

Bienvenue !

Pierre-Julien VILLOUD



- pjvilloud@protonmail.com
- <https://github.com/pjvilloud>
- <https://linkedin.com/pjvilloud>
- <http://pjvilloud.github.io>

Java, une plateforme de programmation

Issu du [Livre blanc](#) :

Simplicité, Orienté Objet, et Familié

Robustesse et Sécurité

Architecture neutre et Portabilité

Interpréte, Multithread, et Dynamique

Performances élevées

Simplicité, Orienté Objet, et Familié

Syntaxe proche de C++

Petite taille

Fonctionnalités orientées objet avancées

Suppression des complexités de C++

Robustesse et Sécurité

Early checking

Run-time checking

Allocation mémoire gérée par la JVM

Conçu pour rendre impossible certaines attaques

Architecture neutre et Portabilité

Bytecode

S'exécute partout où une JVM est présente

Taille et types de données spécifiés

Interprétré, Multithread, et Dynamique

Exécution de bytecode

Chargement de code depuis Internet

Reflection

Performances élevées

Interprétation de bytecode plus lente...

Compilateurs JIT

Exemple de programme java

Mot-clé 'public'

Ceci est un spécificateur d'accès qui indique la visibilité de la classe par les autres parties du programme.

Mot-clé 'class'

Tout (ou presque...) ce qui est programmé en java se trouve à l'intérieur d'une classe.

public class HelloWorld

{

public static void main(String[] args)

{

System.out.println("Hello World");

}

}

Accolades

Les accolades délimitent le début et la fin de la classe, ou de la méthode, ou d'une instruction.

Nom de la classe

Le nom donné à votre classe, mais également le nom de votre fichier, `HelloWorld.java`

Méthode 'main'

Le point d'entrée de la classe. Tout programme java doit avoir un point d'entrée 'main' dont l'entête est identique à ce qui est présenté ici.

Instruction

Le corps de la méthode contient une ou plusieurs instructions. Une instruction finit toujours par un ';' mais peut être écrite sur plusieurs lignes. Ici cette instruction affiche le texte 'Hello World' dans la console et effectue un retour à la ligne.

Exécuter un programme



Le bytecode généré est exécutable par n'importe quelle machine virtuelle.

Règles d'écriture

💡 Ecrire un programme java demande à suivre certaines **règles et conventions**

Nom de fichier : Le nom de fichier doit s'appeler comme la classe qu'il contient suivi de l'extension `.java`

Identificateurs : Un identificateur est composé de lettres et de chiffres et est sensible à la casse

Séparateurs : Deux identificateurs doivent être séparés par au moins un espace ou une fin de ligne.

Format libre : La mise en page du programme est totalement libre.

Indentation : Il est recommandé d'indenter son programme pour le rendre lisible.

Caractères accentués et spéciaux : Il est recommandé de pas utiliser de caractères spéciaux ou accentués dans les identificateurs

Nom du fichier

Classe `HelloWorld` → fichier `HelloWorld.java` . Pas `helloworld.java`, pas `Helloworld.java`.

Identificateurs

Les majuscules A-Z, les minuscules a-z, les chiffres 0-9 ainsi que le caractère _ sont autorisés. Les identificateurs sont sensibles à la casse. `monentier` est différent de `monEntier` . Les mots réservés java ne peuvent être utilisés comme indicateurs.

Séparateurs

Les espaces, tabulations, retours à la ligne sont considérés comme des séparateurs et il est possible d'en enchaîner plusieurs à la suite.

Format libre

`public class HelloWorld{public static void main(String[] args){System.out.println("Hello World");}}` est une classe valide...

Indentation

L'indentation est souvent réalisée automatiquement lorsqu'on utilise un IDE (comme Eclipse par exemple). Se référer à un [guide de style](#) pour en savoir plus.

Caractères spéciaux

Les identificateurs `tést prix$ réel` sont autorisés mais pas `adresseM@il` . Dans le doute, se contenter des caractères alpha-numériques non accentués pour éviter aussi les problèmes d'encodage.

