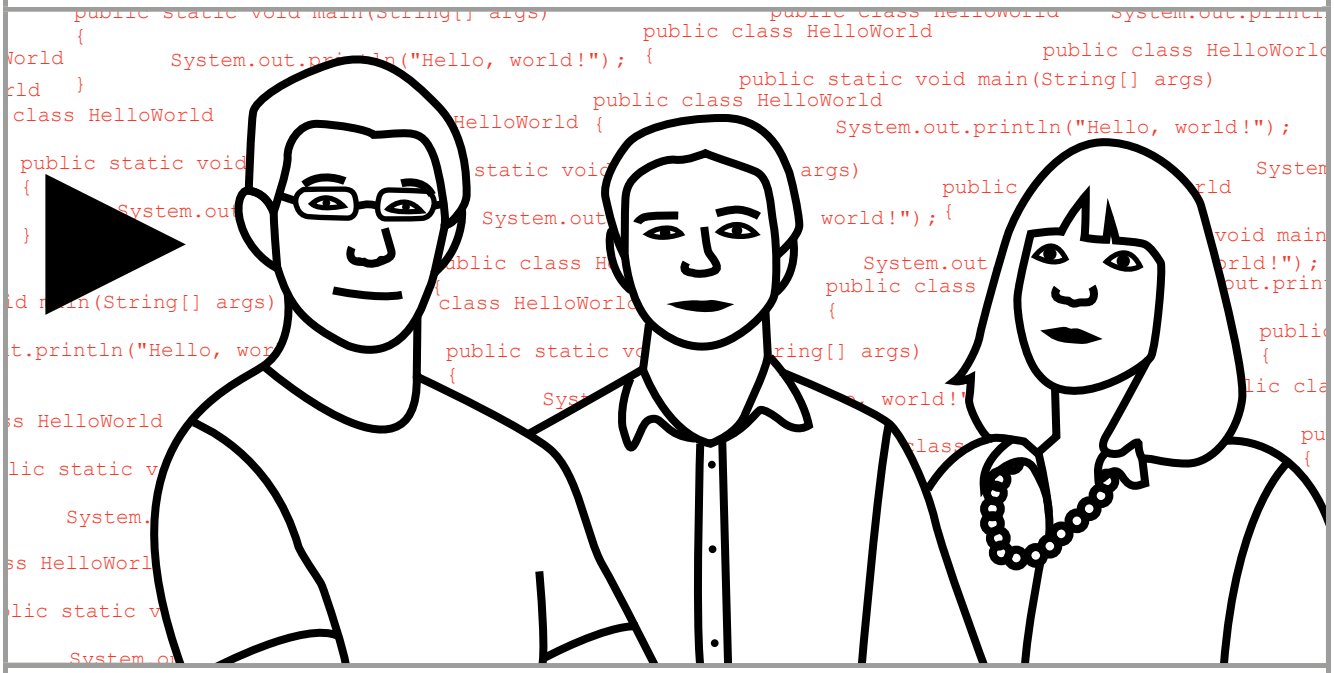
INITIATION À LA PROGRAMMATION EN JAVA

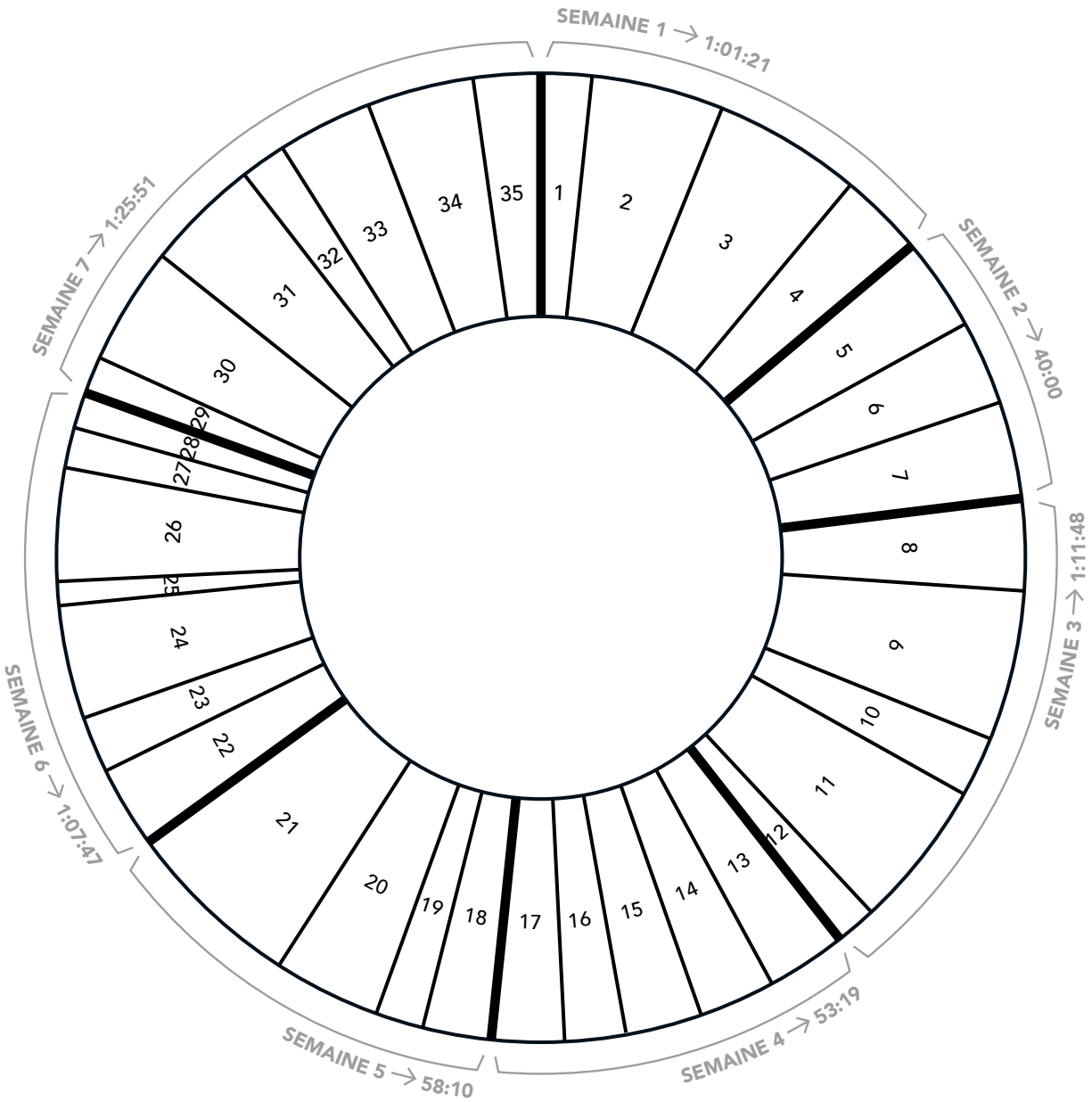


Jamila Sam,
Jean-Cédric Chappelier
et Vincent Lepetit

LEARN FASTER, LEARN BETTER!

BOOCs EPFL

Initiation à la programmation en Java
Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit



CONTENU

SEMAINE 7 : ÉTUDE DE CAS

1. INTRODUCTION

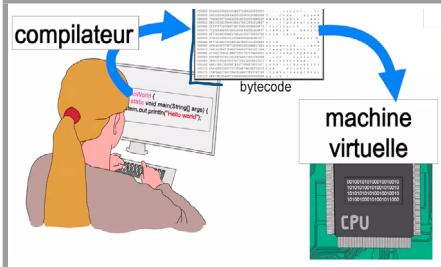


FIGURE 1

1:16 7:26

Les étapes de la traduction d'un programme.

La programmation consiste à écrire des séquences d'instructions dans un langage compréhensible par les humains, tel que le Java, qui seront finalement traduites en instructions élémentaires directement exécutables par le microprocesseur de l'ordinateur. Pour réaliser cette traduction, un programme en langage Java, rédigé dans des fichiers, doit être traduit en une représentation intermédiaire indépendante de l'ordinateur : le « bytecode ». C'est ce que l'on appelle la phase de compilation. Les instructions en « bytecode » sont ensuite interprétées par un programme appelé « machine virtuelle » qui les traduit en instructions pour le microprocesseur.

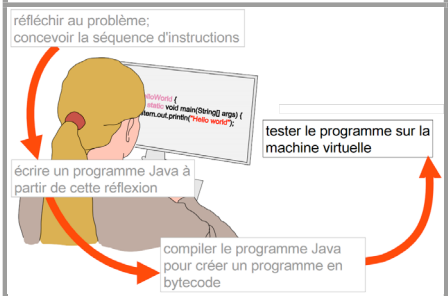


FIGURE 2

2:50 7:26

Étapes de la programmation.

ÉTAPES DE LA PROGRAMMATION

Pour un problème donné, il faut :

– réfléchir à la tâche que l'ordinateur devra exécuter et sa description sous forme de séquence d'instructions ;
– traduire cette séquence d'instructions en un programme Java ;
– le compiler en « bytecode ».

Si le programme ne respecte pas les règles du langage Java, la compilation va échouer. Dans ce cas, il faut corriger le programme. Pour cela on peut s'aider des messages d'erreurs fournis par le compilateur. Une fois le programme compilé, il est possible que son comportement ne soit pas celui prévu lors de la conception. L'erreur provient d'une séquence d'instructions correcte dans le langage Java mais mal conçue. Il faut donc revenir sur l'analyse du programme et repenser la séquence d'instructions voire la tâche elle-même. Cette suite d'étapes est illustrée sur la figure 2.

ÉCRITURE DU PREMIER PROGRAMME

Pour écrire un programme Java, il faut ouvrir un environnement de développement. Nous utilisons Eclipse dans ce cours. Nous allons écrire un programme qui affiche à l'écran le message « Hello, world ! » :

```
public class HelloWorld
{
	public static void main(String[] args)
	{
		System.out.println("Hello, world!");
	}
}
```

COMPILATION ET EXÉCUTION

Avant d'exécuter ce programme, il faut le compiler. Eclipse compile le code à la volée, au fur et à mesure de la rédaction. Si la compilation se déroule correctement, Eclipse permettre de lancer le programme en « bytecode » compilé et afficher le message voulu. L'exécution du programme se fait sur Eclipse en appuyant sur le bouton en forme de flèche verte.

EN CAS D'ERREUR

Si vous avez fait une faute (comme une faute de frappe par exemple), le programme ne va pas compiler et Eclipse va afficher un message d'erreur. Le message d'erreur indique où se trouve la faute pour vous permettre de la corriger. La meilleure façon de corriger vos erreurs est de le faire dans leur ordre d'apparition dans le fichier. En effet, une erreur peut souvent en entraîner d'autres plus loin dans le fichier. La figure 3 nous fournit un exemple de message d'erreur.

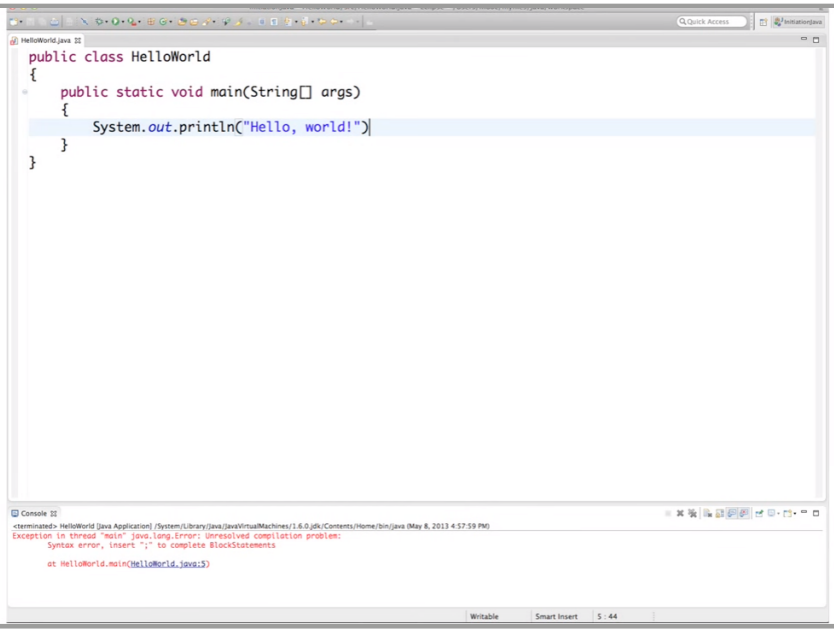


FIGURE 3

6:18 7:26

Exemple d'erreur : l'oubli d'un point-virugle.

2. VARIABLES

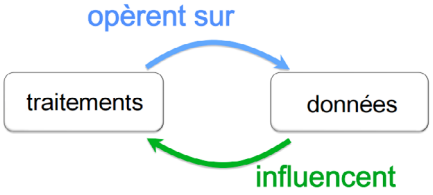


FIGURE 1

2:31 19:28

Interrelation entre les traitements et les données.

TRAITEMENTS ET DONNÉES

Un programme n'est rien d'autre qu'une séquence d'instructions travaillant avec des données. Ces séquences d'instructions sont assimilables à la notion de traitement. Les traitements opèrent sur les données, et les données influencent à leur tour les traitements opérés : selon la nature des données, différents types de traitements sont mis en œuvre.

Une des briques de base pour exprimer des traitements est la notion d'expression, par exemple une expression arithmétique. Pour que les traitements opèrent sur une donnée, il est nécessaire de retrouver la valeur qui lui est associée à différents endroits du programme par le truchement de son nom. Une zone mémoire avec un nom est appelée « variable ».

VARIABLES

Une ***variable*** est une zone en mémoire qui possède trois caractéristiques :
- un ***identificateur***, qui est le nom par lequel la donnée est désignée ;
- une ***valeur***, le contenu de la variable (un nombre par exemple) ;
- un ***type***, qui définit le « genre » de donnée de la variable. Il peut s'agir par exemple d'un entier ou du type chaîne de caractères.

DÉCLARATION DES VARIABLES

Pour mémoriser une donnée au travers d'une variable, il faut ***déclarer*** cette variable, c'est-à-dire spécifier son nom et son type selon l'une des syntaxes suivantes :

```
type identificateur;
type identificateur = valeur_initiale;
```

Lors de la déclaration, on peut omettre l'***initialisation*** (c'est-à-dire le fait de donner une valeur initiale à la variable). La figure 2 offre des exemples de déclarations valides en Java. Il faut cependant être prudent puisqu'une variable non initialisée ne peut pas être utilisée. Dans ce cas, il faut donner une première valeur à la variable avant de l'utiliser, ceci se fait au moyen de l'***affectation***.

Le type associé à la variable au moment de sa déclaration est fondamental. Il conditionne le type de traitement que l'on peut réaliser avec la variable en question. Il faut être attentif au fait que le type d'une variable ne peut pas changer : lorsque l'on déclare une variable d'un certain type, cette variable garde ce type jusqu'à la fin de son existence dans le programme.

Il est licite en Java de déclarer plusieurs variables sur une même ligne en séparant le nom des variables par une virgule et en ne déclarant le type qu'une seule fois. Il ne faut cependant pas en abuser car cette pratique nuit à la lisibilité du programme.

```
int m = 1;
int p = 1, q = 0;
double x = 0.1, y;
```

on peut déclarer plusieurs variables simultanément.
Ne pas en abuser

FIGURE 2

9:59 19:28

Exemple de déclaration de variables.

NOM DE VARIABLES

Il existe un certain nombre de règles à respecter concernant le choix du nom des variables :
– le nom peut être constitué uniquement de lettres, de chiffres et des deux symboles « _ » et « $ » ;
– le premier caractère est nécessairement une lettre ou un symbole ;
– le nom ne doit pas être un mot-clé réservé par le langage Java (par exemple « `if` ») ;
– les majuscules et les minuscules sont autorisées mais pas équivalentes ; les noms `ligne` et `Ligne` désignent deux variables différentes.

