'une autre (et éventuellement, pourquoi vous avez une erreur à l'exécution), alors la méthode de base consiste à instrumenter votre code. C'est à dire à ajouter des affichages dans la zone qui pose question pour savoir par où passe l'exécution ou bien quelles sont réellement les valeurs des différentes variables.

Vous ne devez **jamais** laisser un programme dans un état où il ne compile pas, car vous ne pourriez pas le vérifier. L'excuse classique est "*j'ai commencé à préparer un truc pour plus tard, alors c'est normale que ça ne compile pas*". Mais cela est totalement irrecevable, car vous devez toujours être en capacité de tester votre code. Exemples :

— Si vous avez écrit du texte pour vous en note pour plus-tard, alors commentez le.
— Si vous avez écrit un code approximatif quelque part, mais que c'est "juste pour l'idée", alors commentez le. Si sa présence est nécessaire, alors remplacez le par une valeur constante.
— Si vous faites appel à des méthodes que vous n'avez pas encore écrites, alors commencez par écrire des méthodes bouchons : des méthodes ne faisant rien mais respectant la signature. Si par exemple deux méthodes nécessaires non écrites étaient :

```
public int calculF(double a, double b) {}
public String analyseResultat() {}
```

Alors faites des méthodes bouchon, qui permettrons à votre code de compiler :

```
public int calculF(double a, double b) {return 0; } // Code faux, à faire plus tard
public String analyseResultat() {return ""; } // Code faux, à faire plus tard
```

Une autre excuse entendue est "*oui, mais j'avais trop d'erreurs, alors j'ai abandonné et j'ai continué*". La bonne approche serait plutôt de commenter les parties du code générant le plus d'erreurs pour se concentrer sur un code le plus petit possible et faire des tests (Cf "tester", plus haut). Dans le pire des cas, en cas de désespoir, on commente tout ce qui ne compile pas, on le remplace éventuellement par des méthodes bouchon ou des valeurs constantes et on continue d'avancer en se laissant la possibilité de tester son code.

6.2 Problème de la page blanche : méthodologie de conception d'un programme

Si vous avez un problème à résoudre via un programme, alors idéalement, votre démarche serait la suivante :

1. Description, en français (ou toute autre langue naturelle), d'une succession d'actions (pas forcément élémentaires) menant à la solution
2. Simplification des actions pour les rendre plus simples
3. Traduire les routines les plus internes (le code le plus imbriqué) en Java (et les déclaration des données utilisées)
4. Tester ce qui a été fait.
5. Une fois que les tests sont concluants, revenir en 3, jusqu'à traduction complète

Si les étapes 1 et 2 ont été faites en commentaires dans une classe Java, alors il suffit d'écrire le programme java entre les différents commentaires pour qu'ensuite, le programme produit soit non seulement correct, mais également commenté.

On n'écrit jamais un programme en entier sans le tester au fur et à mesure. Sinon, il sera trop complexe de le corriger. Et il sera plus simple de le recommencer de zéro.

MISE EN SITUATION (EXEMPLE) Problème : calculer et afficher l'ensemble des diviseurs communs à deux nombres donnés.

1. Description, en français, d'une succession d'actions (pas forcément élémentaires) menant à la solution

```
public class DiviseursCommuns {
    public static void main (String[] a){
        // Déclaration et initialisation des deux nombres N et M à étudier
        // Pour chaque i dans [1..min(N,M)]
        //     Si i divise N et i divise M alors
        //         afficher i
    }
}
```

2. Simplification des actions pour les rendre plus simples

```
public class DiviseursCommuns {
    public static void main (String[] a){
        // Déclaration et initialisation des deux nombres N et M à étudier
        // Calcul de min, le minimum entre N et M
        // Pour chaque i dans [1..min]
        //     Si i divise N et i divise M alors
        //         afficher i
    }
}
```

