Programmation Java

Table des matières

Ce support de cours couvre l'intégralité du programme de première année et vous permettra de comprendre les concepts liés à la programmation en Java. La section 3.4.2 introduit quelques concepts de la programmation objet, qui sont vus en deuxième année du premier cycle.

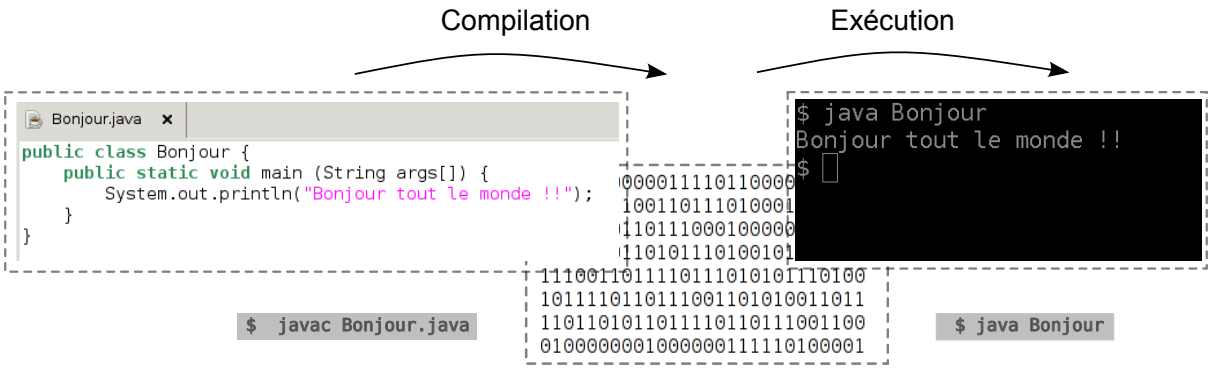
1 Notions élémentaires

1.1 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

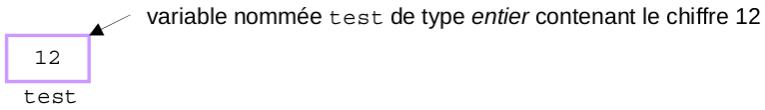
fenêtres comme le fond les programmes tels que Word et Excel) ou plus simplement affichées dans une console*[1] sous forme de texte.

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 microprocesseur de l'ordinateur. Or il est très difficile pour un humain de programmer directement en binaire, 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 :



1.2 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 :

```
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 :

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

```
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.

1.2.1 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
```

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	$[-3.4\,.\,10^{38},\cdots,-1.4\,.\,10^{-45},0,1.4\,.\,10^{-45},\cdots,3.4\,.\,10^{38}]$
`double`	8 octets	$[-1.8\,.\,10^{308},\cdots,-4.9\,.\,10^{-324},0,4.9\,.\,10^{-324},\cdots,1.8\,.\,10^{308}]$

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
```

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

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 caractere = 'a'; // La variable caractere contient la valeur 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.

1.2.2 Les conversions

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

typeVariableA variableA = (typeVariableA) valeurB

Si `variableA` et `valeurB` ne sont pas du même type, cette instruction affecte à `variableA` la conversion de la valeur de `valeurB` dans le type `typeVariableA` :

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 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
```

1.2.3 Les 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 :

==	comparaison d'égalité
!=	différence
<	inférieur strict
<=	inférieur ou égal (s'écrit de la même manière dont on le prononce)
>	supérieur strict
>=	supérieur ou égal (s'écrit de la même manière dont on le prononce)

Attention, l'opérateur = correspond à l'affectation alors que l'opérateur == correspond à la comparaison d'égalité, c'est-à-dire au signe = utilisé en mathématiques !!

Les expressions composées doivent être complètement parenthésées :

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

1.3 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
        //exécutées 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.

1.4 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.

1.4.1 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.

1.4.2 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.

1.4.3 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 (while en anglais) la condition est réalisée.

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 Les méthodes

Une méthode est un bloc d'instructions pouvant être exécutées par un simple appel* de la méthode dans le bloc du programme principal (méthode `main`) ou dans une autre méthode. Les méthodes permettent d'exécuter dans plusieurs parties du programme un même bloc d'instructions. On est amené à créer une méthode dans deux cas de figure :

- Pour regrouper un ensemble d'instructions qui participent à la réalisation d'une même tâche. 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[3]). A ce bloc d'instructions est associé un nom, choisi en rapport avec le traitement réalisé par la méthode.
- Pour regrouper un ensemble d'instructions qui sont répétées à différents endroits du programme (contrairement aux formes itératives qui répètent le bloc d'instructions de manière consécutive).

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.

2.1 Les méthodes prédéfinies

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.`. Lorsqu'on appelle 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);
```