En plus de ces règles de nommage, il existe des conventions qu'il n'est pas impératif de respecter pour la compilation du programme, mais que la plupart des programmeurs Java appliquent. Par exemple si le nom d'une variable est constitué de plusieurs mots, ils sont séparés par des majuscules alors que le premier mot commence par une minuscule : `unExemple`.

TYPES DE VARIABLES

Les trois types élémentaires fondamentaux en Java sont :
– `int`, pour les valeurs entières (*integer* en anglais) ;
– `double`, pour les nombres à virgule ;
– `char`, pour les caractères.

AFFECTATIONS

Pour changer la valeur d'une variable déjà déclarée, il faut utiliser la notion d'affectation. Une affectation se fait à l'aide du signe `=` selon la syntaxe suivante :

```
identificateur = expression;
```

Il est important de noter qu'une affectation n'est pas une égalité mathématique. L'exécution d'une affectation s'effectue en deux temps, la partie droite de l'affectation est évaluée en premier puis stockée dans la variable à gauche du signe `=`. La valeur calculée à droite et le type de la variable doivent être les mêmes.

Considérons l'exemple suivant :
Cas 1. `a = b;`
Cas 2. `b = a;`

En mathématique, ces deux lignes signifient que `a` et `b` ont les mêmes valeurs dans les deux cas. Cependant, en Java, les résultats que l'on obtient à la fin sont différents. Dans le cas 1, on copie la valeur de `b` dans `a` alors que dans le cas 2, on copie la valeur de `a` dans `b`. Dans les deux cas `a` et `b` se retrouvent finalement avec les mêmes valeurs, mais dans le cas 1, `a` et `b` ont la valeur que `b` contient avant l'affection, alors que dans le cas 2, ils ont la valeur que `a` contient avant l'affectation !

DÉCLARATION DE CONSTANTES

Il peut arriver que la valeur d'une variable ne doive pas changer après sa déclaration ; il s'agit d'une constante. Dans ce cas, il faut ajouter le mot-clé `final` devant sa déclaration. Par convention, les noms des constantes sont en majuscules, chaque mot est séparé par un caractère « __ ».

3. VARIABLES : LECTURE / ÉCRITURE

AFFICHAGE À L'ÉCRAN

On peut comprendre comment afficher une variable à l'écran à partir de l'exemple suivant : Après avoir déclaré et initialisé une variable `n` qui prend comme valeur 4, on écrit :

```
System.out.println("la variable n contient" + n + ".");
```

afin d'afficher le message « La variable `n` contient 4 ». Cette ligne de code est composée de deux parties principales. La moitié gauche (`System.out.println`) fait référence à la ***sortie standard*** ou ***terminal***, et à droite se trouve le message que l'on souhaite envoyer vers le terminal. Dans le message, on peut distinguer les ***chaînes littérales*** entre guillemets (`" "`) qui vont être affichées directement dans le terminal et les ***expressions*** Java telles que la lettre `n` qui doivent être évaluées avant d'être affichées. Pour séparer les différents types de messages, on utilise le signe `+` qui effectue une concaténation des messages. Cette syntaxe est résumée dans la figure 1. Notons que si l'on remplace `println` par `print`, le message s'affiche sans effectuer un retour à la ligne.

Une expression peut être plus complexe qu'une simple variable. Ainsi, `2 * n` affiche la valeur de n multipliée par 2. Il faut prendre garde à ajouter alors des parenthèses lorsqu'il peut y avoir une ambiguïté entre le signe + utilisé pour concaténer deux messages et celui qui désigne l'addition arithmétique.

```
System.out.println("n plus nCarre = " + n + nCarre);
```

affiche `n plus nCarre = 416` alors que

```
System.out.println("n plus nCarre = " + (n + nCarre));
```

affiche bien `n plus nCarre = 20`.

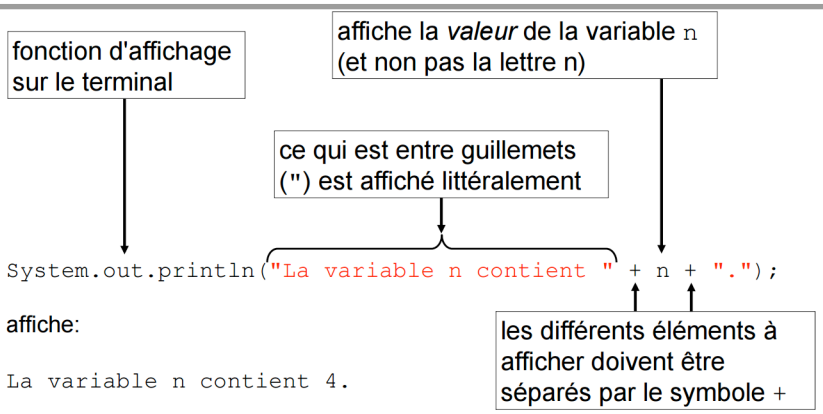


FIGURE 1

3:07 21:35

L'affichage sur l'écran.

ÉCHANGE DE VALEUR

Supposons que l'on ait déclaré et initialisé deux variables `a` et `b`. Pour échanger leurs valeurs, il faut avoir recours à une troisième variable intermédiaire comme l'illustrent les figures 2 et 3.

```
int a = 1;
int b = 2;

a = b;
b = a;

System.out.println(a + ", " + b);
```

Affiche:
2, 2

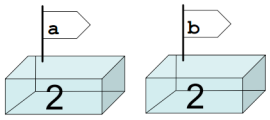
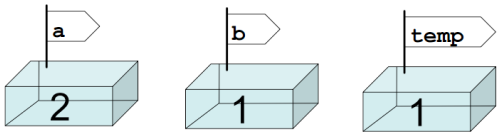


FIGURE 2

9:29 21:35

Exemple d'échange incorrect de valeur.




```
int a = 1;
int b = 2;
int temp = a;

a = b;
b = temp;
```

FIGURE 3

11:00 21:35

Échange correct de valeur entre deux variables.

LECTURE AU CLAVIER

La valeur affectée à une variable n'est pas nécessairement fournie de manière explicite dans le code du programme, il peut s'agir d'une valeur lue au clavier par exemple. La lecture d'un entier que l'on souhaite affecter à la variable `n` se fait selon les étapes suivantes:

- Il faut d'abord importer la classe Scanner pour la rendre visible au compilateur; pour cela on écrit une fois au début du programme la ligne:
  `import java.util.Scanner;`
- On peut maintenant créer une variable de type `Scanner`, en utilisant la syntaxe de cet exemple:
  `Scanner keyb = new Scanner(System.in);`
- La variable `keyb` peut être utilisée autant de fois que nécessaire pour demander des valeurs au clavier. Pour affecter à `n` une valeur lue au clavier on écrit:
  `int n = keyb.nextInt();`

Il faut d'abord importer la classe `Scanner` pour la rendre visible au compilateur:

```
import java.util.Scanner;
```

On peut maintenant créer un objet `Scanner`, par exemple:

```
Scanner keyb = new Scanner(System.in);
```

`keyb` est une variable qu'on peut utiliser pour demander des valeurs au clavier.
Par exemple:

```
int n = keyb.nextInt();
```

FIGURE 4

13:21 21:35

Lecture d'une valeur depuis le clavier.

L'instruction «`int n = keyb.nextInt();`» arrête le programme jusqu'à ce que l'utilisateur entre un entier au clavier et appuie sur la touche `return`. La valeur entrée est ensuite affectée à la variable `n` et le programme reprend son cours. Il existe d'autres méthodes similaires :

- `nextDouble()` permet de lire une valeur de type `double`.
- `nextLine()` permet de lire une ligne complète constituée de tous les caractères tapés par l'utilisateur jusqu'à l'appui de la touche `return`. L'utilisation de cette méthode est décrite dans la figure 5.

```
Scanner keyb = new Scanner(System.in);
String s = keyb.nextLine();
System.out.println("Vous avez saisi : " + s);
```

FIGURE 5

17:56 21:35

Lecture d'une ligne complète tapée au clavier. `s` est une chaîne de caractères. Après l'exécution de l'instruction :
`String s = keyb.nextLine();`
`s` contient tous les caractères tapés par l'utilisateur, jusqu'à l'appui de *return*.

Lors de l'utilisation de `nextLine()`, il faut prendre garde à ne pas involontairement lire des chaînes de caractères vides. En effet, `nextLine()` permet de lire tous les caractères tapés par l'utilisateur jusqu'au retour à la ligne, donc dans le cas où l'utilisateur entre :

```
14
euros
43
```

Avec le programme :

```
int i = keyb.nextInt();
String s1 = keyb.nextLine();
String s2 = keyb.nextLine();
```

La variable `i` contient `14`, la chaîne de caractères `s1` est vide et `s2` contient «`euros`». `s1` contient ce qui suit `14` jusqu'à la fin de la ligne, c'est-à-dire rien. Cet exemple est également décrit dans la figure 6.

```
int i = keyb.nextInt();
String s1 = keyb.nextLine();
String s2 = keyb.nextLine();

Si on tape:
25 francs
23 francs
➔ i contient 25, s1 contient francs, s2 contient 23 francs

Si on tape:
14
euros
43
➔ i contient 14, s1 est vide, s2 contient euros !

car nextLine() lit ce qui suit 14 jusqu'à la fin de la ligne, c'est-à-dire rien !
```

FIGURE 6

21:06 21:35

Difficultés de `nextLine()`.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en Java
Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

4. EXPRESSIONS

Une ***expression*** apparaît par exemple dans une affectation de la forme «`nom_de_variable = expression;`». Elles calculent une valeur qui doit être du même type que la variable. Il peut s'agir d'une simple valeur littérale (comme `4` ou `3.14`) ou d'une formule contenant des ***opérateurs*** (`n * n` par exemple).

TYPES DES VALEURS LITTÉRALES

Le type d'une expression est important puisqu'il doit être identique à celui de la variable qui lui est associée. Or une expression peut être une valeur littérale, donc ces valeurs ont toutes un type qu'il est important de connaître. Ainsi :

– `1` est de type `int`.
– `1.0` et `1.` sont de type `double`. Cependant, il vaut mieux écrire `1.0` plutôt que `1.` qui est moins lisible.
– Une valeur de type `double` peut également être écrite en notation scientifique. Par exemple, dans le cas `double x = 1.3e3;` la variable `x` est de type `double` et vaut $1.3 \times 10^3 = 1300$.

OPÉRATEURS

Les opérateurs usuels sont présents en Java : `+` pour l'addition, `-` pour la soustraction, `*` pour la multiplication, `/` pour la division. Si la division se fait entre valeurs de type `int`, il s'agit de la ***division entière***. Par exemple, `1/2` vaut `0`, mais `1/2.0` vaut bien `0.5`. L'opérateur ***modulo***, `%`, ne peut être utilisé qu'avec des valeurs de type `int` et renvoie le reste de la division entière. `11 % 4` renvoie donc la valeur `3`.

On dispose également des opérateurs `+=`, `-=`, `*=`, `/=` qui correspondent à certaines opérations courantes. Ainsi, `a += 5;` correspond à `a = a + 5;`. De la même façon, `a *= b;` et `a = a * b;` sont identiques. Les opérateurs `++` et `--` permettent respectivement d'***incrémenter*** et de ***décrémenter***, c'est-à-dire d'ajouter et de soustraire 1 à une variable. L'expression `++i;` est équivalente à `i = i + 1;`. Ces opérateurs sont particulièrement utilisés avec les instructions `for` étudiées lors des leçons 8 à 10.

AFFECTATION D'UNE VALEUR DÉCIMALE À UNE VARIABLE ENTIÈRE.

En Java, il est impossible d'affecter une valeur décimale à une variable entière. Le message d'erreur produit par le compilateur est représenté dans la figure 1. En revanche, il est possible d'affecter une valeur entière à une variable de type décimale. Par exemple, on peut écrire `int n = 3;` suivi de `double x = 2 * n;`.

En Java, il est **impossible** d'affecter une valeur décimale par exemple de type `double` à une variable de type `int`:

Exemple:

```
double x = 1.5;
int n = 3 * x;  // Erreur !!!
```

Le compilateur produit le message d'erreur suivant :

```
error: possible loss of precision
        n = 3 * x;
              ^
  required: int
  found: double
```

FIGURE 1

7:14 12:52

Perte de précision lors d'une affectation.

FONCTIONS MATHÉMATIQUES

Java fournit certaines fonctions et constantes mathématiques usuelles qui s'utilisent selon les notations `Math.nomFonction(expression1, expression2, …)` ; et `Math.nomConstante`. Par exemple, `Math.sin(x)` ; est la fonction sinus et `Math.PI` fournit la valeur de π. On peut trouver l'ensemble des fonctions disponibles sur le site de la documentation Java :
http://docs.oracle.com/javase

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en Java
Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

5. BRANCHEMENTS CONDITIONNELS

STRUCTURES DE CONTRÔLE

Jusqu'ici, tous les programmes que nous avons écrits exécutaient leurs instructions les unes après les autres sans dépendre des données présentes. Dorénavant, afin que les traitements soient conditionnés par les valeurs des variables, nous aurons recours aux ***structures de contrôle***. Il en existe trois :

- les ***branchements conditionnels*** qui permettent de choisir l'exécution de certaines lignes de code en fonction de conditions sur les données ;
- les ***itérations*** qui sont des boucles permettant d'exécuter des lignes de codes un certain nombre de fois (leçons 8-10) ;
- les ***boucles conditionnelles*** répètent des instructions en fonction d'une condition évaluée sur des données (leçon 11).

BRANCHEMENTS CONDITIONNELS

Les branchements conditionnels permettent de choisir l'exécution de certaines parties du programme. Par exemple, pour afficher un message particulier si une valeur `n` est plus petite que `5` on écrit :

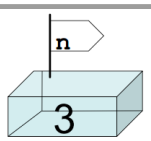
```
if (n < 5) {
    System.out.println("Votre nombre est plus petit que 5. ");
} else {
    System.out.println("Votre nombre est plus grand ou egal a 5. ");
}
```

Le mot-clé `if` marque le début du branchement conditionnel. Il est suivi d'une condition entre parenthèses. Si cette condition est vérifiée, le bloc d'instructions situé entre les deux premières accolades est exécuté. On peut ajouter de manière optionnelle le mot-clé `else` pour introduire un deuxième bloc d'instructions qui est exécuté si la condition que l'on a testée au départ est fausse. Les deux possibilités sont illustrées dans les figures 1 et 2.

```
System.out.print("Entrez votre nombre:");
int n = scanner.nextInt();

if (n < 5) {
    System.out.println("Votre nombre est plus petit que 5.");
} else {
    System.out.println("Votre nombre est plus grand ou egal a 5.");
}

System.out.println("fin du programme");
```



Ce qui s'affiche dans la fenêtre Terminal:

```
Entrez votre nombre:
3
Votre nombre est plus petit que 5.
fin du programme
```

FIGURE 1

5:05 13:39

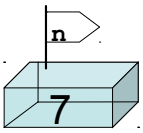
Cas où la condition du branchement est vérifiée.

```
int n;

cout << "Entrez votre nombre:" << endl;
cin >> n;

if (n < 5) {
  cout << "Votre nombre est plus petit que 5." << endl;
} else {
  cout << "Votre nombre est plus grand ou egal a 5." << endl;
}

cout << "Au revoir" << endl;
```



Ce qui s'affiche dans la fenêtre Terminal:

```
Entrez votre nombre:
7
Votre nombre est plus grand ou egal a 5.
Au revoir
```

FIGURE 2

5:35 13:39

L'autre bloc d'instructions est exécuté si la condition est fausse.

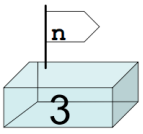
Lorsqu'un bloc ne contient qu'une seule instruction, il n'est pas obligatoire d'utiliser des accolades pour délimiter les instructions à exécuter. Néanmoins, une bonne pratique est de toujours les utiliser car cela facilite l'ajout futur d'instructions.