Bases du langage

Commentaires

Types primitifs de données

Types énumérés

Variables

Constantes

Méthodes

Opérateurs

Conversions et transtypages

Tableaux

Commentaires

```
/*
 * Première classe d'exemple
 * pour le cours Java 210
 */
public class HelloWorld
{
    /**
     * Point d'entrée du programme
     *
     * @param args Tableau des arguments passés au programme
    */
    public static void main(String[] args)
    {
        //Affiche Hello World dans la console
        System.out.println("Hello World");
    }
}
```

Commentaire sur plusieurs lignes

Les commentaires n'apparaissent pas dans le programme exécutable. Ils n'ont donc pas d'influence sur la taille ou les performances du programme. Les commentaires /* */ ne sont pas imbriquables. Pour mettre en commentaire un bloc de code, le mettre entre /* */ est insuffisant s'il contient lui même un commentaire /* */

Javadoc

Commentaires utilisés pour la génération de la documentation

Commentaire sur une ligne

Commentaire pour donner des informations sur la ou les instructions suivantes.

Types primitifs de données

i Java est un langage **fortement typé**. Toute variable doit donc avoir un type.

Types primitifs prédefinis en java

Entiers : `int short long bytes`

Réels : `float double`

Caractère : `char`

Vrai/faux : `boolean`

Entiers

❶ Les types primitifs entiers représentent les nombres sans partie décimale, positifs ou négatifs.

Type	Occupation en mémoire	Intervalle (limites incluses)
byte	1 octet	-128 à 127
short	2 octets	-37 768 à 32767
int	4 octets	-2 147 489 648 à 2 147 483 647
long	8 octets	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807

Réels

❶ Les types primitifs à virgule flottante expriment les nombres réels avec une partie décimale, positifs ou négatifs.

Type	Occupation en mémoire	Intervalle (limites incluses)	
float	4 octets	$\pm 3.40282347 \times 10^{38}$	6 ou 7 décimales significatives
double	8 octets	$\pm 1.79769313486231570 \times 10^{308}$	15 décimales significatives

❶ Le type `double` (pour double précision) sera souvent préféré au type `float` (plus rapide mais moins précis).

Caractère

i Le type `char` permet de représenter une *unité de code* en codage UTF-16 (sur 2 octets).

A Deux octets ne suffisant pas à représenter l'ensemble des caractères de toutes les langues, il faut donc parfois utiliser deux *unités de code* UTF-16 pour représenter un *point de code* (= caractère). Ex : U+D835 et U+DD6B sont *deux unités de code* UTF-16 pour afficher le *point de code* U+1D56B représentant le caractère Unicode ☞

A Il est donc déconseillé d'utiliser le type `char` dans les programmes. Préfere le type `String`.

Booléen

i Le type `boolean` peut avoir deux valeurs, `true` (vrai) ou `false` (faux). Il est utilisé pour évaluer des conditions logiques.

Types énumérés

❶ Lorsqu'une variable ne peut contenir qu'un ensemble limité de données, il est possible de définir un type énuméré. Ce type possède un nombre fini de valeurs nommées.

```
//Définition du type énuméré Taille
enum Taille {SMALL, MEDIUM, LARGE, EXTRA_LARGE};
//L'ordre de définition des constantes SMALL, MEDIUM... est important !
```

Variables

i Une variable est un contenant nommé permettant de stocker une valeur d'un type, ou une référence à un objet. On la déclare en spécifiant son type, puis son nom. Toute variable doit être initialisée avant d'être utilisée. Sinon une erreur de compilation sera signalée par le compilateur.

```
boolean enConge; //Déclaration de la variable enConge  
enConge = false; //Initialisation de la variable en affectant la valeur false  
int joursVacances = 25; //Déclaration et initialisation de la variable joursVacances en une ligne.  
double salaire, prime; //Déclaration de deux variables en une ligne. Non recommandé.  
System.out.println(salaire); //Erreur de compilation  
Taille s = Taille.SMALL; //Déclaration d'une variable du type énuméré Taille
```

i Il est fortement recommandé de donner un nom pertinent à chaque variable et d'éviter les noms à rallonge

```
int a, var1, monEntier; // Déconseillé car non pertinent  
double Salaire, primeFinAnnée; //Déconseillé car majuscule et accents  
short mon Age, m@x; //Erreur car espaces ou caractères non alpha-numériques  
boolean estPresent; //OK  
int age, Age, AGE; //Trois variables différentes car sensible à la casse.  
boolean public; //Interdit car public est un mot réservé de java
```