Si cette méthode renvoie un résultat, il faut alors affecter ce résultat à une variable de type compatible pour pouvoir ensuite utiliser ce résultat :

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

2.2 Les méthodes propres

Déclaration d'une méthode La définition d'une méthode s'appelle déclaration*. La déclaration d'une méthode se fait selon la syntaxe suivante :

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

Le fait que l'on doive précéder la déclaration d'une méthode par le mot clé `static` est expliqué page 15. `TypeRetour` est le type de `valeurRetournee`, la valeur renvoyée par la méthode. Si la méthode ne renvoie aucune valeur, le mot-clé `void` est utilisé à la place du type de la valeur retournée.

Pour une raison inconnue des enseignants, il y a souvent confusion entre le fait de retourner une valeur et le fait de l'afficher.

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 la variable x utilisée en mathématique lorsque l'on définit une fonction $f(x)$. Les paramètres sont facultatifs, mais s'il n'y a pas de paramètres, les parenthèses doivent rester présentes.

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.

[3] Commenter la ligne d'appel de la fonction commente tout le bloc d'instructions de la méthode à cet endroit du programme.

```
static int addition(int x, int y) {//x et y sont les paramètres formels
    return x + y;
}
public static void main (String[] args) {
    int a = 7;
    int somme = addition(a,3);//a et 3 sont les paramètres effectifs de l'appel
                              //x prend la valeur de a et y prend la valeur 3
}
```

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.

Rien ne vaut un petit exemple ;-)

```
static int addition(int entierA, int entierB) {
    entierA = entierA + 1;   //entierA est modifié
    return entierA + entierB;
}
public static void main (String[] args) {
    int a = 7;
    int b = 3;
    int somme = addition(a,b); // la valeur de a est affectée à entierA
    // a vaut toujours 7 après l'appel de la méthode addition
}
```

2.3 Les méthodes récursives

Nous avons précédemment précisé qu'il était possible qu'une méthode appelle dans son bloc une autre méthode. Il est également possible qu'elle s'appelle elle-même. A priori, l'intérêt peut paraître limité, mais la récursivité permet d'effectuer certains calculs mathématiques, en particulier avec les suites définies par récurrence. Par exemple, on peut calculer la factorielle d'un nombre entier en utilisant la récursivité. Pour cela, on peut procéder de la manière suivante, par exemple si l'on souhaite calculer la factorielle de 4 :

$$\begin{aligned} 4! &= 4 \times 3! \\ 3! &= 3 \times 2! \\ 2! &= 2 \times 1! \\ 1! &= 0! \\ 0! &= 1 \end{aligned}$$

Dans ce calcul, on constate une récurrence : pour calculer 4!, il suffit de calculer 3!, puis 2!, et ainsi de suite, jusqu'à un cas directement résolu (sans expression récursive, le cas de base). Nous pouvons alors créer une méthode `calculeFactorielle` ayant comme paramètre un entier n, et renvoyant l'entier $n!$. On voit qu'il suffit que la fonction s'appelle elle-même suivant ce même principe pour calculer facilement la factorielle de n'importe quel nombre entier. Cependant, arrivé au calcul de 0!, il est nécessaire que la fonction renvoie directement la valeur 1, sans faire d'appel récursif. En d'autres mots, il ne faut pas que l'appel de `calculeFactorielle(0)` provoque l'appel de `calculeFactorielle(-1)`. Il renvoie directement 1, afin de ne pas provoquer une infinité d'appels :

```
public class Factorielle {
    static int calculeFactorielle(int n) {
        if(n > 0) { // cas général
            return n * calculeFactorielle (n-1);
        }
```

```
        else { //cas de base ou d'arrêt
            return 1;
        }
    }
    public static void main (String[] args) {
        int valeur = 4;
        System.out.println(calculeFactorielle(valeur)); // 5 appels en tout
    }
}
```