Une instruction `if` peut ne pas avoir de partie `else`. Dans ce cas, le bloc d'instruction est exécuté lorsque la condition est vérifiée, sinon le programme continue normalement comme le montre la figure 3.

```
System.out.print("Entrez votre nombre: ");
int n = scanner.nextInt();

if (n < 5) {
    System.out.println("Votre nombre est plus petit que 5.");
}

System.out.println("Au revoir");
```



Ce qui s'affiche dans la fenêtre Terminal:

```
Entrez votre nombre: 7
Au revoir
```

FIGURE 3

9:02 13:39

Un branchement conditionnel avec un bloc.

CHOIX IMBRIQUÉS

Les blocs d'instructions exécutés dans les branchements conditionnels sont quelconques et peuvent donc contenir d'autres instructions `if`, il s'agit alors de ***choix imbriqués***. Il ne faut pas en abuser, car le code devient vite illisible. Un exemple détaillé de choix imbriqué est présent dans la figure 4.

```
if (x == y) {
  if (y == z) {
    System.out.println("Les trois valeurs sont egales.");
  } else {
    System.out.println("Seules les deux premieres valeurs sont egales.");
  }
} else {
  if (x == z) {
    System.out.println("Seules la 1ere et la 3eme valeurs sont egales.");
  } else {
    if (y == z) {
      System.out.println("Seules les deux dernieres valeurs sont egales.");
    } else {
      System.out.println("Les trois valeurs sont differentes.");
    }
  }
}
```

FIGURE 4

13:16 13:39

Utilisation d'un choix imbriqué.

6. CONDITIONS

OPÉRATEURS DE COMPARAISON

Dans la leçon précédente, nous avons comparé des valeurs dans le cadre des branchements conditionnels. Ceci nous permettait de formuler des ***conditions simples***, c'est-à-dire des conditions qui comparent deux expressions grâce à un ***opérateur de comparaison***. Les opérateurs de comparaison du langage Java sont :

– `<` qui signifie « inférieur à » ;
– `>` qui signifie « supérieur à » ;
– `==` qui signifie « égal à », à ne pas confondre avec `=` qui représente l'affectation ;
– `<=` qui signifie « inférieur ou égal à » ;
– `>=` qui signifie « supérieur ou égal à » ;
– `!=` qui signifie « différent de ».

L'évaluation d'une expression retourne toujours un résultat ***booléen***, c'est-à-dire soit `true` (vrai) qui signifie que la condition est vérifiée, soit `false` (faux) qui indique que la condition est fausse. Un exemple d'utilisation des conditions simples est donné dans la figure 1.

Les opérateurs de comparaison permettent de comparer non seulement les valeurs de deux variables, mais aussi les valeurs de deux expressions de façon plus générale. Lorsque l'expression se complexifie, il est conseillé de mettre les termes entre parenthèses, pour rendre l'expression plus lisible.

```java
int a = 1;
int b = 2;

if (a == b) {
  System.out.println("Cas 1");
} else {
  System.out.println("Cas 2");
}

if (2 * a == b){
 System.out.println("b est egal au double de a.");
}
```

affiche

```
Cas 2
b est egal au double de a.
```

FIGURE 1

4:51 12:13

Exemple d'utilisation des conditions simples.

BOOCs EPFL

Initiation à la programmation en Java
Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

OPÉRATEURS LOGIQUES

Il est souvent nécessaire de formuler des conditions en combinant des conditions simples. Ceci se fait au moyen d'***opérateurs logiques***:

- L'opérateur `&&` (ET) permet de vérifier si deux conditions sont simultanément vérifiées. L'évaluation d'une expression avec `&&` est vraie si et seulement si les deux conditions qu'il évalue sont vraies. La figure 2 constitue un exemple concret de l'utilisation de `&&`.
- L'opérateur `||` (OU) retourne `true` lorsqu'au moins l'une des deux conditions qu'il évalue est vraie comme l'illustre la figure 3.
- L'opérateur `!` (NON) retourne la négation de la valeur de la condition à sa droite. Ainsi, `!(a < b)` est vraie si `(a < b)` est fausse, et est fausse si `(a < b)` est vraie.

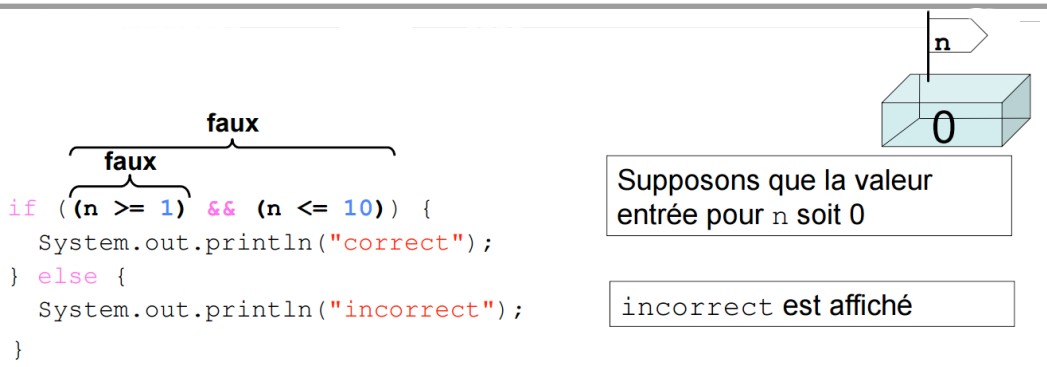


FIGURE 2

7:58 12:13

Exemple d'utilisation de l'opérateur logique ET.

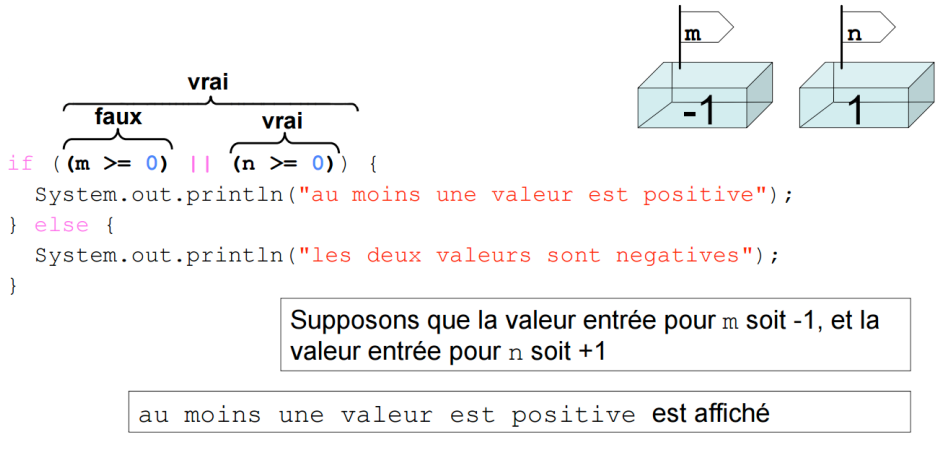


FIGURE 3

10:37 12:13

Exemple d'utilisation de l'opérateur logique OU.

7. ERREURS DE DÉBUTANT, LE TYPE BOOLEAN

ERREURS CLASSIQUES

Les programmeurs débutants commettent souvent les mêmes erreurs lors de l'écriture de branchements conditionnels, nous les reproduisons ici pour que vous puissiez les éviter :

– Le test d'égalité s'écrit `==` et non pas `=`. Le code `if (a = 1)` n'est donc pas accepté par le compilateur.

– L'exécution du code suivant peut paraître étonnante :

```
if (a == 1);
  System.out.println("a vaut 1");
```

Quelle que soit la valeur de `a`, le programme affichera toujours le message `a vaut 1`. L'erreur provient du point-virgule après la condition du `if` qui est considéré comme une instruction qui ne fait rien. Le code suivant est identique pour le compilateur :

```
if (a == 1)
  ;
System.out.println("a vaut 1");
```

Il est donc clair que l'instruction `System.out.println("a vaut 1");` est en réalité située après le `if` et n'est donc pas soumise à sa condition.

– L'indentation en Java ne permet pas de délimiter des blocs d'instructions, c'est le rôle des accolades. Ainsi, comme l'illustre la figure 1, il est impératif de délimiter un bloc d'instruction de plus d'une ligne, sans quoi l'on reçoit une erreur indiquant la présence d'un `else` sans `if` associé.

Ne pas oublier les accolades, l'indentation ne suffit pas:

```
if (n < p)
  System.out.println("n est plus petit que p");
  max = p;
else
  System.out.println("n est plus grand ou egal a p");
```

génère à la compilation l'erreur:

```
error: 'else' without 'if'
```

Voici une meilleure présentation du code précédent:

```
if (n < p)
  System.out.println("n est plus petit que p");

max = p;
else
  System.out.println("n est plus grand ou egal a p");
```

FIGURE 1

3:13 14:08

Oubli des accolades dans un bloc d'instructions.

LE TYPE BOOLEAN

Le type `boolean` (booléen en français) est le type des conditions. Il permet de déclarer des variables contenant une valeur de vérité, c'est-à-dire uniquement `true` ou `false`. L'initialisation des booléens se fait directement à l'aide de `false` et `true` ou à l'aide de conditions. On peut utiliser un booléen comme condition dans un `if`. La figure 2 fournit un exemple de déclaration, d'initialisation et d'utilisation de booléens.

```
int a = 1, b = 2;

boolean c = true;
boolean d = (a == b);
boolean e = (d || (a < b));

if (e) {
  System.out.println("e vaut true");
}
```

FIGURE 2

13:58 14:08

Usage des booléens dans un programme.

8. ITÉRATIONS : INTRODUCTION

BOUCLE FOR

Une ***boucle*** `for` est une itération (il s'agit donc plus généralement d'une structure de contrôle) qui permet de répéter un nombre donné de fois le même bloc d'instructions. Une boucle `for` est donc idéale pour effectuer une action telle qu'afficher les carrés des entiers de 0 à 4, c'est ce que fait le programme suivant :

```
for(int i = 0; i < 5; ++i) {
      System.out.println("le carre de" + i + "vaut" + i * i);
}
```

La boucle commence par le mot-clé `for` suivi de la déclaration et de l'initialisation d'une variable qui s'effectuent une fois avant d'entrer dans le corps de la boucle (le bloc d'instructions entre accolades qui est exécuté à chaque tour de boucle). Une condition détermine si la boucle doit continuer à exécuter son bloc d'instructions. Lorsque sa valeur avant l'exécution est fausse, on sort de la boucle. Finalement une incrémentation est effectuée à la fin de chaque tour de boucle pour changer la valeur du compteur de boucle. La situation est résumée dans la figure 1.

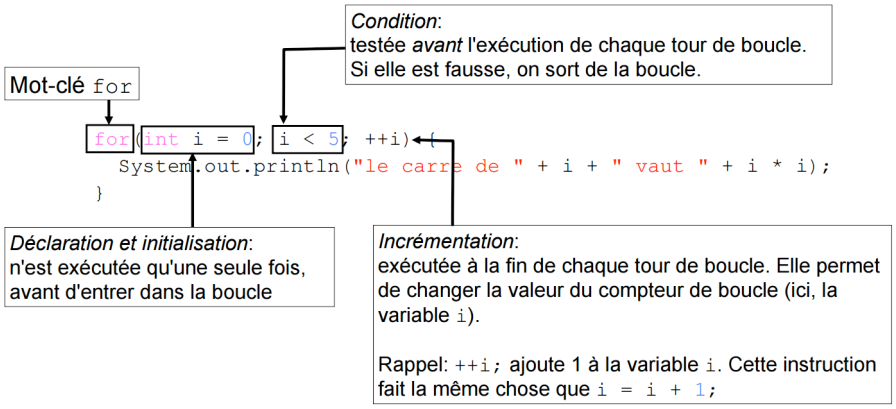


FIGURE 1

2:22 13:29

Exemple de boucle `for` affichant le carré des cinq premiers entiers.

Comme pour le `if`, les accolades ne sont obligatoires que si plusieurs instructions forment le bloc à exécuter. Il est cependant conseillé de toujours garder les accolades afin de minimiser les erreurs. La variable initialisée dans le `for` (ici `i`) n'est déclarée que pour l'intérieur de la boucle et n'est donc pas utilisable à l'extérieur. Notons que si la condition fournie dans la boucle `for` ne devient jamais fausse, les instructions de la boucle seront répétées indéfiniment. La figure 2 nous fournit un exemple de boucle `for` permettant d'afficher la table de multiplication de 5.

On peut remplacer:

```
System.out.println("5 multiplie par 1 vaut " + 5 * 1);
System.out.println("5 multiplie par 2 vaut " + 5 * 2);
System.out.println("5 multiplie par 3 vaut " + 5 * 3);
System.out.println("5 multiplie par 4 vaut " + 5 * 4);
System.out.println("5 multiplie par 5 vaut " + 5 * 5);
...
```

par

```
for(int i = 1; i <= 10; ++i) {
  System.out.println("5 multiplie par " + i + " vaut " + 5 * i);
}
```

La variable `i` prend ici les valeurs de 1 à 10.

FIGURE 2

9:01 13:29

Affichage d'une table de multiplication grâce à `for`.

9. ITÉRATIONS : APPROFONDISSEMENT ET EXEMPLES

EXEMPLE D'AUTRES FORMES DE BOUCLES FOR

Le comportement d'une boucle `for` ne se limite pas à l'incrémentation d'un compteur entre deux valeurs comme l'illustrent ces boucles (on suppose que les variables ne sont pas modifiées dans le corps de la boucle) :

- `for(int p = 0; p < 10; p += 2)` ici, la variable `p` prendra les valeurs de 0, 2, 4, 6, 8.
- `for(int k = 10; k > 0; --k)` ici, la variable `k` prendra les valeurs 10, 9, 8 … jusqu'à 1.
- `for(int i = 0; i >= 0; ++i)` ici, la condition est toujours vraie et la boucle est répétée indéfiniment, donc la variable `i` prendra toutes les valeurs positives que le type `int` peut représenter.

ERREURS AVEC LES BOUCLES FOR

Une boucle `for` peut ne pas s'arrêter (on parle alors de ***boucle infinie***). Cela se produit lorsque la condition est toujours vraie, et peut se produire pour plusieurs raisons :

- On peut s'être trompé sur la condition, par exemple : `for(int i = 0; i > -1; ++i)`.
- On peut s'être trompé sur l'incrémentation, dans la boucle `for(int i = 0; i < 10; ++j)`, `j` est incrémenté au lieu de `i`, la valeur de `i` reste donc 0 et la boucle ne s'arrête pas.

Au contraire, ajouter un point-virgule après une boucle `for` aura pour conséquence d'exécuter son corps une seule fois. En effet dans l'exemple suivant

```
for (int i = 0; i < 10; ++i);
  System.out.println("bonjour");
```

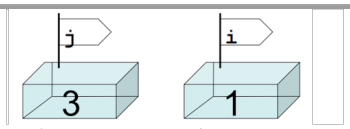
le point-virgule après le `for` est considéré comme une instruction (qui ne fait rien). L'exemple est donc identique au code suivant :

```
for (int i = 0; i < 10; ++i)
 ;
System.out.println("bonjour");
```

où l'affichage de *bonjour* se situe après la boucle.

BOUCLES IMBRIQUÉES

Rien ne nous empêche de placer une boucle `for` dans le corps d'une autre boucle `for`. Ainsi la figure 1 illustre une telle boucle imbriquée permettant d'afficher les tables de multiplication de 2 jusqu'à 10.




```
for(int j = 2; j <= 10; ++j) {
  System.out.println("Table de multiplication par " + j + ": ");
  for(int i = 1; i <= 10; ++i) {
    System.out.println(j + " multiplie par " + i + " vaut " + j * i);
  }
}
```

```
Table de multiplication par 2:
2 multiplie par 1 vaut 2
2 multiplie par 2 vaut 4
...
2 multiplie par 10 vaut 20
Table de multiplication par 3:
3 multiplie par 1 vaut 3
```

FIGURE 1

Affichage des tables de multiplication de 2 à 10.