Constantes

❶ Une constante ne peut plus changer de valeur après son initialisation. On la déclare avec le mot `final`. Par convention, les noms des constantes sont en majuscules et les mots sont séparés par des _

```
public class ExempleConstantes {  
    //Visible dans toute la classe  
    public static final short NOMBRE_MOIS_ANNEE = 12;  
    public static final short NOMBRE_TRIMESTRE_ANNEE = NOMBRE_MOIS_ANNEE / 3;  
    public static void main(String[] args){  
        final short NOMBRE_JOURS_SEMAINE = 7; //Visible uniquement dans main  
        System.out.println("Nombre de mois dans l'année : " + NOMBRE_MOIS_ANNEE);  
        NOMBRE_JOURS_SEMAINE = 9; //Interdit ! Erreur de compilation  
    }  
}
```

❶ L'utilisation du terme `static` permet de définir une constante de classe qui sera accessible en dehors de la classe :

`ExempleConstantes.NOMBRE_MOIS_ANNEE`

Méthodes

❶ Une méthode est un bloc de code qui effectue un traitement. On peut comparer une méthode à une fonction mathématique.

Type de la valeur de retour	Nom de la méthode	Visibilité
Le type de retour de la méthode doit être précisé. Une méthode ne peut renvoyer qu'un seul type (ou aucun en utilisant le mot-clé <code>void</code>)	C'est ce nom qui sera utilisé pour appeler la méthode <code>int auCarre(int x)</code>	Mot-clé facultatif précisant la visibilité de la méthode parmi <code>protected</code> , <code>private</code> , <code>public</code> . Lorsque rien n'est mentionné, la méthode est visible par tous le package
Corps de la méthode	Arguments	Pas d'arguments
Le corps de la méthode contient une ou plusieurs instructions qui représentent le traitement que doit effectuer cette méthode. Elle utilise les éventuels arguments en entrée pour éventuellement produire une valeur en sortie (mot clé <code>return</code>).	Les arguments sont les valeurs d'entrées fournies à la méthode pour lui permettre d'effectuer son traitement. Lorsqu'il y en a plusieurs, on les sépare par une virgule. <code>//Multiplie l'argument x par //lui-même et retourne du résultat</code> <code>return x * x;</code>	Lorsqu'il n'y a pas d'arguments, on se contente de mettre des parenthèses vides
		<code>public void afficheBonjour()</code> <code>{</code> <code> //Cette méthode ne fait qu'afficher</code> <code> //Pas de return</code> <code> System.out.println("Bonjour !");</code> <code>}</code>
		<code>afficheBonjour(); //Affiche "Bonjour !"</code>

❶ La notion de visibilité sera abordée en détail dans le cours Java 220.

Opérateurs arithmétiques

❶ Les opérateurs arithmétiques + - * / % permettent d'effectuer les opérations de bases que sont l'addition, la soustraction, la multiplication et la division

```
int quinze = 3 * 5; int deux = quinze - 13;  
int sept = quinze / deux; //Division entière car les variables sont des entiers  
int un = 15 % 2; //Reste de la division entière de 15 par 2 (modulo)  
double quinzeDouble = 15.0;  
double septVirguleCinq = quinzeDouble / 2; //Division entre un double et un entier  
int x = 5; x++; //équivalent à x = x + 1; x vaut 6  
int y = 2 * --x; //y vaut 10, x vaut 5  
int z = 2 * x++; //z vaut 10, x vaut 6  
x -= 2; //équivalent à x = x - 2; x vaut 4
```

⚠ L'utilisation de ++ ou de -- doit être effectuée avec précaution car cela peut rendre le code confus.