2.4 Un exemple

L'exemple suivant présente un programme avec deux méthodes : `CelsiusVersFahrenheit` convertit `valeurAConvertir`, une valeur réelle représentant une température en degré Celsius, en une température en degré Fahrenheit. La valeur en Fahrenheit est retournée par la méthode. `CelsiusVersKelvin` convertit `valeurAConvertir`, une valeur réelle représentant une température en degré Celsius, en une température en degré Kelvin. La température en Kelvin est affichée et aucune valeur n'est retournée par la méthode.

Observez la différence entre les appels des deux méthodes dans le `main` : la valeur retournée par `CelsiusVersFahrenheit` est affectée à la variable `temperature`, puis elle est affichée. La méthode `CelsiusVersKelvin` est directement appelée, sans affectation de sa valeur de retour, puisque celle-ci n'en n'a pas (mot clé `void` dans la déclaration).

Le programme considère les températures entières, en degré Celsius, de 0 à 39 inclus. 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.

```
public class Conversion {
    static double CelsiusVersFahrenheit(double valeurAConvertir) {
        double Fahrenheit = ((9.0/5.0) * valeurAConvertir) + 32.0;
        return Fahrenheit;
    }
    static void CelsiusVersKelvin(double valeurAConvertir) {
        double Kelvin = 273.15 + valeurAConvertir;
        System.out.println(Kelvin);
    }
    public static void main (String[] args) {
        boolean calculeFahrenheit = true;
        for(int celsius = 0; celsius < 40; celsius = celsius + 1) {
            if(calculeFahrenheit == true) {
                double temperature = CelsiusVersFahrenheit(celsius);
                System.out.println(temperature);
            } else {
                CelsiusVersKelvin(celsius);
            }
            calculeFahrenheit = !calculeFahrenheit;
        }
    }
}
```

3 Les types non primitifs

3.1 Généralités

Nous avons vu (section 1.2) qu'une variable permet de stocker dans un emplacement mémoire une donnée d'un certain type, et que cette donnée peut être transformée en utilisant des opérateurs spécifiques de son type. Or, au lieu de manipuler une seule valeur, il est parfois beaucoup plus commode qu'une variable soit

associée à une collection de valeurs. C'est ce que permettent les types non primitifs, appelés `objet` en Java.

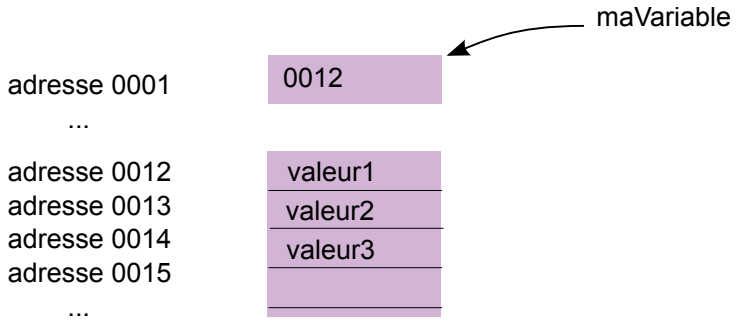
Supposons que l'on souhaite réaliser un programme permettant d'écrire des courriers de manière automatique aux abonnés d'une bibliothèque. Pour chaque abonné, nous connaissons son nom, prénom, le nombre de volumes empruntés ainsi que le nombre de jours écoulés depuis l'emprunt. Nous devons alors manipuler des centaines d'abonnés, chacun décrit par 4 valeurs, certaines de ces valeurs (nom, prénom) étant également une collection de valeurs primitives (une séquence de caractères). Ce type de traitement est impossible en n'utilisant que des types primitifs.

Tout comme des opérateurs spécifiques sont associés à chaque type primitif*, il peut être commode de définir des opérateurs ou méthodes permettant d'interroger ou transformer les valeurs de ces objets.