10. ITÉRATIONS : QUIZ

Afin de tester notre connaissance des boucles, essayons de trouver ce qu'affiche l'exécution du code de la figure 1. Il est conseillé de répondre au Quiz avant de lire la suite.

Que s'affiche-t-il quand on exécute le code :

```
for(int i = 0; i < 3; ++i) {
  for(int j = 0; j < 4; ++j) {
    if (i == j) {
      System.out.print("*");
    } else {
      System.out.print(j);
    }
  }
  System.out.println("");
}
```

A:
```
*123
*123
*123
```

C:
```
****
****
****
```

?

B:
```
012*
012*
012*
```

D:
```
*123
0*23
01*3
```

FIGURE 1

0:12 8:53

Première question.

La bonne réponse est la D. En effet, la variable `i` prend des valeurs de 0 à 2 pour afficher 3 lignes alors que la variable `j` prend les valeurs 0, 1, 2 et 3 pour afficher les quatre caractères sur chaque ligne. Notons que dans la boucle `for` de `j`, c'est la fonction `System.out.print` qui est utilisée et qui n'effectue donc pas de retour à la ligne.

La deuxième question, sur la figure 2 est un peu plus difficile que la première.

Que s'affiche-t-il quand on exécute le code :

```
for(int i = 0; i < 3; ++i) {
  for(int j = 0; j < i; ++j) {
    System.out.print(j);
  }
  System.out.println("");
}
```

A:
```
0
01
```

C:
rien

?

B:
```
0
01
012
```

D:
```
0123
0123
0123
```

FIGURE 2

5:20 8:53

Deuxième question.

La bonne réponse est la A. La variable `i` prend comme dans la question précédente les valeurs 0, 1 et 2. Trois lignes sont donc affichées alors que dans chaque ligne la variable `j` va prendre les valeurs entre 0 et `i - 1`. Donc à la première ligne il n'y aura aucun caractère, à la deuxième ligne il n'y en aura qu'un seul, etc.

11. BOUCLES CONDITIONNELLES

Nous avons vu (leçons 8 à 10) que les itérations permettent de répéter une partie du programme. On peut les utiliser lorsque le nombre de répétitions est connu avant d'entrer dans la boucle. Il arrive cependant que l'on ne sache pas combien de fois la boucle devra être exécutée. On utilise alors une ***boucle conditionnelle***, c'est-à-dire une boucle `do…while` ou `while`. La figure 1 fournit un exemple d'utilisation de la boucle `do…while` où l'on souhaite répéter le corps de la boucle tant que l'utilisateur du programme n'a pas entré une valeur strictement positive.

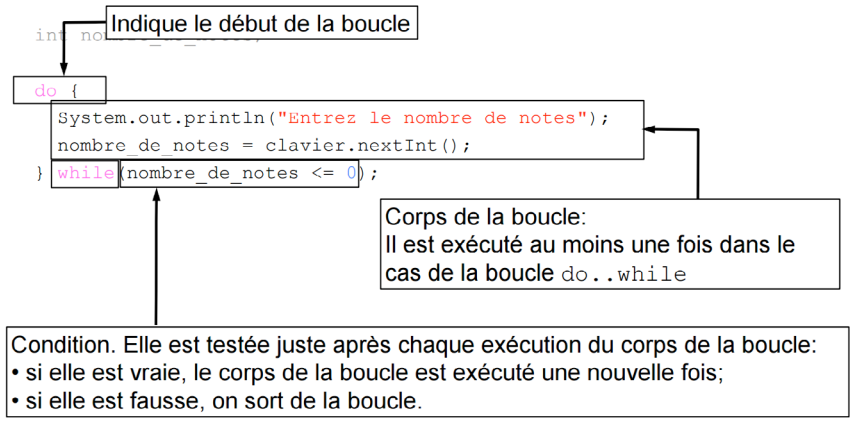


FIGURE 1

3:04 21:37

Exemple d'exécution d'une boucle `do…while`.

SYNTAXE DE LA BOUCLE DO...WHILE

Une boucle `do…while` permet de répéter le corps de la boucle tant qu'une condition est respectée tout en garantissant que le corps sera exécuté au moins une fois. La syntaxe de cette boucle est :

```
do {
    bloc
}   while(condition);
```

La condition se formule de la même façon que pour une instruction `if`. Remarquons que si la condition ne devient jamais fausse, les instructions de la boucle seront répétées indéfiniment. De plus, cette condition est évaluée a posteriori, c'est-à-dire après une première exécution du corps. Il existe une variante de la boucle `do…while` où la condition est évaluée a priori, c'est la boucle `while`.

SYNTAXE DE LA BOUCLE WHILE

Une boucle `while` s'écrit de la manière suivante :

```
while(condition) {
    bloc
}
```

Le principe est similaire à celui de la boucle `do…while` à la différence que la condition est testée avant d'entrer dans la boucle. Si la condition est fausse, les instructions dans la boucle ne sont donc pas exécutées. La différence entre les deux types de boucles conditionnelles est illustrée dans la figure 2.

```
int i = 100;
do {
  System.out.println("bonjour");
} while (i < 10);
```

affichera une fois `bonjour`.

```
int i = 100;
while (i < 10) {
 System.out.println("bonjour");
}
```

n'affichera rien.

Dans les 2 cas, la condition `i < 10` est fausse.

FIGURE 2

9:20 21:37

Différence entre `while` et `do…while`.

Une erreur commune à ne pas commettre est d'ajouter un point-virgule à la fin de la condition du while puisque ce point serait considéré comme le corps de la boucle (une instruction qui ne fait rien).

CHOIX DE BOUCLE À UTILISER

Quand le nombre d'itérations est connu avant d'entrer dans la boucle, il vaut mieux utiliser la boucle `for`. Si les instructions doivent être effectuées au moins une fois, la boucle `do…while` est adaptée. Sinon, on utilise la boucle `while`.

12. BLOCS D'INSTRUCTION

BLOCS

En Java, les instructions peuvent être regroupées en ***blocs*** identifiés par des délimiteurs de début et de fin : `{` et `}`. Nous les avons déjà utilisés par exemple dans les structures de contrôle, mais on peut regrouper les instructions en blocs en dehors de ces structures de contrôle comme l'illustre la figure 1.

En Java, les instructions peuvent être regroupées en **blocs**.

Les blocs sont identifiés par des délimiteurs de début et de fin : { et }

Exemple:

```
{
  Scanner keyb = new Scanner(System.in);
  int i;
  double x;

  System.out.println("Valeurs pour i et x : ") ;
  i = keyb.nextInt();
  x = keyb.nextDouble();
  System.out.println("Vous avez saisi : i = " + i +
                     ", x = " + x);
}
```

FIGURE 1

0:40 5:59

Exemple de bloc.

Les blocs en Java ont une grande autonomie puisqu'ils peuvent contenir leurs propres déclarations et initialisation de variables. Ces variables déclarées à l'intérieur d'un bloc sont appelées ***variables locales*** (au bloc). Elles ne sont accessibles qu'à l'intérieur du bloc où elles ont été déclarées. Les variables déclarées en dehors de la fonction `main` sont de ***portées globales*** (à la classe), elles sont accessibles dans toute la classe. Pour ces dernières, on distinguera (dans le cours de Programmation Orientée Objet) les ***variables de classes*** et les ***variables d'instances***.

Une bonne pratique est de déclarer les variables au plus près de leur utilisation comme on peut l'observer sur la figure 2.

Par exemple, si la variable `j` n'est pas utilisée après la condition,

écrivez:

```
if (i != 0) {
  int j = 0;

  ...
  j = 2 * i;
  ...
}
```

plutôt que:

```
int j = 0;
if (i != 0) {

  ...
  j = 2 * i;
  ...
}
```

FIGURE 2

2:44 5:59

Déclarer les variables au plus près de leur utilisation.

PORTÉE

Cette notion d'endroits où l'on peut utiliser une variable s'appelle la ***portée***. La portée d'une variable est l'ensemble des lignes de code où cette variable est accessible et a donc un sens. En Java, on ne peut pas utiliser le nom d'une variable déclarée plus globalement pour déclarer une autre variable. La notion de portée est par exemple essentielle aux boucles `for`: la déclaration d'une variable à l'intérieur de la boucle est locale à son bloc et ses instructions de test et d'incrémentation comme on peut le voir dans la figure 3.

La déclaration d'une variable à l'intérieur d'une itération est une déclaration **locale au bloc de la boucle**, et aux deux instructions de test et d'incrément:

```
for(int i = 0; i < 5; ++i) {
  System.out.println(i);
}
// A partir d'ici, on ne peut plus utiliser ce i
```

FIGURE 3

5:53 5:59

Portée dans les boucles `for`.

13. TABLEAUX : INTRODUCTION

Nous avons vu que tout programme manipule des données stockées sous la forme de variables, constituées d'une valeur et d'un type. Jusqu'à présent nous n'avons étudié que des types simples, comme des entiers, mais il est souvent nécessaire de représenter des données de type plus évolué, de manipuler des ensembles de données comme un tout. Par exemple, si l'on a besoin de modéliser un ensemble d'âges, qui est un ensemble homogène d'entiers, il est naturel de penser à un tableau que l'on peut parcourir et manipuler comme un tout.

Âge
20
35
26
38
22

Nom	Taille	Âge	Sexe
Dupond	1.75	41	M
Dupont	1.75	42	M
Durand	1.85	26	F
Dugenou	1.70	38	M
Pahut	1.63	22	F

FIGURE 1

1:02 11:31

Exemples de tableaux de données.

EXEMPLE

Pour illustrer l'utilité des tableaux, prenons un exemple. Imaginons que l'on souhaite écrire un programme manipulant des scores de joueurs. Le but est simplement d'afficher les scores, de calculer la moyenne de tous les scores, et pour chacun de calculer son écart à la moyenne. Essayons d'écrire ce programme pour deux joueurs seulement, avec les moyens dont nous disposons jusqu'ici, c'est-à-dire les données de type élémentaire.

```
...
Scanner keyb = new Scanner(System.in);

// Lecture des donnees et calculs
System.out.println ("Score Joueur 1:");
int score1 = keyb.nextInt();
System.out.println ("Score Joueur 2:");
int score2 = keyb.nextInt();
// Calcul de la moyenne
double moyenne = (score1 + score2);
moyenne /= 2;
// Affichages
System.out.println("Score     Ecart Moyenne");
System.out.println(score1 + " " + (score1 - moyenne));
System.out.println(score2 + " " + (score2 - moyenne));
...
```

FIGURE 2

2:58 11:31

Solution avec les moyens actuels.

L'exemple de la figure 2 est simple, mais comment ferions-nous si nous avions plus de joueurs ? La figure 3 montre le même programme avec cinq joueurs.

```
System.out.println ("Score Joueur 1:");
int score1 = keyb.nextInt();
System.out.println ("Score Joueur 2:");
int score2 = keyb.nextInt();
...
System.out.println ("Score Joueur 5:");
int score5= keyb.nextInt();

// calcul de la moyenne
double moyenne = (score1 + score2 + score3 + score4 + score5);
moyenne /= 5;


// Affichages
System.out.println("Score          Ecart Moyenne");
for(int i = 1; i <= 5; ++i) {
  System.out.println(scorei + "     " + scorei -  moyenne);
}
```

FIGURE 3

4:33 11:31

Même programme avec plus de joueurs.

Nous devons créer autant de variable score qu'il y a de joueurs et complexifier le calcul de la moyenne. Pour les affichages, nous pourrions penser à une boucle, cependant ici la notation `scorei` n'est pas comprise par le compilateur. Cette solution n'est pas satisfaisante puisque le programme prendrait des dimensions ingérables si nous avions non pas cinq joueurs, mais 100 ou 1000. De plus, cette approche ne gère pas le cas où le nombre de joueurs n'est pas connu au moment de l'écriture du programme. Ainsi, pour ce type de programme, l'outillage que nous avons actuellement ne suffit pas, nous devons introduire les tableaux. La figure 4 montre à quoi ressemblerait le programme précédant en utilisant la notion de tableaux.

```
System.out.print ("Donnez le nombre de joueurs:");
int n = keyb.nextInt();
if (n > 0) {
   double moyenne = 0;
   int scores [] = new int[n];

    // Lecture des scores
   for (int i = 0; i < n; ++i) {
      System.out.println ("Score Joueur " + i + " :");
      scores[i] = keyb.nextInt();
      moyenne += scores[i];
   }
   moyenne /= n; // calcul de la moyenne

    // Affichages
   System.out.println(" Score " + " Ecart Moyenne");
   for (int i = 0; i < n ; ++i) {
      System.out.println(scores[i] + " " + (scores[i] - moyenne));
   }
}
```

FIGURE 4

6:00 11:31

Même programme en utilisant les tableaux.

L'idée est d'utiliser un nouveau type de donnée, ici le type « tableau d'entiers », dont nous verrons la syntaxe plus en détail dans la leçon 14. L'utilisation de tableaux nous permet d'utiliser une boucle `for` pour afficher les éléments, avec une syntaxe correcte cette fois. Ensuite, nous n'avons pas besoin de connaître le nombre de joueurs au moment de l'écriture du programme, et celui-ci s'écrira de la même façon que nous ayons 5, 100 ou 1000 joueurs.

En résumé, un tableau en programmation est un type de données qui nous permet de manipuler une collection de valeurs du même type. Ici nous avons défini un tableau de valeurs de type `int`, mais nous pouvons définir des tableaux d'autres types, notamment de tableaux.

LES TYPES DE TABLEAUX

En programmation on peut considérer qu'il existe quatre types de tableaux (fig. 5), en fonction de si la taille est connue au départ et de si celle-ci peut varier lors de l'utilisation.

		taille initiale connue *a priori* ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	`ArrayList`	`ArrayList`
	non	tableaux de taille fixe	tableaux de taille fixe

FIGURE 5

11:07 11:31

Différents types de tableaux en Java.

La plupart des langages de programmation n'offrent pas un type spécifique pour chaque configuration vue ici. En Java, on dispose principalement de deux types, seule la distinction de savoir si la taille doit varier à l'exécution est faite. Si la taille varie au cours du temps alors nous utiliserons le type prédéfini `ArrayList`, sinon nous utiliserons des tableaux de taille fixe, que nous allons étudier dans les prochaines leçons.

14. TABLEAUX : DÉCLARATION

Maintenant que nous savons ce qu'est un tableau, et les différents types de tableaux que l'on peut trouver, nous allons voir comment les déclarer et les initialiser. Nous n'étudierons pour l'instant que des tableaux de taille fixe.

DÉCLARATION D'UN TABLEAU DE TAILLE FIXE

Un tableau se déclare de la même manière que les types simples que nous avons vus, en écrivant le type puis le nom de la variable. On déclare le type d'un tableau en rajoutant des crochets au type des éléments de ce tableau. Par exemple, on déclare un tableau d'entiers de la manière suivante :

```
int[] scores;
```

Notons que Java autorise aussi la syntaxe :

```
int score[];
```

Nous allons voir comment initialiser des tableaux de taille fixe, mais avant toute chose il faut savoir que les tableaux sont des types évolués, comme les chaînes de caractères, et ils se manipulent différemment des types simples vus jusqu'à présent.

Les types évolués sont ***stockés via des références***, c'est-à-dire que les variables de ces types ne stockent pas directement la valeur, mais une référence vers l'endroit où est stockée la vraie valeur (fig. 1). Cela impacte la sémantique de l'affectation, de la comparaison et de l'affichage de ces données.

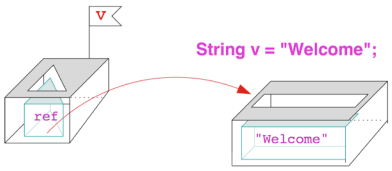


FIGURE 1

3:27 11:09

Stockage des types évolués via une référence.