3. Traduire les routines les plus internes en Java (et les déclaration des données utilisées)

```
public class DiviseursCommuns {
    public static void main (String[] a){
        // Déclaration et initialisation des deux nombres N et M à étudier
        final int N=9;
        final int M=12;

        // Calcul de min, le minimum entre N et M
        // Pour chaque i dans [1..min]
        int i=3;
        //     Si i divise N et i divise M alors
        if(N%i==0 && M%i==0) {
        //         afficher i
            System.out.println(i+" divise "+N+" et "+M+".");
        }
    }
}
```

4. Tester ce qui a été fait Compilez et exécutez. Changez la valeur de i et vérifiez que les résultat est toujours cohérent.
5. Une fois que les tests sont concluants, revenir en 3, jusqu'à traduction complète

```
public class DiviseursCommuns {
    public static void main (String[] a){
        // Déclaration et initialisation des deux nombres N et M à étudier
        final int N=9;
        final int M=12;

        // Calcul de min, le minimum entre N et M
        int min = N ;
        if(min>M) {
            min=M ;
        }

        // Pour chaque i dans [1..min]
```

```
        for(int i=1 ; i<=min ; i++) {
        //      Si i divise N et i divise M alors
            if(N%i==0 && M%i==0) {
        //        afficher i
                System.out.println(i+" divise "+N+" et "+M+".");
            }
        }
    }
}
```

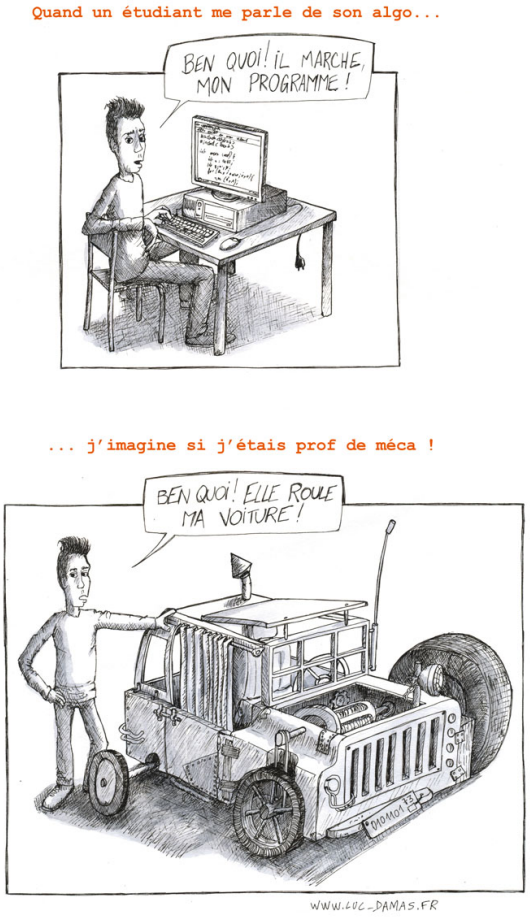
6.2.1 Ressources complémentaires

— Vidéo de 20' parlant de méthode de travail sur un exemple concret :
`https://videos.insa-lyon.net/videos/?video=MEDIA171005135015998`

6.2.2 Exercice

Problème : Calculez et affichez la liste des diviseurs premiers d'un nombre donné.
Rappel : un nombre est premier s'il n'est divisible que par 1 et lui même.
À faire, en respectant les conventions de codage :
— Proposez une décomposition algorithmique du problème.
— Proposez une programme Java résolvant ce problème. À défaut, proposez des traductions Java des parties clefs de votre algorithme.

6.3 Conventions de codage pour Java



> « **Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.** » *Martin Golding*

Ce guide est un extrait traduit en français de « *Code Conventions for the Java Programming Language* »

mis en ligne par Oracle [5] et utilisé dans l'industrie.

RÈGLES DE NOMMAGE Il ne faut **jamais** utiliser de caractères spéciaux ou d'espace dans les noms d'identifiants (classes, variables, constantes, méthodes, ...). Il faut donc se restreindre au lettres (**sans accent**), aux chiffres et à la barre de soulignement.