Nous souhaitons disposer d'une méthode `estEnRetard` qui renvoie la liste des abonnés dont le dernier emprunt a été effectué il y a plus de 21 jours.

Dans cette section nous allons introduire les concepts relatifs aux objets nécessaires à la création d'un tel programme. Avant toute chose, nous allons expliquer les mécanismes communs à tout objet Java, c'est-à-dire, 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. Nous verrons également comment fonctionnent les opérateurs de comparaison sur les objets.

Stockage des objets en mémoire Tout comme les variables de type primitif, une variable de type `objet` est associé 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 les deux variables font référence au même emplacement mémoire, donc au même objet.

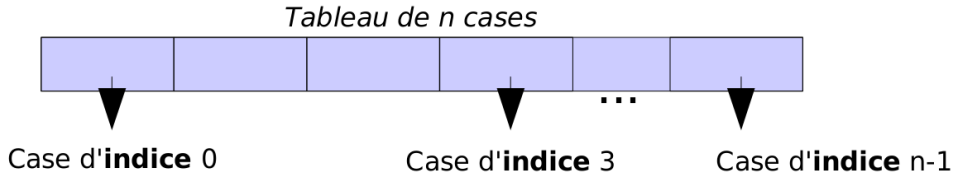
```
if(variableA == variableB){
    System.out.println("Les deux variables font référence au même objet.");
}
```

3.2 Les tableaux

Les tableaux permettent de regrouper un ensemble de valeurs de même type.

3.2.1 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
```

3.2.2 Tableaux à n dimensions

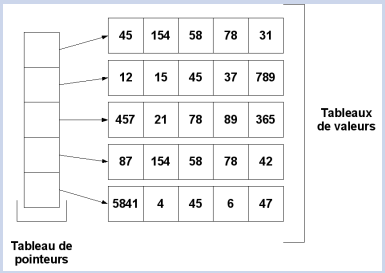
Un tableau à 2 dimensions est un cas particulier de tableaux à 1 dimension. En effet, c'est simplement un tableau dont chaque case contient un tableau à 1 dimension d'éléments. Par récurrence, il est donc possible de définir un tableau à n dimensions. Du point de vue de la notation, chaque dimension 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.
```

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 $n-1$.

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 = new int[5][5];
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)

3.3 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.

3.3.1 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 :

```
static 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;
}
```

3.3.2 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 :

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

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

3.4 Les objets

Les différentes valeurs d'un objet son appelées attributs*. Pour accéder à la valeur d'un attribut, on utilise la syntaxe suivante :

```
//accès à la valeur de l'attribut
TypeAttribut monAttribut = monObjet.nomAttribut;
//affectation d'une valeur à l'attribut
 monObjet.nomAttribut = valeurAttribut;
```

Il peut exister des méthodes spécifiques permettant d'interroger ou de transformer les valeurs des attributs d'un objet. Pour appeler* une méthode sur un objet, on utilise la syntaxe suivante :

```
TypeRetour valeurRetour = monObjet.nomMethode(parametre_1,.., parametre_N);
```

La méthode `nomMethode` permet d'accéder ou de modifier les valeurs des attributs de `monObjet`.

Si une méthode n'accède pas ou ne modifie pas les attributs de l'objet, on dit que cette méthode est `static`. La déclaration de la méthode est précédée du mot clé `static`. L'appel de la méthode se fait comme expliqué dans la section 2.

3.4.1 L'objet prédéfini String

Il existe de nombreux objets prédéfinis en Java. Nous allons dans cette section présenter l'un d'entre eux, incontournable : le type "chaîne de caractères", appelé `String`. Cet objet est le seul qui sera réellement manipulé en première année.

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 et initialisé. Ces opérations se font selon la syntaxe :

```
//Déclaration, instanciation et initialisation
String ecole = new String("INSA-Lyon");
```

Noter 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).