INITIALISATION D'UN TABLEAU DE TAILLE FIXE

Il existe deux façons en Java d'initialiser les éléments d'un tableau. La première est d'initialiser le tableau avec des éléments donnés au moment de sa déclaration :

```
int[] scores = {1000, 1500, 2000};
```

La seconde manière est d'initialiser un tableau sans élément, puis de le remplir ailleurs dans le programme. Dans ce cas, il faut quand même initialiser le tableau avec une taille fixée, et on utilisera la syntaxe suivante :

```
tableau = new type[taille];
```

Ce qui donne par exemple :

```
int[] scores = new int[4];
```

En pratique cette syntaxe remplit le tableau avec des valeurs par défaut (fig. 2).

int	`0`
double	`0.0`
boolean	`false`
char	`'\u0000'`
`(objet quelconque)`	`(null)`

FIGURE 2

7:53 11:09

Valeurs par défaut des types de base.

Pour remplir ensuite le tableau, on aura besoin d'accéder à ses éléments. On utilise pour cela l'***indexation***, c'est-à-dire que l'on accède aux éléments selon leur position dans le tableau. On utilisera la syntaxe `tab[i]` pour accéder au i-ème élément du tableau `tab`.

Attention toutefois, le premier élément d'un tableau est situé à l'index 0, et non 1 ! Ainsi le dernier élément d'un tableau de taille T est à l'index T-1. De plus, si l'on essaye d'accéder à un élément qui n'est pas dans le tableau, une exception est lancée, ce qui se traduit pour l'instant pour nous par l'arrêt du programme.

15. TABLEAUX : TRAITEMENTS COURANTS

Nous savons déclarer et initialiser un tableau, nous allons maintenant voir les manipulations classiques faites sur un tableau, et les erreurs qu'il faut éviter.

ACCÈS AUX ÉLÉMENTS D'UN TABLEAU DE TAILLE FIXE

Nous aurons souvent besoin d'itérer sur les éléments d'un tableau, de le parcourir. Par exemple, pour afficher les éléments d'un tableau, nous ne pouvons pas simplement faire :

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

En effet, puisque les tableaux sont des types évolués, ceci affiche la référence du tableau (voir leçon 14 sur la déclaration des tableaux). Il paraît alors logique d'utiliser une boucle `for` pour itérer sur un tableau. Il existe deux types de boucle `for` pour accéder aux éléments d'un tableau.

Le premier type de boucle est la boucle `for` classique telle que nous la connaissons :

```
for(int i=0; i < tab.length; i++) {
    System.out.println(tab[i]);
}
```

Cette boucle utilise l'indexation pour accéder au i-ème élément du tableau. Aussi, il est nécessaire de connaître la taille du tableau. Pour cela il existe en Java l'instruction `tableau.length`. Attention, cette tournure représente la taille *possible* du tableau, c'est-à-dire la taille avec laquelle il a été initialisé, et non sa taille effective (du nombre d'éléments que l'on a explicitement stocké).

Il existe un deuxième type de boucle `for`, qui itère sur un ensemble de valeurs :

```
for(int val: tab) {
    System.out.println(val);
}
```

Cette boucle est plus compacte, mais elle présente des inconvénients :
- Elle ne permet pas de modifier le contenu du tableau.
- Elle ne permet d'itérer que sur un seul tableau à la fois : on ne peut pas comparer deux tableaux par exemple.
- Elle ne permet l'accès qu'à un seul élément, on ne peut pas récupérer l'élément précédent ou le suivant.
- Elle itère d'un pas en avant seulement.

ERREURS COURANTES AVEC LES TABLEAUX

Une erreur courante avec les tableaux est d'utiliser un indice inadéquat pour accéder à un élément. Il faut retenir qu'un indice est toujours de type `int` et que, pour un tableau de taille T, celui-ci doit être compris entre 0 et T-1. Attention notamment dans les boucles `for`, car par exemple `tableau[tableau.length]` n'existe pas !

Une autre erreur est de vouloir accéder aux éléments d'un tableau avant sa construction. Nous avons vu (leçon 14) qu'il existe deux manières pour construire un tableau. Il faut bien différencier la déclaration de l'initialisation :

```
int[] entiers1;
int[] entiers2 = {1, 2, 3};
```

La première ligne déclare un tableau mais ne l'initialise pas, tandis que la deuxième déclare et initialise le tableau.

16. TABLEAUX : AFFECTATION ET COMPARAISON

Comme les tableaux en Java sont des types évolués, la sémantique de l'affectation et de la comparaison est différente de celle que nous connaissons avec les types simples.

AFFECTATION DE TABLEAUX

Les types évolués sont manipulés avec des références en Java. Donc lorsque nous réalisons l'affectation `a = b`, c'est la référence qui est copiée et non l'objet (fig. 1).

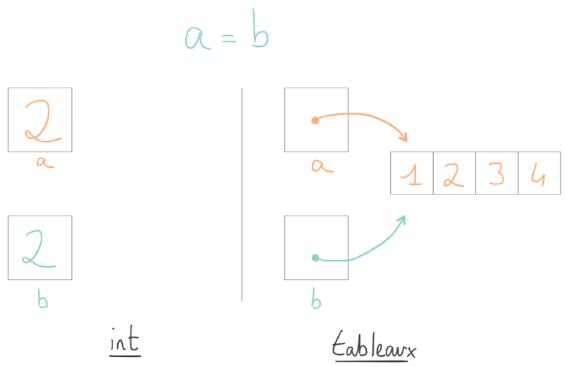


FIGURE 1

0:15 9:03

Affectation de tableaux en Java.

Ceci implique que les variables `a` et `b` sont indépendantes pour des types simples, mais totalement dépendantes pour des tableaux, puisque les deux noms de variables désignent le même objet. Ainsi, si nous modifions `a` après l'affectation, alors `b` sera modifié également (fig. 2).

```
// Les tableaux a et b pointent vers deux emplacements
// différents en memoire
int[] a = new int[10]; // tableau de 10 entiers
int[] b = new int[10]; // tableau de 10 entiers

for (int i = 0; i < a.length; ++i) {
   a[i] = i; // remplissage du tableau pointé par a
}
b = a; // opérateur =  (affectation)
System.out.println("a[2] vaut " + a[2] + " et b[2] vaut " + b[2]);
a[2] = 42;
System.out.println("a[2] vaut " + a[2] + " et b[2] vaut " + b[2]);
```

ce qui affiche :

```
a[2] vaut 2 et b[2] vaut 2
a[2] vaut 42 et b[2] vaut 42
```

FIGURE 2

2:07 9:03

Exemple d'affectation entre deux tableaux.

Nous utiliserons donc l'opérateur `=` avec des tableaux uniquement lorsque nous voudrons avoir deux noms de variable pour le même tableau, ce qui est assez rare en pratique. Pour copier les éléments d'un tableau vers un autre, il faudra explicitement écrire une « copie profonde » (copie au-delà des références « de surface ») :

```
for (int i=0; i < a.length; i++) {
    b[i] = a[i];
}
```

COMPARAISON DE TABLEAUX

Pour des types évolués, l'opérateur `a == b` teste si les variables `a` et `b` référencent le même emplacement mémoire. C'est donc le cas lors de l'affectation `a = b`. En revanche, cet opérateur ne teste pas l'égalité des valeurs contenues dans les tableaux pointés par `a` et `b` ! Pour vérifier l'égalité de contenu des tableaux, il faut écrire explicitement les tests (fig. 3).

```
if (a == null || b == null || a.length != b.length) {
    System.out.println("contenus différents ou nuls");
}
else {
    int i = 0;
    while(i < a.length && (a[i] == b[i])) {
        ++i;
    }
    if (i >= a.length) {
        System.out.println("contenus identiques");
 }
    else {
        System.out.println("contenus différents")
    }
}
```

FIGURE 3

5:17 9:03

Tests d'égalité de contenu des tableaux.

17. TABLEAUX À PLUSIEURS DIMENSIONS

Il est possible en Java de faire des tableaux à plusieurs dimensions. Par exemple, si nous voulons faire un tableau d'élèves où chaque élève est représenté par un tableau de notes. La manière d'implémenter cela est assez logique. Il s'agit simplement d'un tableau de tableaux (attention, le premier tableau stocke des références vers les autres tableaux). Ainsi, il suffit de rajouter un niveau de `[]` lors de la déclaration (fig. 1).

Il est également possible d'initialiser des tableaux multidimensionnels lors de leur déclaration. Dans ce cas, les lignes peuvent avoir des longueurs différentes (fig. 2).

```
double[][] statistiques =
            new double[nbCantons][nbCommunes];

int[][] scores =
        new int[nbJoueurs][nbParties];
```

FIGURE 1

2:16 10:33

Exemples de déclarations de tableaux à plusieurs dimensions.

```
int[][] tableau = {
    { 0, 1, 2, 3, 42 },
    { 4, 5, 6 },
    { 7, 8 },
    { 9, 0, 1 }
};
```

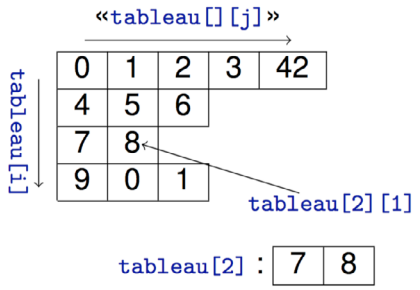


FIGURE 2

4:54 10:33

Initialisation de tableaux à plusieurs dimensions.

ACCÈS AUX ÉLÉMENTS D'UN TABLEAU À PLUSIEURS DIMENSIONS

Pour accéder aux éléments d'un tableau à plusieurs dimensions, il faut indiquer un ou plusieurs index, de la même manière que pour un tableau à une seule dimension. Prenons par exemple le tableau :

```
int[][] tab = { {1, 2}, {3, 4}, {5, 6} };
```

Les éléments `tab[0]`, `tab[1]` et `tab[2]` sont des tableaux de type `int[]`, et valent respectivement `{1, 2}`, `{3, 4}` et `{5, 6}`.

Les éléments `tab[0][0]`, `tab[0][1]`, `tab[1][0]`, etc. sont des valeurs de type `int`, et valent respectivement `1`, `2`, `3`, etc.

PARCOURIR UN TABLEAU À PLUSIEURS DIMENSIONS

Le moyen le plus naturel de parcourir un tableau multidimensionnel consiste à utiliser des boucles for imbriquées. La figure 3 montre comment parcourir un tableau de dimension 2. Dans ce code, la première boucle itère sur les lignes et la seconde sur les colonnes.

```
for(int i = 0; i < y.length; ++i) {
 for(int j = 0; j < y[i].length; ++j) {
      System.out.println(y[i][j]);
 }
}
```

FIGURE 3

8:56 10:33

Parcours d'un tableau à deux dimensions.

Nous avons maintenant vu tout ce qu'il faut savoir sur les tableaux de taille fixe. Il nous reste à voir les tableaux dynamiques, que nous verrons à la leçon 21.

18. STRING : INTRODUCTION

Nous allons maintenant nous intéresser au type `String` qui permet de définir des ***chaînes de caractères***.

TYPE STRING

Pour déclarer une chaîne de caractères, on utilise le type évolué `String`, avec la syntaxe habituelle :

```
String unNom;
```

Pour initialiser une chaîne de caractères, on utilise des guillemets :

```
String message = "Bonjour tout le monde !";
```

TYPE CHAR

Il existe aussi le type simple `char` pour représenter des caractères. Ce type ne peut contenir qu'un seul caractère, et s'initialise en mettant le caractère entre des apostrophes :

```
char c1 = 'm';
char c2 = ' ';
char c3 = '2';
```

AFFECTATION ET COMPARAISON

Comme pour les tableaux, une variable de type `String` contient une référence vers une chaîne de caractères. La sémantique des opérateurs `=` et `==` est donc la même que pour les tableaux (fig. 1). Cependant, une différence avec les autres types évolués est que les littéraux de type `String` occupent une zone mémoire unique. C'est ce qu'on appelle le ***«Pool» des littéraux***. Cela signifie que deux chaînes créées avec les mêmes littéraux seront stockées à la même adresse (fig. 2).

```
String chaine = "";     // chaine pointe vers ""
String chaine2 = "foo"; // chaine2 pointe vers "foo"
chaine = chaine2 ;      // chaine et chaine2 pointent vers "foo"
(chaine == chaine2)     // retourne true
```

FIGURE 1

2:27 9:48

Affectation et comparaison de chaînes de caractères.

```
String chaine1 = "foo";  // chaine1 pointe vers le litteral "foo"
String chaine2 = "foo" ; // chaine2 pointe vers le litteral "foo"
if (chaine1 == chaine2 ) // true : chaine1 et chaine2 contiennent la même
                         //adresse
```

FIGURE 2

3:43 9:48

Conséquences du « Pool » des littéraux.

AFFICHAGE

Comme les variables de type `String` contiennent une adresse vers la zone mémoire contenant les chaînes de caractères, on pourrait penser que le code suivant affiche une adresse (comme pour les types évolués en général):

```
String chaine = "Welcome";
System.out.print(chaine);
```

Cependant l'affichage du type `String` prend en compte l'objet référencé plutôt que la référence. Le code précédent affichera donc bien le message *Welcome*. Nous verrons pourquoi dans le cours sur la POO.

CONCATÉNATION

L'opérateur `+` est défini pour les chaînes de caractères de telle manière que `chaine1 + chaine2` produise une nouvelle chaîne associée à la valeur littérale constituée des valeurs littérales de `chaine1` et `chaine2`. C'est ce que l'on appelle la ***concaténation*** (fig. 3).

Exemple : constitution du nom complet à partir du nom de famille et du prénom :

```
String nom = "Dupont";
String prenom = "Jean";
nom = nom + " " + prenom; // "Dupont Jean"
```

FIGURE 3

5:43 9:48

Concaténation de chaînes de caractères.

La concaténation peut aussi s'appliquer entre une chaîne de caractères et un type simple quelconque (`char`, `int`, etc.).

19. STRING : COMPARAISONS

Nous avons vu que des objets de type `String` initialisés avec des littéraux peuvent être comparés par les opérateurs `==` et `!=`, puisque ceux-ci pointent vers la même zone en mémoire. Cependant ceci n'est vrai que lorsque les chaînes sont initialisées avec des littéraux (fig. 1). Pour comparer les objets référencés au lieu des références, on dispose de l'instruction `equals()`. L'instruction `chaine1.equals(chaine2)` teste si les chaînes de caractères référencées par `chaine1` et `chaine2` sont constituées des mêmes caractères (fig. 2). En pratique, il est préférable d'utiliser l'instruction `equals` plutôt que l'opérateur `==` pour les objets de type `String`.

```
String s1 = "abc";     // s1 pointe vers le littéral "abc"
String s2 = "abc";     // idem (donc même zone mémoire que s1)
String s3 = s2;        // s3 stocke la même adresse que s2
String s4 = s1 + "";   // s4 contient l'adresse d'une nouvelle chaîne
                       // (construite par concaténation)
System.out.println((s1==s2) && (s2==s3)); // affiche true
System.out.println(s4);                   // affiche abc
System.out.println((s1==s4));             // affiche false
```

FIGURE1

0:37 7:08

Comparaison de `Strings` avec `==`.

```
String s1 = "abc";
String s2 = "aBc";
String s4 = s1 + "";

System.out.println(s1.equals(s4)); // true
System.out.println(s1.equals(s2)); // false
```

FIGURE 2

4:28 7:08

Comparaison de `Strings` avec `equals`.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en Java
Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

20. STRING : TRAITEMENTS

Il existe de nombreux traitements que nous pouvons appliquer aux objets de type `String`. Ces traitements s'appellent des ***méthodes*** (ou ***fonctions***) en Java. C'est un élément primordial de la programmation, abordé plus en détail à partir de la leçon 22.