Terminologie : *un mot est dit **capitalisé** s'il est constitué de plusieurs mots, commençant chacun par une majuscule. Par exemple, « BonjourMonsieurLeProfesseur » est un mot capitalisé.*

Type	Commentaire	Exemples
Classes	Les noms de classes devraient être des noms communs. Ils doivent être **capitalisés**.	`class Voiture;` `class GestionDesImages;`

Variables	Les noms des variables doivent être expressifs. Il faut limiter autant que possible les noms de variables à 1 caractère. Ils doivent être **capitalisés**, mais **doivent commencer par une minuscule**.	`int i;`
		`char cp;`
		`double myWidth;`

Constantes	Les noms de constantes doivent être **écrits en majuscules**. S'ils sont composés de plusieurs mots, alors la barre de soulignement ('_') est utilisée comme séparateur.	`final int MIN_WIDTH = 4;`
		`final int MAX_WIDTH = 999;`
		`final int NB_OF_CPU = 1;`

Methodes	Les noms de méthodes devraient être, ou contenir, des verbes. Ils ont les **mêmes règles de syntaxe que les variables** (capitalisés, commençant par une minuscule).	`run();`
		`runFast();`
		`getBackground();`

PRÉSENTATION DES BLOCS Dans un programme Java, il faut :

- Ouvrir les accolades "{" sur la ligne de la structure contenant le bloc.
- Fermer les accolades "}" sur la première ligne qui suit la dernière instruction. L'accolade doit être seule sur la ligne (sauf dans le cas du `else`, voir ci-dessous).
- Décaler le contenu du bloc de 4 espaces à droite (via la touche tabulation, si Geany est configuré correctement) par rapport au bloc conteneur (on parle d'**indentation**).

```
class ProgrammeExemple {
    ProgrammeExemple(int i, int j) {
        int ivar1 = i;
        int ivar2 = j;
    }

    int emptyMethod() {
        ...
    }
}
```

Exemple de classe

```
class␣ProgrammeExemple␣{
␣␣␣␣ProgrammeExemple(int␣i,␣int␣j)␣{
␣␣␣␣␣␣␣␣int␣ivar1␣=␣i;
␣␣␣␣␣␣␣␣int␣ivar2␣=␣j;
␣␣␣␣}

␣␣␣␣int␣emptyMethod()␣{
␣␣␣␣␣␣␣␣...
␣␣␣␣}
}
```

Même exemple avec matérialisation des espaces

COMMENTAIRES Toute déclaration de méthode doit être précédée d'un petit commentaire entre /** et */. Il décrit ce que fait la méthode (**ATTENTION !** On ne décrit pas **comment** le fait la méthode).

```
/** Methode principale. Elle verifie la validite des parametres
  * avant de commencer le programme.
  */
public static void main(String[] args) {
    instructions;
}
```

Pour les commentaires courts et relatif à une seule ligne, on utilise deux symboles / comme illustré ci-dessous :

```
int level; // commentaire court
```

5. http://www.oracle.com/technetwork/java/codeconv-138413.html

NOMBRE D'INSTRUCTIONS PAR LIGNE Chaque ligne doit contenir au plus une instruction. Par exemple :

```
argv++; argc--; // INTERDIT!
```

Cette règle est également vraie pour les déclarations. En effet, cela encourage la description des variables par des commentaires. Par exemple :

```
int level; // Niveau d'indentation
int size;  // Taille du tableau
```

est préférable à :

```
int level, size;
```

PRÉSENTATION DES STRUCTURES CONDITIONNELLES Voici les différentes formes que peuvent avoir les structures conditionnelles, y compris quand elles sont successives :

```
if (condition) {
    instructions...;
}
```

```
if (condition) {
    instructions...;
} else {
    instructions...;
}
```

```
if (condition) {
    instructions...;
} else if (condition) {
    instructions...;
} else (condition) {
    instructions...;
}
```

PRÉSENTATION DES STRUCTURES DE BOUCLES Les structures de boucles doivent respecter les formes suivantes :

```
for(... ; ... ; ...) {
    instructions...;
}
```

```
while (condition) {
    instructions...;
}
```

```
do {
    instructions...;
} while(condition);
```

CONSTANTES NUMÉRIQUES Il faut **limiter** le nombre de constantes numériques apparaissant directement dans le code. À part -1, 0 et 1 dans le cas des boucles, il est **souvent** préférable d'introduire des constantes nommées (précédées du mot-clé **final**).