Comme le type `String` est très utilisé, une particularité de ce type est que l'on accède à l'attribut principal de l'objet, la chaîne de caractères, comme si c'était un type primitif :

```
System.out.println(ecole); // affiche "INSA-Lyon"
```

Les objets de type String sont fournis avec un grand nombre de méthodes permettant une manipulation facile de ces objets. Les quatre principales méthodes permettent d'accéder à chaque caractère de la chaîne, de connaître le nombre de caractères de la chaîne (sa longueur), d'extraire une partie de la chaîne et de comparer le contenu de deux chaînes :

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

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

```
String ecole = new String("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.

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

- `equals(String s)` : cette méthode renvoie une variable de type booléen qui vaut vrai si et seulement si la chaîne de caractères *s* est la même que la chaîne de caractères de l'objet sur lequel on appelle la méthode :

```
String ecoleA = new String("INSA-Lyon");
String ecoleB = new String("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
    }
}
```

Une opération fréquente consiste à "coller" bout-à-bout deux chaînes de caractères. On appelle cela la concaténation. L'opérateur associé est + :

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

Il peut être nécessaire de pouvoir convertir un type primitif en une chaîne de caractères ou réciproquement. La conversion d'un type primitif en une chaîne de caractères se fait à l'aide de l'instruction

```
String.valueOf(valeurPrimitive);
```

```
int valeur = 22;
String chaine = String.valueOf(valeur); // chaine 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(i);             // renvoie une valeur de type char
```

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

3.4.2 Les objets propres

Cette section couvre une partie du programme de deuxième année.

On peut également définir ses propres objets. C'est ce que nous allons apprendre à faire ici, en illustrant notre propos sur l'exemple du début de section.

Les classes Les classes permettent de définir des types ou des catégories d'objets (on pourra prendre la métaphore du moule des objets). Elles contiennent la description des objets, (ex : les abonnés), c'est-à-dire qu'elles définissent les attributs et les méthodes communes aux objets d'un certain type.

Les objets construits à partir de ces classes sont des *instances*. Les instances sont des éléments créés depuis une classe : un abonné particulier. La syntaxe de déclaration d'une classe est :

```
class NomType {
    // le corps de la classe vient ici
    // les attributs
    // les méthodes
}
```

On commence généralement par déclarer les attributs de la classe :

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

Le constructeur Toute classe doit contenir au moins une méthode particulière appelé *constructeur*. Cette méthode porte le même nom que la classe et est appelée lorsque l'on instancie un nouvel objet. C'est cette fonction qui généralement initialise les valeurs des attributs. Cette méthode ne renvoie aucune valeur.

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

    Abonne() { // premier constructeur
        nom =  new String("");
        prenom = new String("");
        nbLivres = 0;
        nbJours = 0;
        age = 0;
    }
    Abonne(String nom, String prenom, int nbLivres, int nbJours) {
        // deuxième constructeur
        this.nom =  new String(nom);
        this.prenom = new String(prenom);
        this.nbLivres = nbLivres;
        this.nbJours = nbJours;
    }
}
```

Il peut exister plusieurs constructeurs qui se distinguent par leurs paramètres formels. Ainsi, en fonction des paramètres effectifs (nombre de paramètres ou types des paramètres) passés au constructeur au moment de l'appel, l'un ou l'autre des constructeurs sera appelé. Par exemple, l'instruction `Abonne()` déclenchera l'exécution du premier constructeur, alors que l'instruction `Abonne("Dupond", "Luc",4,0)` déclenchera l'exécution du second constructeur qui instancie les attributs aux valeurs correspondant à l'abonné Luc Dupond.

Le mot clé `this` permet de spécifier que l'attribut que l'on référence est celui de l'objet en cours de construction. En effet, 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 de type String du constructeur.