TRAITEMENTS DES CARACTÈRES

- L'instruction `chaine.charAt(index)` donne le caractère occupant la position `index` dans la `String chaine`.
- L'instruction `chaine.indexOf(caractere)` donne la position de la première occurence du `char caractere` dans la `String chaine`, et `-1` si caractere n'est pas dans la chaîne.
- L'instruction `chaine.length()` donne la taille de la `String chaine` en nombre de caractères (attention c'est différent des tableaux, il y a des parenthèses après `length`).

Notons que les caractères sont numérotés comme les éléments d'un tableau (à partir de 0).

```
String s1 = "abcmbx";
int longueur = s1.length();         // 6
char c1 = s1.charAt(0);             // a
char c2 = s1.charAt(longueur - 1); // x
int i = s1.indexOf('b');            // 1
```

FIGURE 1

0:12 15:24

Exemple d'utilisation des traitements.

AUTRES TRAITEMENTS

- Un littéral introduit par l'utilisateur n'est pas dans le « pool » des littéraux. L'instruction `chaine.intern()` permet de l'y mettre explicitement (fig. 2).
- L'instruction `chaine.replace(char1, char2)` construit une nouvelle chaîne en remplaçant dans toute la chaîne `char1` par `char2`. Exemple :

```
String exemple = "abracadabra";
String avecDesEtoiles = exemple.replace('a', '*');
```

Ce code construit la nouvelle chaîne `"*br*c*d*br*"`.

- L'instruction `chaine.substring(position1, position2)` retourne la sous-chaîne comprise entre les indices `position1` (compris) et `position2` (non compris). Exemple :

```
String exemple = "anticonstitutionnel";
String avecDesEtoiles = exemple.substring(4, 16);
```

Ce code construit la nouvelle chaîne `"constitution"`.

```java
Scanner s = new Scanner(System.in);
String response;
do {
   response = s.nextLine();
   //on met le littéral lu dans le pool
   response = response.intern();
   System.out.println("Read: " + response);
   // sans le intern, la boucle ne s'arrête pas!
} while (response != "oui");
```

FIGURE 2

8:39 15:24

Ajout d'un littéral au pool des littéraux.

INTRODUCTION AUX FONCTIONS

Nous avons vu que certains traitements sont spécifiques aux `String`. Ils s'utilisent en fait tous avec la syntaxe particulière suivante :

```
nomDeChaine.nomDeTraitement(arg1, arg2 …);
```

21. TABLEAUX DYNAMIQUES

Nous avons vu en détail les tableaux de taille fixe dans les leçons 13 à 17. Nous allons maintenant étudier les tableaux dynamiques. Les tableaux dynamiques ont la particularité de pouvoir changer de taille pendant l'exécution du programme. On les définit en Java à l'aide du type `ArrayList`. Pour les utiliser, il faut d'abord importer les définitions associées à l'aide de la ligne :

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

Cette ligne est à placer en tout début de fichier.

DÉCLARATION ET INITIALISATION

Une variable correspondant à un tableau dynamique se déclare de la façon suivante :

```
ArrayList<type> identificateur;
```

où *type* est le type des éléments du tableau. Ce type doit obligatoirement être un ***type évolué***.

Un tableau dynamique initialement vide (sans aucun élément) s'initialise de la manière suivante :

```
ArrayList<type> identificateur = new ArrayList<type>();
```

MÉTHODES COURANTES

Comme les `String`, de nombreuses méthodes sont spécifiques aux `ArrayList`. L'utilisation de ces méthodes se fait avec la syntaxe suivante :

```
nomDeTableau.nomDeMethode(arg1, arg2 …);
```

Voici les principes :
- `tableau.size()` renvoie la taille du tableau (de type `int`);
- `tableau.get(i)` renvoie l'élément à l'indice `i` dans le tableau (`i` doit être un entier compris entre `0` et `tableau.size()-1`);
- `tableau.add(valeur)` ajoute valeur à la fin de tableau ;
- `tableau.set(i, valeur)` affecte valeur à la case `i` du tableau ;
- `tableau.isEmpty()` détermine si le tableau est vide ou non (`boolean`);
- `tableau.clear()` supprime tous les éléments du tableau ;
- `tableau.remove(i)` supprime l'élément à l'indice `i` du tableau.

TABLEAU DE TYPES DE BASE

Le type des éléments d'un tableau dynamique ne peut pas être un type simple. Pour remédier à cela, on utilisera donc des types évolués correspondant aux types simples que nous connaissons : `Integer` pour `int`, `Double` pour `double`, etc. La conversion du type simple au type évolué se fait automatiquement (*autoboxing*).

Comme les éléments d'un `ArrayList` sont toujours des références, on les comparera à l'aide de la méthode `equals()`. Cette méthode vue précédemment pour les `String` est en fait commune à tous les types évolués.

22. FONCTIONS : INTRODUCTION

Nous allons maintenant voir un des éléments les plus importants de la programmation en Java, il s'agit des ***fonctions***. Les fonctions en programmation sont des traitements, qui agissent sur des données. Dans un langage orienté objet comme le Java, on préférera l'appellation ***méthode*** plutôt que fonction.

Les méthodes sont des portions de code que l'on définit à un endroit du programme et que l'on peut appeler depuis un ou plusieurs autre(s) endroit(s). Elles permettent d'éviter la duplication de code, le « copier-coller ». En programmation, il faut absolument éviter le copier-coller, puisque cela rend le programme inutilement long, difficile à comprendre et difficile à ***maintenir*** : si l'on veut changer le code, il faut penser à le changer partout.

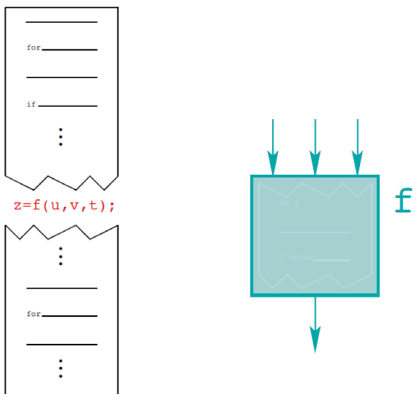


FIGURE 1

8:42 12:00

Illustration d'une méthode.

CARACTÉRISATION D'UNE MÉTHODE

Une méthode fonctionne de la manière suivante : elle prend des valeurs en entrées, réalise des traitements en utilisant ces valeurs et retourne un résultat sous la forme d'une valeur de sortie (fig. 1). Elle est caractérisée par :

- un ***corps*** : la portion de code à exécuter ;
- un ***nom*** : celui par lequel on désignera cette fonction ;
- des ***paramètres*** : ensemble de variables que la fonction prend en entrée, et dont le corps a besoin pour fonctionner ;
- un ***type*** et une ***valeur de retour*** : la sortie de la fonction, ce qu'elle renvoie au reste du programme.

LES TROIS FACETTES D'UNE MÉTHODE

Dans un programme, une méthode comporte trois facettes (fig. 2) :

- L'***entête*** comporte le type de retour, le nom et les paramètres de la méthode. C'est en quelque sorte un résumé de celle-ci, puisque l'on y voit déjà ses entrées et sa sortie (et son nom).
- La ***définition*** est constituée de l'entête et du corps de la méthode.
- L'***appel*** est l'endroit où l'on utilise la méthode. Il contient le nom de la méthode et les arguments que l'on veut lui passer.

En pratique, le ***programmeur concepteur*** (celui qui écrit la définition de la méthode) n'est pas forcément la même personne que le ***programmeur utilisateur*** (celui qui appelle la méthode). Dans ce cas, l'utilisateur n'a pas besoin de voir la définition de la méthode, il n'a besoin que de l'entête de celle-ci. L'entête, communément appelé ***API*** (*Application Programming Interface*), sert donc d'accord entre le concepteur et l'utilisateur.

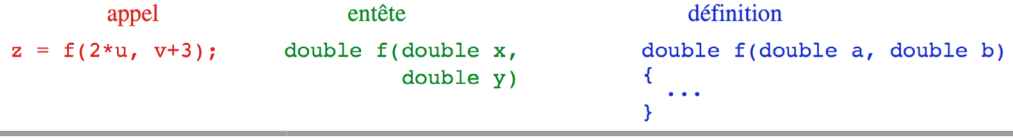


FIGURE 2

9:47 12:00

Les trois facettes d'une méthode.

23. FONCTIONS : APPEL

Dans la leçon 22, on a vu que la notion de fonction est composée de trois facettes : l'entête, la définition et l'appel. Dans cette leçon, on détaille ce qui se passe au moment de l'appel. La figure 1 présente un programme qui affiche la moyenne de deux notes, entrées par l'utilisateur. Le calcul de la moyenne est réalisé au moyen d'un appel de fonction. Les arguments passés à la fonction au moment de l'appel correspondent aux paramètres attendus par la fonction pour qu'elle puisse s'exécuter.

Notons que l'on appelle usuellement ***paramètres*** les données nécessaires à la fonction pour qu'elle puisse s'exécuter telle que décrite dans l'entête, et ***arguments*** les valeurs que l'on passe effectivement à la fonction au moment de l'appel.

ÉVALUATION D'UN APPEL DE MÉTHODE

Dans le cas plus général où une méthode nécessite pour des paramètres entrants pour fournir en sortie une valeur concrète, l'appel de méthode se passe en cinq étapes :

1. Les expressions passées en argument sont évaluées.
2. Les valeurs correspondantes sont affectées aux ***paramètres*** de la méthode.
3. Le corps de la méthode est exécuté avec ces valeurs.
4. L'expression suivant la première commande `return` rencontrée est évaluée.
5. La valeur obtenue est retournée comme ***résultat*** de l'appel : cette valeur remplace l'expression de l'appel.

Dans le cas d'une méthode sans arguments, les étapes 1 et 2 n'ont pas lieu. Dans le cas d'une méthode sans valeur de retour (type de retour `void`), les étapes 4 et 5 n'ont pas lieu. La figure 1 illustre l'évaluation de l'appel d'une méthode.

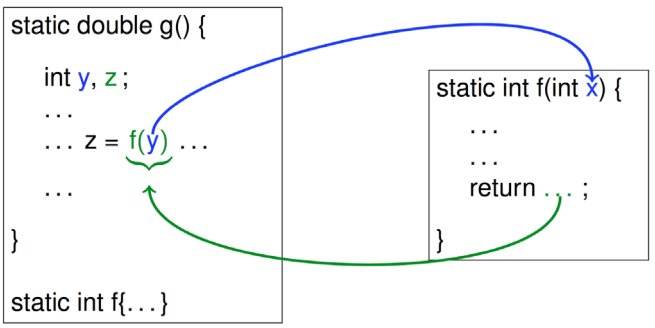


FIGURE 1

6:16 8:17

Schématisation de l'appel d'une méthode.

Il existe un certain jargon utilisé au moment de l'appel :

- « Appeler la méthode `f` » : utiliser la méthode. Exemple : dans le code `x = 2*f(3);`, on *appelle* la méthode `f`.
- « La valeur est passée en argument » : (lors d'un appel) la valeur est copiée dans un paramètre de la méthode. Exemple : dans le code `x = 2*f(3);`, la valeur 3 est *passée* en argument.
- « La méthode retourne la valeur » : l'expression de l'appel de la méthode sera remplacée par la valeur retournée. Exemple : `cos(0)` retourne le cosinus de 0, donc retourne la valeur 1.

24. FONCTIONS : PASSAGE DES ARGUMENTS

Nous avons vu comment est évalué l'appel d'une méthode lorsque les arguments passés à celle-ci sont des valeurs ou des expressions. Nous allons maintenant voir ce qu'il se passe lorsque les arguments passés à une méthode sont des variables. En particulier, si une méthode modifie la valeur d'un paramètre, celui-ci est-il modifié en dehors de la méthode ?

En programmation, de façon générale, on dira que :
– Un argument est ***passé par valeur*** si la méthode ne peut pas le modifier : la méthode créé une ***copie locale*** de cet argument.
– Un argument est ***passé par référence*** si la méthode peut le modifier.

En Java, il n'existe que le passage par valeur, mais cela a des conséquences différentes selon que le type du paramètre est simple (`int`, `double`, etc.) ou évolué (`objet`).

ARGUMENTS DE TYPE SIMPLE

Si un argument passé à une méthode est de type simple, une modification de celui-ci dans cette méthode n'aura pas de conséquences sur sa valeur en dehors de la méthode puisque le passage par valeur crée une copie locale de cet argument. Par exemple, le code de la figure 1 produira l'affichage :

```
x=1 val=2
```

```
public static void main(String[] args) {
 int val = 1;
 m(val);
 System.out.println(" val=" + val);
}

static void m(int x) {
  x = x + 1;
  System.out.print(" x=" + x);
}
```

FIGURE 1

10:28 16:12

Schématisation de l'appel d'une méthode.

ARGUMENTS DE TYPE ÉVOLUÉ

Pour des arguments de type évolué, qui sont pour rappel manipulés via des ***références***, c'est la référence qui est passée à la méthode. Comme en Java les arguments sont passés par valeur, la référence de l'objet passée à la méthode est copiée. Ainsi, si la référence est modifiée dans la méthode, cela n'aura pas d'influence sur la variable en dehors de la méthode. Par exemple, le code de la figure 2 produira l'affichage :

```
x[0]= 100 tab[0]= 1
```

En revanche, même si la référence est copiée, cette copie pointe toujours vers le même objet. Ainsi, si l'objet référencé est modifié dans la méthode, cela ***aura une influence*** sur l'objet même en dehors de la méthode. Par exemple, le code de la figure 3 produira l'affichage suivant :

```
x[0]= 100 tab[0]= 100
```

```
public static void main(String[] args) {
   int[] tab = {1};
   m(tab); // tab (référence)
           // PASSAGE PAR VALEUR AUSSI
   System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
   int[] t = {100};
   x = t; //Modification de la référence
          //(on met une autre adresse dans x)
   System.out.print("x[0]= " + x[0]);
}
```

FIGURE 2

11:52 16:12

Exemple de modification de la référence (type évolué).

```
public static void main(String[] args) {
   int[] tab = {1};
   m(tab);
   System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
   x[0] = 100; // modification de l'objet
               // référencé par x
   System.out.print("x[0]= " + x[0]);
}
```

FIGURE 3

14:19 16:12

Exemple de modification de l'objet référencé (type évolué).

25. FONCTIONS : ENTÊTES

Nous avons vu que les 3 facettes d'une méthode sont l'entête, la définition et l'appel. Nous nous intéressons ici à l'entête. L'entête d'une méthode donne un aperçu de ce que fait la méthode. Elle est constituée de son ***nom***, de ses ***paramètres*** et de son ***type de retour***. La syntaxe est la suivante :

```
type_retour nom(type_1 param_1, …, type_n param_n)
```

Dans ce cours, nous ajouterons toujours le mot-clé `static` au début de chaque entête, mais celui-ci deviendra une exception dans le cours de « Programmation Orientée Objet », où nous verrons sa signification.

Exemples d'entêtes :

```
static double moyenne(double x, double y)
static int nbAuHasard()
```

Dans le cas d'une méthode sans valeur de retour, on utilisera le type spécial `void` comme type de retour. Dans le cas d'une méthode sans paramètres, on écrira simplement une liste de paramètres vide (voir exemple `nbAuHasard` plus haut).

BONNES PRATIQUES

- Une méthode ne doit faire que ce pour quoi elle est prévue, il faut absolument éviter des effets cachés, des « ***effets de bords*** ».
- Il faut choisir des noms pertinents pour les méthodes et les paramètres : cela facilitera la compréhension du code.
- Il est recommandé de toujours commencer par écrire l'entête d'une méthode : c'est un résumé de celle-ci, ce qu'elle prend comme données en entrée et ce qu'elle retourne.

26. FONCTIONS : DÉFINITIONS

Nous avons vu que les trois facettes d'une méthode sont l'entête, la définition et l'appel. Nous nous intéressons ici à la définition. La définition d'une méthode représente la méthode à proprement parler, elle contient tout ce que fait la méthode. La définition est constituée de l'***entête*** et du ***corps*** de la méthode. La syntaxe est la suivante :

```
type nom( liste de paramètres )
{
    instructions du corps de la méthode;
    return expression;
}
```

Nous avons vu ce qu'était l'entête de la méthode dans la leçon précédente. Le corps de la méthode est un bloc de code contenant toutes les instructions de la méthode, pouvant utiliser les paramètres de la méthode. La valeur retournée par la méthode est l'expression suivant le mot-clé `return`.