EXEMPLE

```
import java.awt.Graphics;
import java.awt.Color;

public class Polygone extends Courbe {

    private APoint[] points;

    /**
     * Le construteur duplicant le tableau en parametre
     */
    public Polygone(APoint[] p){
        points = new APoint[p.length];
        for(int i=0; i<p.length; i++) {
            points[i] = new APoint(p[i].x,p[i].y);
            if (p[i].y>=0) {
                if (p[i].x>=0) {
                    System.out.println("1er quadrant");
                } else {
                    System.out.println("2nd quadrant");
                }
            } else {
                if (p[i].x<=0) {
                    System.out.println("3eme quadrant");
                } else {
                    System.out.println("4eme quadrant");
                }
            }
        }
        System.out.println("Polygone cree.");
    }

    /**
```

```
     * Pour calculer le perimetre d'un polygone
     */
    public double longueur(){
        double resultat=0;
        for(int i=1; i<points.length; i++) {
            resultat += points[i-1].distance(points[i]);
        }
        resultat += points[points.length-1].distance(points[0]);
        return resultat;
    }
}
```

7 Outils

7.1 Java

— Compilateur : JDK
— Éditeur : Geany
— Ligne de commande sous Windows : git bash

L'ensemble des informations nécessaires sont disponibles sur la page moodle de ressources communes en Algorithmique et programmation.

7.2 Diagrammes de classe – UML

Le site web `https://www.draw.io/` permet notamment d'éditer des diagrammes de classe de manière efficace.