Opérateurs relationnels et booléens

i Les opérateurs relationnels et booléens permettent de tester des conditions

Test	Signe	Exemple
Egalité	<code>==</code>	<code>4 == 5 // false</code>
Différence	<code>!=</code>	<code>4 != 5 // true</code>
"ET" logique	<code>&&</code>	<code>true && false // false.</code>
"OU" logique	<code> </code>	<code>true false // true</code>
"NON" logique	<code>!</code>	<code>!true // false.</code>
Infériorité	<code>< et <=</code>	<code>5 < 5 // true</code> <code>5 <= 5 // true</code>
Supériorité	<code>> et >=</code>	<code>4 > 5 // false</code> <code>5 >= 5 // true</code>
Opérateur ternaire	? et :	<code>x < y ? x : y // Renvoie x si x < y, sinon renvoie y</code>

La deuxième expression n'est pas calculée si la première a pu être évaluée à `true` (puisque le résultat serait `true` de toute façon. Ainsi `x == 0 || 1 / x > 0` ne lance jamais d'erreurs car la division par 0 est évitée avec le premier test. L'inversion du test peut cependant lever une exception.

Opérateurs binaires

❶ Les opérateurs binaires permettent de travailler directement sur les bits des types entiers.

Opération	Signe	Exemple
"ET" logique	&	<code>10 & 3 // 1010 & 0011 => 0010 soit 2</code>
"OU" logique		<code>10 3 // 1010 0011 => 1011 soit 11</code>
"OU" exclusif	^	<code>10 ^ 3 // 1010 ^ 0011 => 1001 soit 9</code>
"NON" logique	~	<code>~10 // ~1010 => 0101 soit 5</code>
Décalage de bits	<< ou >>	<code>10 >> 1 // 1010 >> 1 => 0101 soit 5 (bit de poids faible perdu)</code> <code>10 << 1 // 1010 >> 1 => 0100 soit 4 (bit de poids fort perdu)</code>

⚠ Les opérateurs `&` et `|` donnent une valeur `boolean` lorsqu'ils sont appliqués à des `boolean` mais évaluent les deux arguments de chaque côté de l'opérateur, contrairement à `&&` et `||`.

Priorités des opérateurs

i Les règles de priorités entre opérateurs arithmétiques sont les mêmes qu'en algèbre.

Expression	Équivalent
a + b * c ou a b && c	a + (b * c) ou a (b && c)
a * b + c * d ou a && b c && d	(a * b) + (c * d) ou (a && b) (c && d)
-c * d ou !c && d	(-c) * d ou (!c) && d
-a * -b + c ou !a && !b c	((-a) * (-b)) + c ou ((!a) && (!b)) c

i Il est recommandé de mettre les parenthèses pour les opérateurs relationnels et booléens même si elles ne sont pas nécessaires pour faciliter la lecture et éviter les erreurs (en cas de changement d'opérateur par exemple).

Conversions

⚠ Java peut effectuer des conversions automatiques entre types. Cela peut cependant mener à des pertes d'information.

Exemple de perte de précision :

```
int n = 987654321;  
float f = n; // f vaut 9.8765434E8
```