INSTRUCTION RETURN

L'instruction `return` fait deux choses :
- Elle précise la valeur qui sera fournie par la méthode en résultat.
- Elle met fin à l'exécution des instructions de la méthode.

Le compilateur affichera une erreur concernant cette instruction si :
- Le type de l'expression suivant le mot-clé `return` n'est pas le même que le type de retour défini dans l'entête de la méthode.
- L'instruction `return` n'est pas la dernière instruction du corps de la méthode : celle-ci met fin à l'exécution du corps, donc tout code placé après celle-ci (dans le même bloc) ne sera jamais exécuté.

De plus, il peut y avoir plusieurs instructions `return` dans une méthode, par exemple dans une structure `if else`. Cependant, il faut être sûr qu'une instruction `return` soit exécutée ***dans tous les cas***. Par exemple, une méthode avec une unique instruction `return` placée dans un `if` produira une erreur de compilation, puisque celle-ci n'est pas atteinte dans tous les cas.

Dans le cas d'une méthode sans valeur de retour, il n'y a pas besoin d'instruction `return`. On peut cependant l'utiliser malgré tout pour arrêter la méthode à un endroit précis (par exemple dans un `if`).

MÉTHODE MAIN

Nous utilisons, dans tous nos programmes depuis le début, une méthode spéciale : la méthode `main`. Par convention, tout programme Java doit avoir une méthode `main`. Cette méthode est la première méthode qui est appelée dans notre programme. L'entête autorisée pour la méthode `main` est la suivante :

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

27. FONCTIONS : MÉTHODOLOGIE

Voici un résumé de la méthodologie à suivre lors de la création d'une méthode :

1. Identifier clairement ce que ***doit faire*** la méthode. Il faut se demander pourquoi nous faisons cette méthode, dans quel but. Choisir à ce moment-là le nom de la méthode, et choisir un nom cohérent et parlant.
2. Identifier ce dont la méthode a besoin pour fonctionner : les ***arguments***. Énumérer les entrées de la méthode et leur type.
3. Identifier ce que ***retourne*** la méthode. Est-ce que l'instruction `z = f(…)` a un sens ? Si oui, trouver le type de retour, sinon indiquer `void`.
4. Se préoccuper du « comment ». Maintenant, et seulement maintenant, on peut réfléchir à comment la méthode va faire ce qu'elle doit faire : on écrit alors le ***corps*** de la méthode.

28. FONCTIONS : SURCHARGE

En Java, il est possible de définir plusieurs méthodes qui ont le même nom si le nombre ou le type des paramètres sont différents : c'est ce que l'on appelle la ***surcharge de méthodes***. Elle est utile pour des méthodes de même nature opérant sur des données différentes.

En Java, deux méthodes sont différenciées non seulement par leur nom, mais aussi par le type de leurs paramètres, que l'on appelle la ***signature*** de la méthode. Remarquons que l'on n'a pas le droit d'avoir deux méthodes de même nom et de mêmes paramètres, même si le type de retour est différent. Par exemple, on ne peut pas avoir deux méthodes `int f(int)` et `double f(int)`.

```
static void affiche(int x) {
 System.out.println("entier : " + x);
}
static void affiche(double x) {
 System.out.println("reel : " +  x);
}
static void affiche(int x1, int x2) {
 System.out.println("couple : " + x1 + "," +  x2);
}
```

FIGURE 1

1:51 5:19

Exemple de surcharge de méthode.

29. PUISSANCE 4 : INTRODUCTION

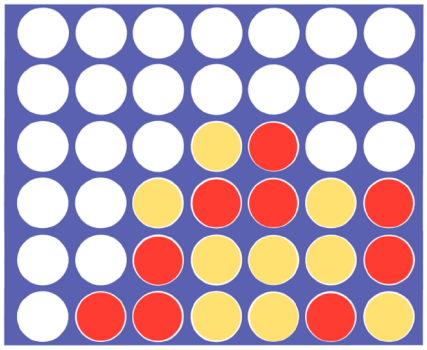


FIGURE 1

0:45 5:34

Illustration du puissance 4.

```
  | | | | | | | |
  | | | | | | | |
  | | | |X| | | |
  | | |X|O|O|X|O|
  | | |O|X|X|X|O|
  | |O|O|X|X|O|X|
==1=2=3=4=5=6=7==

Joueur O :  entrez un
numéro de colonne 5

  | | | | | | | |
  | | | | | | | |
  | | | |X|O| | |
  | | |X|O|O|X|O|
  | | |O|X|X|X|O|
  | |O|O|X|X|O|X|
==1=2=3=4=5=6=7==

Le joueur O a gagne !
```

FIGURE 2

0:49 5:34

Affichage dans la console.

Pour conclure ce cours d'introduction à la programmation, nous souhaitons présenter (leçons 29 à 35) un projet un peu plus conséquent que ce que nous avons vu jusqu'à présent. Ce projet consiste à programmer un jeu de *Puissance 4* (fig. 1).

Rappel des règles du jeu : le jeu se joue à 2 joueurs, sur une grille de 7 colonnes et 6 lignes. Chaque joueur laisse tomber tour à tour un pion de sa couleur dans la colonne de son choix. Le but du jeu est d'aligner 4 pions de sa couleur (fig. 1). Le jeu s'arrête si un joueur y parvient ou si la grille est remplie.

Pour ce projet nous ne ferons pas d'interface graphique, mais un affichage dans la console (fig. 2).

Lors du développement d'un projet, il faut avoir une certaine rigueur et suivre quelques règles :
- Ne pas écrire tout le projet d'un seul coup : il faut ***décomposer le projet*** en sous-parties, et développer le programme par étapes. À chaque étape, ***tester le code développé***.
- Identifier dans l'ordre : les types nécessaires au programme, puis les méthodes qui portent sur ces types.
- Lorsqu'un code se répète ou répond à un besoin précis, le ***modulariser*** à l'aide d'une méthode, et décomposer les méthodes complexes en utilisant d'autres méthodes.

Pour notre jeu, la donnée principale est la grille de jeu et les cases qu'elle contient.
- La grille peut être modélisée simplement et intuitivement par un tableau à deux dimensions (*type_des_elements[][]*). Nous décidons arbitrairement que le premier indice du tableau correspond à la ligne, le deuxième à la colonne, et l'origine se trouve en haut à gauche.
- Une case peut être vide, rouge ou jaune. Il existe plusieurs moyens pour modéliser ce type d'éléments, nous choisissons d'utiliser le type `int`. Ainsi nous définissons des constantes pour chaque type (fig. 3).

```
private final static int VIDE  = 0;
private final static int JAUNE = 1;
private final static int ROUGE = 2;
```

FIGURE 3

3:30 5:34

Constantes du programme.

30. PUISSANCE 4 : PREMIÈRES FONCTIONS

Abordons maintenant nos premières fonctions ; les premières tâches à effectuer : construire le jeu et l'afficher.

INITIALISATION DE LA GRILLE

Il faut écrire une méthode pour initialiser la grille avec des cases vides. Lorsque nous écrivons une méthode il faut d'abord se demander comment nous allons l'utiliser. Si nous avons le code suivant :

```
int[][] grille = new int[6][7];
```

il est logique d'appeler notre méthode comme suit :

```
initialise(grille);
```

Nous voyons que notre méthode prend la grille en argument et ne retourne rien (`void`). Nous avons donc déjà l'entête de la méthode (voir leçon 25 pour plus de détails) :

```
static void initialise(int[][] grille)
```

Il reste maintenant à écrire le corps de notre méthode. Celle-ci est assez simple, puisqu'elle se contente de parcourir toutes les cases pour y mettre la valeur `VIDE` (une constante, voir fig. 3 de la leçon 29). Pour parcourir un tableau à deux dimensions, nous avons vu qu'il faut utiliser deux boucles `for` imbriquées.

La fonction complète est donnée dans la figure 1.

```
static void initialise(int[][] grille)
{
  for(int i = 0; i < grille.length; ++i) {
    for(int j = 0; j < grille[i].length; ++j) {
      grille[i][j] = VIDE;
    }
  }
}
```

FIGURE 1

7:30 17:22

Fonction `initialise`.

AFFICHAGE DE LA GRILLE

Il faut maintenant écrire la méthode qui affiche la grille. Comme précédemment, il est logique d'appeler notre méthode comme suit :

```
affiche(grille);
```

Ainsi l'entête de cette méthode est très similaire à celle de la méthode `initialise`, seul le nom change :

```
static void affiche(int[][] grille)
```

Encore une fois nous devons parcourir toutes les cases de la grille, donc nous utiliserons des boucles `for` imbriquées. Ensuite, pour chaque case, on affiche un caractère différent selon ce qui se trouve sur la case. On implémentera cela avec une structure `if`, `else if`, `else`.

La fonction complète est donnée dans la figure 2. On n'oubliera pas de commenter notre méthode, surtout quand celle-ci utilise des conventions choisies arbitrairement.

```
private final static int VIDE  = 0;
private final static int JAUNE = 1;
private final static int ROUGE = 2;

...
// affiche O pour une case rouge, X pour une case jaune
static void affiche(int[][] grille)
{
  for(int[] ligne : grille) {
    for(int cellule : ligne) {
      if (cellule == VIDE) {
        System.out.print(' ');
      } else if (cellule == ROUGE) {
        System.out.print('O');
      } else {
        System.out.print('X');
      }
    }
    System.out.println();
  }
}

...
```

FIGURE 2

11:34 17:22

Fonction affiche .

Maintenant que nous avons notre méthode fonctionnelle, on peut choisir de l'améliorer pour produire un affichage plus spécifique à notre jeu. Par exemple on peut vouloir rajouter les colonnes, et leur numérotation. La fonction améliorée est donnée dans la figure 3.

```
// affiche O pour une case rouge, X pour une case jaune
static void affiche(int[][] grille)
{
  System.out.println();
  for(int[] ligne : grille) {
    System.out.print(" |");
    for(int cellule : ligne) {
      if (cellule == VIDE) {
        System.out.print(' ');
      } else if (cellule == ROUGE) {
        System.out.print('O');
      } else {
        System.out.print('X');
      }
      System.out.print('|');
    }
    System.out.println();
  }

  System.out.print('=');
  for(int i = 1; i <= grille[0].length; ++i) {
    System.out.print("=" +  i);
  }
  System.out.println("==\n");
}
```

FIGURE 3

14:30 17:22

Fonction affiche améliorée.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en Java
Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

31. PUISSANCE 4 : MÉTHODE JOUE (PREMIÈRE VERSION)

Intéressons-nous maintenant au fait même de jouer. Pour cela nous allons devoir :
1. demander à un joueur où il joue ;
2. valider son coup ;
3. demander à l'autre joueur, c'est-à-dire alterner les joueurs ;
4. vérifier si la partie est finie : si un joueur a gagné ou si le jeu est plein.

Dans cette leçon nous allons implémenter le point 2, la validation du coup d'un joueur.

PREMIÈRE VERSION DE LA MÉTHODE

Pour cela nous allons écrire une méthode, et la première étape est de se demander ce dont a besoin cette méthode, et ce qu'elle retourne. Cette méthode a besoin de ***la grille à remplir***, du ***numéro de colonne*** où le pion doit être placé, et de ***la couleur*** du pion. Elle ne retourne rien pour l'instant.

L'entête de la méthode est donc la suivante (voir leçon 25 pour plus de détails) :

```
static void joue(int[][] grille, int colonne, int couleur)
```

Il faut maintenant écrire le corps de la méthode. Pour cela, on va devoir parcourir la colonne en partant du bas, jusqu'à trouver une case vide, où l'on placera le pion. Lorsque l'on utilise « *jusqu'à* » ou « *tant que* » pour décrire un programme, on sait que l'on va devoir utiliser une boucle conditionnelle.

La fonction complète est donnée dans la figure 1. On utilise un compteur (`ligne`) pour trouver la case vide, puis on ajoute le pion à la position voulue. Cependant, cette version est incomplète : en effet, que se passe-t-il si le joueur ajoute un pion dans une colonne pleine ? Notre programme actuel produirait une erreur d'exécution.

```
static void joue(int[][] grille, int colonne, int couleur)
{
  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide :
  int ligne = grille.length - 1;

  while (grille[ligne][colonne] != VIDE) {
    --ligne;
  }

  // on remplit la case vide trouvée :
  grille[ligne][colonne] = couleur;
}

...

joue(grille, 3, ROUGE);
joue(grille, 2, JAUNE);
joue(grille, 3, ROUGE);
```

FIGURE 1

6:06 16:43

Fonction `joue`, première version.

CORRECTION DE LA MÉTHODE

Pour prendre en compte le cas où la colonne est pleine, on pourrait rajouter un booléen, qui vaudrait `true` quand la colonne est pleine. On ne rajoute alors le pion que si la colonne n'est pas pleine.

Puisqu'il est maintenant possible que le pion ne soit pas ajouté, il faut que la méthode puisse dire si elle a pu ajouter le pion ou non. Pour cela, celle-ci doit retourner un booléen. Il faut donc changer l'entête de la méthode, et rajouter une instruction `return`.

La fonction complète est donnée dans la figure 2.

```
static boolean joue(int[][] grille, int colonne, int couleur)
{
  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide,
  // ou jusqu'en haut de la colonne si la colonne est pleine :
  int ligne = grille.length - 1;

  boolean pleine = false;
  while ((!pleine) && (grille[ligne][colonne] != VIDE)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jusqu'en haut de la colonne, on remplit la case vide trouvée,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (!pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
```

FIGURE 2

12:26 16:43

Fonction `joue`, après correction.

32. PUISSANCE 4 : MÉTHODE JOUE (RÉVISION)

Dans cette leçon nous allons voir deux alternatives à la méthode `joue` que nous avons faite dans la leçon précédente.

PREMIÈRE ALTERNATIVE

La méthode `joue` doit vérifier que la colonne n'est pas pleine pour pouvoir ajouter un pion. Pour cela, une manière alternative à la solution précédente serait de vérifier cette condition au début de la méthode. Il faut ainsi vérifier que la case à la ligne 0 et à la colonne donnée est vide. La suite du code est similaire à la première version de la méthode (voir leçon 31), si ce n'est qu'il faut penser à rajouter l'instruction `return`. La fonction complète est donnée dans la figure 1.

```
static boolean joue(int[][] grille, int colonne, int couleur)
{
  // si la colonne est pleine, le coup n'est pas valide :
  if (grille[0][colonne] != VIDE) {
    return false;
  }

  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide :
  int ligne = grille.length - 1;
  while (grille[ligne][colonne] != VIDE) {
    --ligne;
  }

  // on remplit la case vide trouvée :
  grille[ligne][colonne] = couleur;
  return true;
}
```

FIGURE 1

3:36 6:36

Fonction `joue`, première alternative.

SECONDE ALTERNATIVE

Une alternative serait de supprimer le booléen de la version originale, et de le remplacer par un test : le compteur (`ligne`) est-il supérieur ou égal à 0 ? Ainsi, si la colonne est pleine, la boucle s'arrête quand le compteur arrive en haut de la colonne, et on n'ajoute pas le pion. La fonction complète est donnée dans la figure 2.

```
static boolean joue(int[][] grille, int colonne, int couleur)
{
  int ligne = grille.length - 1;

  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide.
  //
  // Si le test (ligne >= 0) devient faux, c'est qu'on a
  // soustrait 1 à ligne quand elle valait 0, ce qui arrive quand la
  // colonne est pleine.
  while ((ligne >= 0) && (grille[ligne][colonne] != VIDE)) {
    --ligne;
  }

  // si ligne >= 0, on a trouvé une case vide, on la remplit,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (ligne >= 0) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
```

FIGURE 2

5:35 6:36

Fonction joue, seconde alternative.