8 Référentiel compétences des modules d'algorithmique

8.1 Bilan synthétique de première année EUR/AMER

Référentiel des compétences abordées en Algo1 et Algo2

	Compétences	Infos complémentaires	Niveau visé S1	Niveau visé S2
Démarche : Savoir …				
Savoir être	utiliser la **ligne de commandes**	*java, javac, ls, pwd, cd*	1	2
	compiler / **exécuter** / **tester**	*Prendre l'initiative de faire des tests. Rendre un programme qui compile. Ajout potentiel de code bouchon ou commenté.*	1	2
	respecter les **conventions de codage**	*Identifiants expressifs, indentation, commentaires*	1	2
Analyse, conception et algorithmique : Savoir …				
Algorithmique	écrire un programme implémentant un **algorithme fourni**.		1	2
	écrire un programme **tiré d'un problème mathématique**	*(Ex : Calcul d'une série)*	1	3
	identifier les **variables** nécessaires à l'écriture d'un programme		1	3
	écrire une **expression booléenne** répondant à un problème		1	3
	écrire une **expression non-booléenne** répondant à un problème	*(Ex : arithmétiques)*	1	3
	appeler une **méthode pour résoudre un problème**	*(avec des paramètres adéquats)*		3
	écrire du algorithme **résolvant un problème**		1	3
	décomposer un problème en **sous-problèmes**, **classes** et **méthodes**.			2
	écrire un algorithmes utilisant des **tableaux** (1D ou 2D)	*(Ex : recherche min/max, tri de tableau, …)*	1	3
	écrire un algorithmes utilisant des **chaines de caractères**.			1
	prendre en compte le **coût à l'exécution**			
Bons usages	utiliser des **boucles à bon-escient**	*Notamment, distinction For/while*	2	3
	utiliser des **conditionnelles à bon escient**		2	3
	respecter l'**encapsulation**	*Attributs privés + accesseurs*		3
	écrire des **tests fonctionnels**	*Pertinents & couvrants*		3
Langage et codage (SYNTAXE uniquement) : Savoir …				
Variables et types	**Variables** : **type**, **déclaration** et **portée**	*Types au programme : **int**, **double**, **boolean** et **char***	1	2
	écrire une **expression** (**non-Booléennes** ou **booléenne**)	*(Ex : arithmétiques)*	1	2
	utiliser des **opérateurs raccourcis** (+=, -=, …)	*Seul i++ est au programme.*		
	manipuler des types non-primitifs		1	2
	convertir les **types primitifs** (entre eux ou en String)		1	2
	convertir les **types non-primitifs**			
	la syntaxe liée à l'usage d'un **Tableaux** (1D et 2D)			2
	la syntaxe liée à l'usage d'une **chaine de caractères**			2
Structures de contrôle	écrire une conditionnelle **if**		1	3
	écrire un e conditionnelle **switch**			
	écrire une boucle **for**		1	3
	écrire une boucle **while**		1	3
	écrire une boucle **do**/**while**			
Orienté objet	**appeler une méthode** fournie	*(Notamment via JavaDoc)*	1	2
	déclarer une méthode			2
	créer une **classe**			2
	Méthode ou attribut **static**	*Sauf le main*		
Général	respecter les **contraintes syntaxiques** générales	*Nom classe / nom fichier, définition du main, accolades et blocs.*	1	3
	lire une entrée au clavier (via **Scanner**)		M	1
	Faire un tirage aléatoire **Math.random()** dans un intervalle donné		M	1
	utiliser un **break** hors d'un **switch**	*Interdit et sanctionné*	Interdit	
	utiliser des **return multiples** dans une méthodes ou un «**return;**» vide	*Interdits et sanctionnés*	Interdits	
UML	**écrire des classes** conformes à un **diagramme de classes**		1	2
	écrire un diagramme de classes à partir d'un cahier des charges			2

Légende des niveaux de compétence	
	: Hors programme
M	: Mobilisées
1	: Sensibilisation
2	: Appropriation
3	: Maitrise

8.2 Attendus de fin de module Algo 1 - EUR/AMER

- savoir ce qu'est une variable, une constante
- connaître les types primitifs vus en cours : int, double, char, boolean
- connaître les instructions simples de java : l'affectation, l'affichage.
- savoir utiliser à bon escient les différente structures de contrôle :
  - conditionnelles simples (if () et if() else)
  - répétitives (while, for) et savoir en faire un usage adapté.
- comprendre et savoir écrire des structures de contrôle de manière imbriquée
- savoir dérouler "à la main" un programme et expliquer le contenu de ses variables et l'affichage qu'il produit
- savoir écrire un programme java simple avec une méthode `main`
- savoir utiliser des classes de la bibliothèque Java à partir de leur javadoc
- connaitre et mettre en oeuvre la convention de codage dans une classe java
- connaitre la méthode standard `toString`
- connaitre la gestion des tableaux en mémoire et leur référencement, être capable de l'expliquer et de déduire son impact dans un programme
- comprendre la génération des nombres aléatoires en java
- savoir générer des nombres aléatoires en java
- algorithmes standards à connaître pour capitaliser dessus :
  - Calcul d'une série
  - Application une fonction sur un tableau
  - Recherche d'un min/max dans un tableau
  - Au moins un algorithme de tri de tableau

8.3 Attendus de fin de module Algo 2 - EUR/AMER

- tout le bagage d'Algo 1 est considéré comme acquis et est utilisé en bagage
- savoir instancier et utiliser des classes existantes
- savoir écrire une classe Java modélisant un objet
- savoir écrire et appeler des méthodes
- savoir mettre en oeuvre le principe d'encapsulation
- comprendre le caractère spécial de saut de ligne '
  n'
- savoir convertir des données d'un type primitif à un autre
- comprendre la lecture clavier en java
- connaitre la gestion des objets en mémoire et leur référencement, être capable de l'expliquer et de déduire son impact dans un programme
- savoir résoudre un problème par décomposition en classes et méthodes
- savoir créer et manipuler dans un programme des structures tabulaires d'objets (tableaux d'objets)
- savoir lire et écrire un diagramme de classes UML