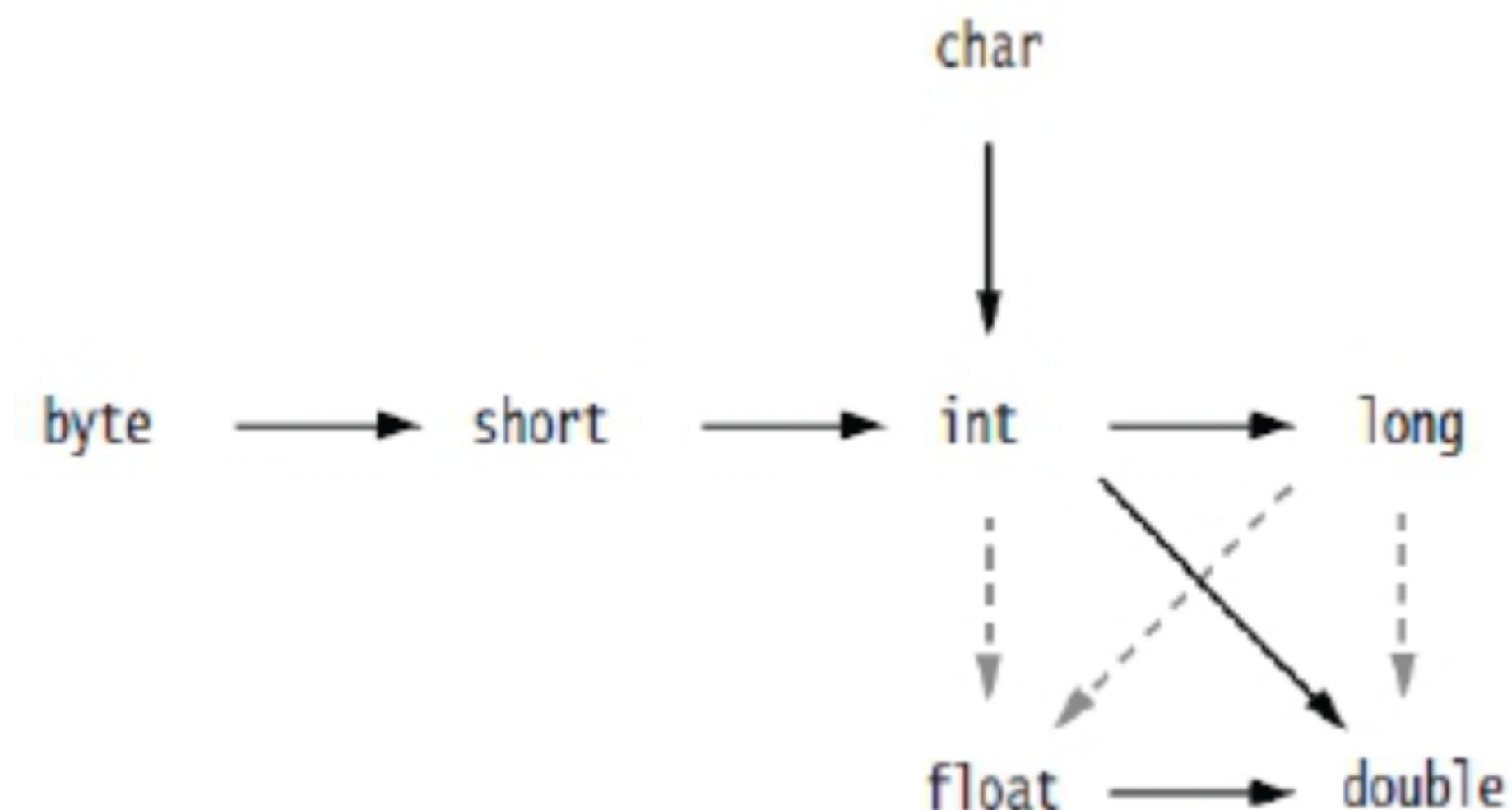
Règles de conversions automatiques lors d'opérations arithmétiques :

Si un des opérandes est un `double`, l'autre sera converti en `double`

Sinon, si un des opérandes est un `float`, l'autre sera converti en `float`

Sinon, si un des opérandes est de type `long`, l'autre sera converti en `long`

Sinon les deux opérandes seront convertis en `int`



⚠ Les flèches noires représentent les conversions sans perte de données, les flèches en pointillées celles avec perte de données.

Transtypages (cast)

💡 Il est également possible de forcer manuellement ces conversions lorsque les besoins du programme le nécessitent. Pour cela, on préfixe la variable à transtyper du type cible entre parenthèses :

```
short s = 56;  
int i = (int) s
```

Exemples de transtypage :

```
//Transtypage double -> int  
double d = 4.567;  
int n = (int)d;// d vaut 4 (la partie décimale est perdue)
```

```
//Transtypage int -> short  
int i = 55000;  
short s = (short)i; //s vaut -10536...
```

⚠ Le transtypage ne doit pas être utilisé pour arrondir un nombre à virgule flottante. Il faut plutôt utiliser `Math.round(x)`.