Enfin, il manque une dernière vérification à notre méthode, dans toutes les versions que nous avons faites. En effet, notre programme ne gère pas le cas où le numéro de colonne est invalide. Le code final de la méthode joue est donné dans la leçon 33.

33. PUISSANCE 4 : MOTEUR DE JEU

Dans cette leçon nous allons continuer le moteur de jeu. Rappelons le déroulement d'une partie :
1. demander à un joueur où il joue ;
2. valider son coup ;
3. demander à l'autre joueur, c'est-à-dire alterner les joueurs ;
4. vérifier si l'un gagne ou si le jeu est plein.

Dans un premier temps nous écrirons tout le code dans la méthode `main`, puis nous le modulariserons à l'aide de méthodes.

MÉTHODE `main`

Décomposons la méthode en deux étapes. Il y a l'initialisation du jeu, et la boucle principale.

L'initialisation du jeu est assez simple. Il faut créer la grille, l'initialiser et l'afficher. Il faut aussi initialiser la variable qui déterminera le joueur dont c'est le tour. Le code de cette partie est donné dans la figure 1.

```
public static void main(String[] args)
{
  int[][] grille = new int[6][7];

  initialise(grille);
  affiche(grille);

  int couleurJoueur = JAUNE;
```

FIGURE 1

1:50 13:57

Méthode `main`, première partie.

La boucle principale du jeu est un peu plus longue. Il faut :
– demander au joueur d'entrer un numéro ;
– récupérer ce numéro ;
– jouer à l'aide de la méthode joue faite dans la leçon précédente ;
– vérifier que le coup est valide ;
– afficher la grille ;
– changer la couleur du joueur actuel.

Le code n'est cependant pas très compliqué. Celui-ci est donné dans la figure 2. Il manque la condition pour continuer la boucle principale, celle-ci sera traitée spécifiquement dans la leçon 34.

```
  do {
    if (couleurJoueur == JAUNE) {
      System.out.println("Joueur X :  entrez un numéro de colonne");
    } else {
      System.out.println("Joueur O :  entrez un numéro de colonne");
    }

    int colonne = clavier.nextInt();
    // les indices des tableaux commencent par 0 en Java:
    --colonne;

    boolean valide = joue(grille, colonne, couleurJoueur);
    if (!valide) {
      System.out.println(" > Ce coup n'est pas valide");
    }

    affiche(grille);
    // on change la couleur pour la couleur de l'autre joueur:
    if (couleurJoueur == JAUNE) {
      couleurJoueur = ROUGE;
    } else {
      couleurJoueur = JAUNE;
    }
  } while(    );
}
```

FIGURE 2

6:21 13:57

Méthode `main`, seconde partie.

MODULARISATION

Plusieurs parties peuvent être écrites à l'aide de méthodes dans le code précédent. On pourrait par exemple écrire une méthode pour demander un numéro au joueur, une autre pour changer la couleur du joueur. Nous allons écrire une méthode `demandeEtJoue` qui s'occupera de demander un numéro au joueur et de jouer le coup correspondant. Le code de cette méthode est donné dans la figure 3. On a simplifié l'affichage demandant au joueur un numéro, en regroupant les parties communes. De plus, on a rajouté une boucle conditionnelle pour gérer le cas où le coup n'a pas pu être joué : il faut alors demander un autre numéro.

```
void demandeEtJoue(int[][] grille, int couleurJoueur)
{
  boolean valide;

  do {
    System.out.print("Joueur ");
    if (couleurJoueur == JAUNE) {
      System.out.print("X");
    } else {
      System.out.print("O");
    }
    System.out.println(" :  entrez un numéro de colonne");

    int colonne = clavier.nextInt();
    // les indices des tableaux commencent par 0 en Java:
    --colonne;

    valide = joue(grille, colonne, couleurJoueur);
    if (!valide) {
      System.out.println(" > Ce coup n'est pas valide");
    }
  } while(!valide);
}
```

FIGURE 3

12:15 13:57

Méthode `demandeEtJoue`.

Enfin, le code final de la méthode joue écrite précédemment est donné dans la figure 4. Ce code gère le cas où le numéro de colonne rentré est invalide (trop grand).

```
static boolean joue(int[][] grille, int colonne, int couleur)
{
  // Si le numéro de colonne n'est pas valide, le coup n'est pas valide :
  if (colonne >= grille[0].length) {
    return false;
  }

  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide,
  // ou jusqu'en haut de la colonne si la colonne est pleine :
  int ligne = grille.length - 1;

  boolean pleine = false;
  while ((!pleine) && (grille[ligne][colonne] != VIDE)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jusqu'en haut de la colonne, on remplit la case vide trouvée,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (!pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
```

FIGURE 4

13:43 13:57

Méthode `joue`, version finale.

34. PUISSANCE 4 : CONDITION FIN DU JEU

Dans la leçon précédente, nous avons écrit la boucle principale du jeu. Il manquait cependant la condition d'arrêt de la boucle, et c'est ce que nous allons faire maintenant. Pour cela nous allons écrire une méthode `estCeGagne`, qui prendra en paramètre la grille et la couleur du joueur à vérifier. On utilisera un booléen pour la boucle principale, comme décrit dans la figure 1.

```
int couleurJoueur = JAUNE;
boolean gagne;

do {
  demandeEtJoue(grille, couleurJoueur);

  affiche(grille);

  gagne = estCeGagne(grille, couleurJoueur);

  // on change la couleur pour la couleur de l'autre joueur:
  if (couleurJoueur == JAUNE) {
    couleurJoueur = ROUGE;
  } else {
    couleurJoueur = JAUNE;
  }
} while(!gagne);
```

FIGURE 1

1:10 15:34

Méthode main, avec condition d'arrêt.

MÉTHODE `estCeGagne`

Selon la description faite précédemment, cette méthode aura l'entête suivante (voir leon 25 pour plus de détails) :

```
static boolean estCeGagne(int[][] grille, int couleurJoueur)
```

Il existe plusieurs stratégies pour implémenter cette méthode. Voici celle que nous avons choisie :
– On parcourt toutes les cases de la grille.
– On teste si cette case est de la couleur du joueur concerné.
– Si oui, alors on parcourt la grille dans toutes les directions à partir de cette case et on compte le nombre de pion à la suite.
– Si l'on compte 4 pions ou plus à un moment donné, alors le joueur a gagné.

On remarque qu'il n'est pas nécessaire de parcourir les 8 directions possibles : en effet, pour chaque ligne, on peut la parcourir dans un sens ou dans l'autre. Ainsi, 4 directions suffisent.

On suit donc notre stratégie pour écrire le corps de la méthode. Tout d'abord, on parcourt les cases de la grille à l'aide de deux boucles for. Ensuite, on teste si la couleur de la case est la couleur passée en argument `(if)`. Vient ensuite la partie un peu plus complexe, celle de compter les cases dans toutes les directions.

Pour compter les cases dans une direction donnée, on introduit une méthode `compte`. Cette méthode prend théoriquement en paramètre la grille, la case et la direction choisie, et retourne un entier. En réalité, nous n'avons pas de type case ou direction. Nous passerons donc la case et la direction par leurs coordonnées (deux entiers, x et y). La coordonnée d'une direction représente le déplacement d'une case à la suivante dans cette direction. Par exemple, la direction vers la droite aura comme coordonnée (0, +1), celle vers le bas (+1,0), etc.

Il suffit donc d'appeler la méthode `compte` pour les 4 directions qui nous intéressent, et si l'un des appels retourne 4 ou plus, alors le joueur a gagné, on retourne `true`. Enfin, à la fin de la méthode, on retourne `false` puisque si le programme arrive là c'est que le joueur n'a pas gagné. La fonction complète est donnée dans la figure 2.

```
// gagne = estCeGagne(grille, couleurJoueur);
static boolean estCeGagne(int[][] grille, int couleurJoueur)
{
  for(int ligne = 0; ligne < grille.length; ++ligne) {
    for(int colonne = 0; colonne < grille[ligne].length; ++colonne) {
      int couleurCase = grille[ligne][colonne];

      if (couleurCase == couleurJoueur) {
        if (
          // en diagonale, vers le haut et la droite:
          (compte(grille, ligne, colonne, -1, +1) >= 4) ||
          // horizontalement, vers la droite:
          (compte(grille, ligne, colonne, 0, +1) >= 4) ||
          // en diagonale, vers le bas et la droite:
          (compte(grille, ligne, colonne, +1, +1) >= 4) ||
          // verticalement, vers le bas:
          (compte(grille, ligne, colonne, +1, 0) >= 4)
        ) {
          return true;
        }
      }
    }
  }

  return false;
}
```

FIGURE 2

9:00 15:34

Fonction estCeGagne.

MÉTHODE `compte`

Selon la description faite précédemment, cette méthode aura l'entête suivante (voir leçon 25 pour plus de détails):

```
static int compte(int[][] grille, int ligneDepart, int colonneDepart,
                  int dirLigne, int dirColonne)
```

Pour cette méthode nous devons itérer dans notre direction tant que la case actuelle est identique à la case de départ. Ainsi nous allons faire une boucle conditionnelle, et nous avons besoin de trois variables: un compteur et deux coordonnées `ligne` et `colonne` représentant la case à une itération donnée.

À chaque itération, on incrémente le compteur et on ajoute à `ligne` et `colonne` respectivement `dirLigne` et `dirColonne`. Enfin, il faut vérifier que nous ne sortons pas de la grille dans notre boucle, pour cela nous rajoutons 4 conditions, une pour chaque borne pour les variables `ligne` et `colonne`. La fonction complète est donnée dans la figure 3.

```java
static int compte(int[][] grille,
                  int ligneDepart, int colonneDepart,
                  int dirLigne, int dirColonne)
{
  int compteur = 0;

  int ligne = ligneDepart;
  int colonne = colonneDepart;

  // on part de la case (ligneDepart, colonneDepart) et on parcourt la grille
  // dans la direction donnee par (dirLigne, dirColonne)
  // tant qu'on trouve des pions de la meme couleur que le pion de depart.
  while (grille[ligne][colonne] == grille[ligneDepart][colonneDepart] &&
         ligne   >= 0 && ligne   < grille.length &&
         colonne >= 0 && colonne < grille[ligne].length) {

    ++compteur;
    ligne   = ligne   + dirLigne;
    colonne = colonne + dirColonne;

  }

  return compteur;
}
```

FIGURE 3

12:50 15:34

Fonction `compte`.

OPTIMISATION MÉTHODE `estCeGagne`

On pourrait améliorer les performances de notre programme en ne gardant que les directions utiles pour chaque case. Par exemple, pour les cases situées à moins de 4 cases du bord droit, il est inutile d'itérer vers la droite, de même pour les cases situées à mois de 4 cases du bas de la grille, il est inutile d'itérer vers le bas. Il faudrait donc modifier nos conditions dans la méthode `estCeGagne`, en ajoutant une condition supplémentaire pour chaque condition. Le code final de cette méthode est donné dans la figure 4.

```java
if (couleurCase == couleurJoueur) {
  final int ligneMax = grille.length - 4;
  final int colonneMax = grille[ligne].length - 4;

  if (
    // en diagonale, vers le haut et la droite:
    (ligne >= 3 && compte(grille, ligne, colonne, -1, +1) >= 4) ||

    // horizontalement, vers la droite:
    (colonne <= colonneMax && compte(grille, ligne, colonne, 0, +1) >= 4) ||

    // en diagonale, vers le bas et la droite:
    (ligne <= ligneMax && colonne <= colonneMax &&
     compte(grille, ligne, colonne, +1, +1) >= 4) ||

    // verticalement, vers le bas:
    (ligne <= ligneMax && compte(grille, ligne, colonne, +1, 0) >= 4)
  ) {
    return true;
  }
}
```

FIGURE 4

15:16 15:34

Fonction `estCeGagne`, version finale.

35. PUISSANCE 4 : FINALISATION

Il ne manque plus que deux choses à notre programme pour être complet : afficher un message lorsqu'un joueur gagne et gérer le cas où la grille est pleine.

FINALISATION MÉTHODE `main`

L'affichage du vainqueur est assez simple. Il suffit d'afficher un message relatif au joueur actuel. Attention toutefois, étant donné que l'on inverse le joueur actuel à la fin de la boucle, la variable `couleurJoueur` contient le perdant à la fin. Il faut donc inverser les messages. Le code de cette partie est donné dans la figure 1. Il faut le rajouter après la boucle principale, dans la méthode main.

```
// attention, on a change la couleur pour la couleur de l'autre joueur !
if (couleurJoueur == JAUNE) {
  System.out.println("Le joueur O a gagne !");
} else {
  System.out.println("Le joueur X a gagne !");
}
```

FIGURE 1

2:48 10:05

Affichage du vainqueur.

Il faut maintenant gérer le cas où la grille est pleine. Pour cela, commençons par rajouter une condition à notre boucle principale. On introduit pour cela une méthode `plein`, qui prend en argument la grille et retourne un booléen. Il faut aussi penser à gérer ce cas dans l'affichage du vainqueur. Le code final de la fin de la méthode `main` est donné dans la figure 2.

```
    // on change la couleur pour la couleur de l'autre joueur:
    if (couleurJoueur == JAUNE) {
      couleurJoueur = ROUGE;
    } else {
      couleurJoueur = JAUNE;
    }
  } while(!gagne && !plein(grille));

  if (gagne) {
    // attention, on a change la couleur pour la couleur de l'autre joueur !
    if (couleurJoueur == JAUNE) {
      System.out.println("Le joueur O a gagne !");
    } else {
      System.out.println("Le joueur X a gagne !");
    }
  } else {
    System.out.println("Match nul !");
  }
}
```

FIGURE 2

5:08 10:05

Méthode `main`, version finale.

MÉTHODE `plein`

Selon la description faite précédemment, la méthode vérifiant si la grille est pleine aura l'entête suivante (voir leon 25 pour plus de détails):

```
static boolean plein(int[][] grille)
```

Pour son fonctionnement, on pourrait itérer sur toutes les cases de la grille et tester si elles sont vides ou non. Cependant, étant donné que les pions s'empilent vers le haut, on remarque qu'il suffit d'itérer sur les cases de la ligne la plus haute et vérifier si elles sont vides ou non. La fonction complète est donnée dans la figure 3.

```java
// plein(grille)
static boolean plein(int[][] grille)
{
  // Si on trouve une case vide sur la premiere ligne, la grille n'est pas pleine :
  for(int cellule : grille[0]) {
    if (cellule == VIDE) {
      return false;
    }
  }

  // Sinon, la grille est pleine :
  return true;
}
```

FIGURE 3

8:00 10:05

Fonction plein.

CONCLUSION DU PROJET

Ceci conclut notre étude de cas. Le but était d'illustrer les règles à suivre lorsque nous devons développer un programme plus ambitieux, données dans l'introduction (leçon 29):

- Ne pas écrire tout le projet d'un seul coup: il faut ***décomposer le projet*** en sous-parties, et développer le programme par étapes. A chaque étape, ***tester le code développé***.
- Identifier dans l'ordre: les types nécessaires au programme, puis les méthodes qui portent sur ces types.
- Lorsqu'un code se répète ou répond à un besoin précis, le ***modulariser*** à l'aide d'une méthode, et décomposer les méthodes complexes en utilisant d'autres méthodes.

Le code complet de l'étude de cas peut être téléchargé **ici**.

IMPRESSUM

Graphisme :
Emphase Sàrl, Lausanne

Résumé : Louis Basseto
et Jean-Baptiste Beau

Développés par EPFL Press, les BOOCs (Book and Open Online Courses) sont le support compagnon des MOOCs proposés par l'École polytechnique fédérale de Lausanne. Valeur ajoutée aux MOOCs, ils rassemblent l'essentiel à retenir pour l'obtention du certificat et constituent un atout pédagogique. Learn faster, learn better. Bonne révision !

ISBN 978-2-88914-397-9