8.4 Attendus de fin de module Algo 3 - EUR/AMER

- tout le bagage d'Algo 1 et 2 est considéré comme acquis et est utilisé en bagage
- savoir écrire une classe dérivée d'une classe java (héritage)
- savoir utiliser le constructeur de la classe mère dans le constructeur d'une classe fille
- savoir utiliser le polymorphisme, par exemple pour manipuler une `LinkedList` ou tableau, dont les éléments appartiennent à une même hiérarchie d'objets
- savoir définir et utiliser un objet de type LinkedList d'objets
- savoir définir une fenêtre graphique contenant les widgets vus en TD *(`JLabel`, `JButton`, `JTextField` et `JTextArea`)*
- savoir écrire des classes d'écouteurs d'événements implémentant l'interface ActionListener
- savoir écrire les méthodes qui permettent la réaction aux événements et savoir lier les widgets de l'interface graphique à ces événements
- savoir mettre en oeuvre les liens entre une classe représentant une fenêtre et les classes écouteurs qui lui correspondent

— savoir dessiner dans une fenêtre graphique ou dans un JPanel
— savoir ouvrir une fenêtre graphique à partir d'une autre fenêtre
— savoir gérer des layouts dans les fenêtres graphiques (`BorderLayout`, `FlowLayout`)
— connaître les noms des classes qui permettent de définir les widgets de base et savoir retrouver leur utilisation dans une javadoc
— savoir utiliser des instructions non vues en TD mais dont l'explication est détaillée dans le sujet et/ou dans la javadoc

9 GLOSSAIRE

ACCESSEUR : Les accesseurs sont des méthodes dont le seul rôle est d'accéder aux attributs. Il y a classiquement 2 familles d'accesseurs : les accesseurs de lecture, appelés *getter* et d'écriture, appelés *setter*. Ce concept est lié à l'encapsulation, vue en section 3.3.4.

ADRESSE : Synonyme de *référence*. Une adresse indique où trouver une information dans la mémoire, au même titre qu'une adresse postale permet de retrouver une maison dans une rue. L'adresse `null` caractérise l'absence d'adresse.

AFFECTATION : L'affectation est l'opération qui attribue une valeur à une variable. En Java cet opérateur est '='. Il se lit *"prend la valeur de"*.

APPEL DE MÉTHODES : L'appel d'une méthode exécute le bloc d'instructions de la méthode. L'appel se fait en écrivant le nom de la méthode (en respectant la casse) suivie d'une paire de parenthèses avec éventuellement une liste de paramètres effectifs* séparés par une virgule. Au moment de l'exécution de l'appel, les valeurs des paramètres effectifs sont affectées aux paramètres formels*, selon l'ordre dans lequel ils apparaissent. Si le type renvoyé par la méthode est différent de `void`, l'appel de la méthode doit être affecté à une variable de même type.

ATTRIBUT : Les attributs sont des variables associées à un objet.

BIBLIOTHÈQUE JAVA : L'ensemble des classes fournies lorsque l'on installe l'outil java constituent la *Bibliothèque*. Ces classes sont organisées en packages (en dossiers). Pour utiliser une classe de la bibliothèque Java, il est nécessaire de préciser le package la contenant ou de l'importer. Par exemple, la classe `Scanner` de lecture du clavier est dans le package `java.util` et nécessite pour être utilisée d'importer la classe ou de parler de `java.util.Scanner`.

Le package `java.lang` (dossier `java/lang`) est appelée la *bibliothèque standard*. Les classes qui s'y trouvent peuvent être utilisées comme si elles étaient dans le dossier du programme (sans import). C'est par exemple le cas de la classe `String`.