Exemple Regardons maintenant comment on peut construire notre programme qui indique les abonnés qui ont emprunté un livre depuis plus de 21 jours. Nous venons de voir la classe `Abonne` qui contient les attributs d'un abonné. Ajoutons une méthode `renseigneUnAbonne` qui permet de spécifier les nouvelles valeurs des attributs :

```
void renseigneUnAbonne(String nom, String prenom, int nbLivres, int nbJours) {
    this.nom =  new String(nom);
    this.prenom = new String(prenom);
    this.nbLivres = nbLivres;
    this.nbJours = nbJours;
}
```

La classe `EnsembleDesAbonnes` va contenir l'ensemble des abonnés de la bibliothèque. Ses attributs sont alors un tableau d'abonnés (`lesAbonnes`) et un entier qui indique le nombre d'abonnés. Son constructeur alloue l'emplacement mémoire nécessaire au tableau `lesAbonnes` et instancie chacun des abonnés :

```
class EnsembleDesAbonnes {
    //Attributs de la classe EnsembleDesAbonnes
    Abonne[] lesAbonnes;
    int nbAbonnes;
    //Constructeur
    EnsembleDesAbonnes(int nbAbonnes) {
        lesAbonnes = new Abonne[nbAbonnes];
        this.nbAbonnes = nbAbonnes;
        int i;
        for(i = 0; i < this.nbAbonnes; i++) {
            lesAbonnes[i] =  new Abonne(); // instancie chaque abonné
        }
    }
```

Les attributs de la classe doivent être initialisés dans le constructeur, et en particulier, les attributs de type non-primitifs doivent être instanciés via le mot clé `new` qui réserve l'emplacement mémoire de cet attribut.

Enfin, la méthode `estEnRetard`, qui renvoie la liste des abonnés dont le dernier emprunt a été effectué il y a plus de 21 jours, compte le nombre d'abonnés en retard, puis crée un nouvel objet `EnsembleDesAbonnes` pouvant contenir tous les abonnés en retard, puis recopie l'adresse des abonnés en retard dans l'objet `resultat` :

```
EnsembleDesAbonnes estEnRetard() {
    int i;
    int nbEnRetard = 0;
```

```
        for(i = 0; i < lesAbonnes.length; i++) {
            if( (lesAbonnes[i].nbJoursEmprunts > 21)
              &&(lesAbonnes[i].nbLivresEmpruntes > 0)) {
                nbEnRetard = nbEnRetard + 1;
            }
        }

        EnsembleDesAbonnes resultat = new Abonnes(nbEnRetard);
        int j = 0;
        for(i = 0; i < lesAbonnes.length; i++) {
            if( (lesAbonnes[i].nbJoursEmprunts > 21)
              &&(lesAbonnes[i].nbLivresEmpruntes > 0)) {
                resultat.lesAbonnes[j] = this.lesAbonnes[i];
                j = j + 1;
            }
        }
        return resultat;
    }
```

Le tableau `lesAbonnes` de l'objet resultat contient des références vers les abonnés du tableau `lesAbonnes` de l'objet en cours de définition.

Enfin, la méthode `main` de la classe `EnsembleDesAbonnes` instancie un objet `mesAbonnes` comportant 4 abonnés, et appel la méthode `estEnRetard` sur cet objet qui renvoie un nouvel objet `abonnésEnRetard` qui contient un tableau de références vers les abonnés en retard :

```
public static void main (String[] args) {
    EnsembleDesAbonnes mesAbonnes = new EnsembleDesAbonnes(4);
    mesAbonnes.lesAbonnes[0].renseigneUnAbonne("Dupond", "Luc",4,0);
    mesAbonnes.lesAbonnes[1].renseigneUnAbonne("Martin", "Jeanne",1,30);
    mesAbonnes.lesAbonnes[2].renseigneUnAbonne("Vaus", "Paul",2,22);
    mesAbonnes.lesAbonnes[3].renseigneUnAbonne("Bon", "Jean",0,27);
    EnsembleDesAbonnes abonnesEnRetard = mesAbonnes.estEnRetard();

    // abonnesEnRetard contient les références à Jeanne Martin et Paul Vaus

}
```

4 Glossaire

Adresse : Indique la localisation d'une information dans la mémoire.

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.

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 ou une méthode, 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. La définition d'une méthode s'appelle également déclaration. Elle consiste à définir le nom de la méthode, le type de la valeur retournée, 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`).

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

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 de 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 constituant un exemplaire de la classe.

Instancier : Réserver l'espace mémoire nécessaire pour stocker toutes les valeurs de l'objet. De manière plus générale, fabriquer un exemplaire d'un élément à partir d'un modèle qui lui sert en quelque sorte de moule.

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 méthode 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.

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.