CONSOLE : La console est une interface textuelle qui permet de demander à l'ordinateur d'exécuter des programmes. Elle est souvent considérée à tord comme étant obsolète. Pour autant il s'agit d'une des interfaces les plus puissantes à utiliser puisque l'on peut directement programmer dans la console. De plus, il s'agit bien souvent de l'unique façon d'accéder à des machines à distance.

COMPILATION : La compilation permet de transformer un code source écrit dans un langage de programmation en langage machine (ou langage binaire) exécutable par l'ordinateur.

DÉCLARATION : Avant de pouvoir utiliser une variable, il faut la déclarer. La déclaration d'une variable associe un type à un nom et réserve un emplacement mémoire dans lequel est stockée la valeur de la variable, si son type est primitif, ou l'adresse du début de la plage mémoire où est stocké la variable si son type est non primitif.

DÉFINITION : Avant de pouvoir utiliser une une méthode, il faut la définir. Définir une méthode consiste à définir le nom de la méthode, le type du résultat retourné, la liste de ses paramètres formels et son bloc d'instructions.

DÉCRÉMENTER : L'opération de décrémentation s'applique à une variable de type entier. Elle consiste à retirer une valeur entière à la variable, classiquement la valeur 1 (`x = x - 1`).

ENCAPSULATION : Le principe d'encapsulation est présenté en section 3.3.4. Il consiste à dire que les attributs d'un objets lui sont privés (Sont encapsulés dans l'objet et n'en sont pas accessibles de l'extérieur). On pourra avoir recours à des accesseurs pour y accéder.

EXÉCUTION : L'exécution est le processus par lequel une machine met œuvre les instructions d'un programme informatique.

GETTER : *accesseur* de lecture. Ce concept est lié à l'encapsulation, vue en section 3.3.4.

IDENTIFIANT : c'est un nom. Cela peut être un nom de classe, de méthode, de variable, de paramètre ou d'attribut. C'est donc par opposition aux mots clefs du langage (`public`, `if` , `for`, `void`, ...), aux constantes littérales (23, « bonjour », 'C', true, ...), aux opérateurs (+, -, %, ...) et aux délimiteurs ([, ', ", ;, ...).

IMMUTABLE : se dit d'une classe permettant de générer des objets qui ne sont pas modifiables (tous les attributs sont privés, pas de setter fourni et aucune méthode n'a pour effet de modifier les attributs). C'est notamment le cas de la classe `String`.

IMPORT : mot clef permettant de déclarer l'usage d'une bibliothèque dans notre programme. Il est présenté en section 3.3.2.

INCRÉMENTER : L'incrémentation est l'opération qui consiste à ajouter une valeur entière fixée à une variable de type entier. La valeur ajoutée est le plus souvent la valeur 1 (`x = x + 1`).

INITIALISATION : Lorsqu'on déclare une variable, un emplacement mémoire est associé à la variable. Or, celui-ci contient une valeur quelconque. Il est nécessaire d'initialiser la valeur de la variable, c'est-à-dire de réaliser une première affectation d'une valeur à la variable, afin que cette dernière contienne une valeur appropriée.

INSTANCE D'UNE CLASSE : On appelle instance d'une classe, un objet avec un comportement (méthodes) et un état (attributs), tous deux définis par la classe. Il s'agit donc d'un objet issu du moule définit par la classe. Développé dans la section 3.3.1.

INSTANCIER : Action de créer une instance*. Réserver l'espace mémoire nécessaire pour stocker toutes les valeurs de l'objet.

MÉTHODE : est l'équivalent d'une instruction en mathématiques. On peut aussi considérer que c'est une macro-instruction. C'est à dire que c'est un morceau de code associé à un nom et que l'on peut appeler à tout moment dans son programme (section 3.2.4).

NULL : caractérise l'absence de référence. C'est la valeur par défaut d'une variable d'un type non primitif qui n'aurait pas été initialisée. Ce mot clef est présenté en section 3.3.1.

OPÉRATEUR : Un opérateur est une fonction spéciale dont l'identificateur s'écrit avec des caractères non autorisés pour les identificateurs ordinaires (les variables). Il s'agit souvent des équivalents aux opérateurs mathématiques pour la programmation informatique.

PARAMÈTRES EFFECTIFS : Il s'agit de valeurs fournies à une méthode lors de son appel. Ces valeurs peuvent être des constantes ou des variables.

PARAMÈTRES FORMELS : On parle aussi de paramètre muet. Il s'agit de variables utilisées dans le bloc d'instructions d'une méthode. Ces paramètres permettent de décrire comment les données en entrée de la sont transformées par celle-ci.

RÉFÉRENCE : Une référence est une valeur qui permet l'accès en lecture et/ou écriture à une donnée en mémoire. Une référence n'est pas la donnée elle-même mais seulement une information de localisation, l'adresse de la valeur dans la mémoire. La constante `null` caractérise l'absence de référence.

STATIC : une méthode ou un attribut peuvent être statiques. L'un et l'autre n'ont plus alors besoin que l'on ait instancié le type pour pouvoir les utiliser. Ce concept est présenté en section 3.3.9.

SETTER : *accesseur* d'écriture. Ce concept est lié à l'encapsulation, vue en section 3.3.4.

SIGNATURE : La signature d'une méthode est sa ligne de déclaration, ce qui regroupe son nom, son type de retour ainsi que ses paramètres attendus (Figure 1, page 18). Exemple de signature d'une méthode :

```
public void maMethode(int param1, String param2)
```

SURCHARGE : Il est possible de définir, dans une même classe, plusieurs méthodes ayant le même nom, à condition qu'elles n'aient pas les mêmes paramètres. Lors de l'appel c'est la liste des types de paramètres fournis qui permettra à la machine virtuelle Java d'exécuter la bonne version de la méthode. On dira que les méthodes ayant plusieurs définitions sont *surchargées*. Ce concept est présenté en section 3.3.5.

THIS : Il a deux usage possibles : référence vers l'objet courant et appel vers un constructeur de l'objet. Ce concept est présenté en section 3.3.6.

TYPES PRIMITIFS : Les types primitifs (entier, flottant, booléen et caractère) permettent de manipuler directement les données les plus courantes et ont la particularité d'être accessibles directement en mémoire.

TYPES NON PRIMITIFS : Les types non primitifs permettent d'associer plusieurs valeurs à une variable. L'emplacement mémoire associé à la variable permet de stocker l'adresse de l'emplacement mémoire où se trouvent ses valeurs. La variable est ainsi liée à ses valeurs de manière indirecte.

VISIBILITÉ : C'est une propriété qui peut être donnée à une classe, un attribut ou une méthode. Il défini qui à le droit d'en faire mention. Il y a 4 visibilités : publique, protégée, paquetage et privée. Ce concept est présenté en section 3.3.3.

10 PERSPECTIVES D'ÉVOLUTION DE CE DOCUMENT

- Ajouter des exercices et des éléments de correction (ou bien dans un document séparé ?)
- Faire des liens avec des ressources externes dans les différents chapitres
- Mettre en avant les choses qui sont hors programme ou au contraintes liées au programme (comment faire ? Pour alléger, mettre en tête ou fin ? Attention au manque de lisibilité)
- Faire un glossaire des messages d'erreur les plus courants + guide de lecture d'un message d'erreur
- Ajouter le programme de deuxième année
- Convertir le poly en syllabus ?
- Mettre quelque part un mot sur les méthodes récursives

11 RÉFÉRENCES

[1] Cyrille Herby. *Apprenez à Programmer en Java*. Simple IT, 2012. Cote : D02 005.133 JAV ; Disponible en MOOC : https ://openclassrooms.com/courses/apprenez-a-programmer-en-java.

[2] John R. Hubbard. *Programmer en Java*. Ediscience, 2002. Cote : D02 005.133 JAV.

[3] Frank Nielsen. *A Concise and Practical Introduction to Programming Algorithms in Java*. Springer London, 2009. Cote : D02 005.133 JAV (077).