Tableaux

💡 Il est souvent utile de pouvoir gérer un ensemble de valeurs du même type. On utilise pour cela un **tableau** java. A noter que la notion de **tableau** n'est pas un type primitif, il faut donc utiliser l'instruction `new` pour l'initialiser.

```
//Déclaration d'un tableau d'entier
int[] tableauEntier, tableauEntier2; //Le tableau est déclaré mais non initialisé

//Initialisation du tableau
tableauEntier = new int[50]; //Le tableau est vide et peut contenir jusqu'à 50 valeurs
tableauEntier2 = new int[] {5, 4, 3, 8, 5}; //Déclare, initialise et remplit le tableau

//Insertion d'un élément dans le tableau
tableauEntier[0] = 5; //Met l'entier 5 dans le premier élément du tableau.
tableauEntier[50] = 10; //Exception, on essaye d'insérer au 51e élément du tableau.

//Accès à un élément dans un tableau
int premierElement = tableauEntier[0];

//Obtenir la taille du tableau
int tailleTableau = tableauEntier.length;
```

⚠️ Les indices d'accès au tableau commencent à 0 !
⚠️ Il est impossible de modifier la taille du tableau une fois créé.

Manipulation de tableaux

💡 Il est souvent nécessaire de manipuler les tableaux (copier, trier, rechercher). Java fournit des utilitaires pour effectuer ces opérations courantes.

```
//Copie de tableaux
//Attention, ceci n'est pas une copie !
tableauEntier = tableauEntier2; // tableauEntier2 et tableauEntier font maintenant référence au même tableau
//Pour copier on utilise Arrays.copyOf(tableauCible, nbElementsACopier) ou Arrays.copyOfRange(tableauCible, indexDepart, indexFin);
tableauEntier = Arrays.copyOf(tableauEntier2, tableauEntier2.length);
tableauEntier = Arrays.copyOfRange(tableauEntier2, 0, tableauEntier2.length);

//Tri d'un tableau
Arrays.sort(tableauEntier);

//Recherche dans un tableau trié
int i = Arrays.binarySearch(tableauEntier, 5); //Renvoie l'indice de la valeur 5 si elle est trouvée ou une valeur négative sinon

//Comparaison de tableau
boolean egal = Arrays.equals(tableauEntier, tableauEntier2); //Renvoie true si les tableaux ont la même longueur et le même contenu

//Remplissage d'un tableau
Arrays.fill(tableauEntier, 6); //Remplie tous les éléments du tableau avec la valeur 6
```

Tableaux multi-dimensionnels

❶ Les tableaux à deux dimensions sont souvent utilisés mais il est également possible de créer des tableaux à trois, quatre dimensions...

```
//Déclaration d'un tableau bidimensionnel
char[][] tableau2D;// [][] pour deux dimensions, [][][] pour trois dimensions...

//Initialisation d'un tableau bidimensionnel
tableau2D = new char[3][4];//Équivalent à un tableau de 3 colonnes et 4 lignes
char [][] sudoku = {
    {'2', '1', '9'},
    {'8', '4', '3'},
    {'6', '7', '?'}

}; // Il n'est pas nécessaire de définir les limites du tableau lorsqu'on connaît tous ses éléments

//Accès à un élément du tableau bidimensionnel
char i = sudoku[0][1];//i vaut '1'

//Affectation d'un élément du tableau bidimensionnel
sudoku[2][2] = '5';//La valeur '?' est remplacée par '5'
```

❶ Les tableaux multi-dimensionnels ne sont rien d'autre que des tableaux de tableaux (de tableaux...).

Flux d'exécution

i Le **flux d'exécution** du programme peut être contrôlé pour satisfaire les besoins du programme.

Portée d'un bloc

Conditions

Boucles

Interruptions

Portée d'un bloc

❶ La notion de **bloc** est importante dans la gestion du flux d'exécution. Les **blocs** permettent de regrouper des instructions, déterminent la portée des variables et peuvent être imbriqués les uns dans les autres. Ils sont délimités par des `{ }`.

```
public static void main(String[] args)
{
    int entierGlobalMethode = 5;
    {
        System.out.println(entierGlobalMethode); //entierGlobalMethode est accessible ici
        int entierLocalBloc = 4;
        {
            System.out.println(entierGlobalMethode); //entierGlobalMethode est accessible ici
        }
        int entierGlobalMethode = 5; //Exception : entierGlobalMethode est déjà défini
    }
    System.out.println(entierLocalBloc); //Exception : entierLocalBloc n'est plus accessible ici
    int entierLocalBloc = 4; //entierLocalBloc n'est plus accessible, il peut être de nouveau défini
}
```

Conditions

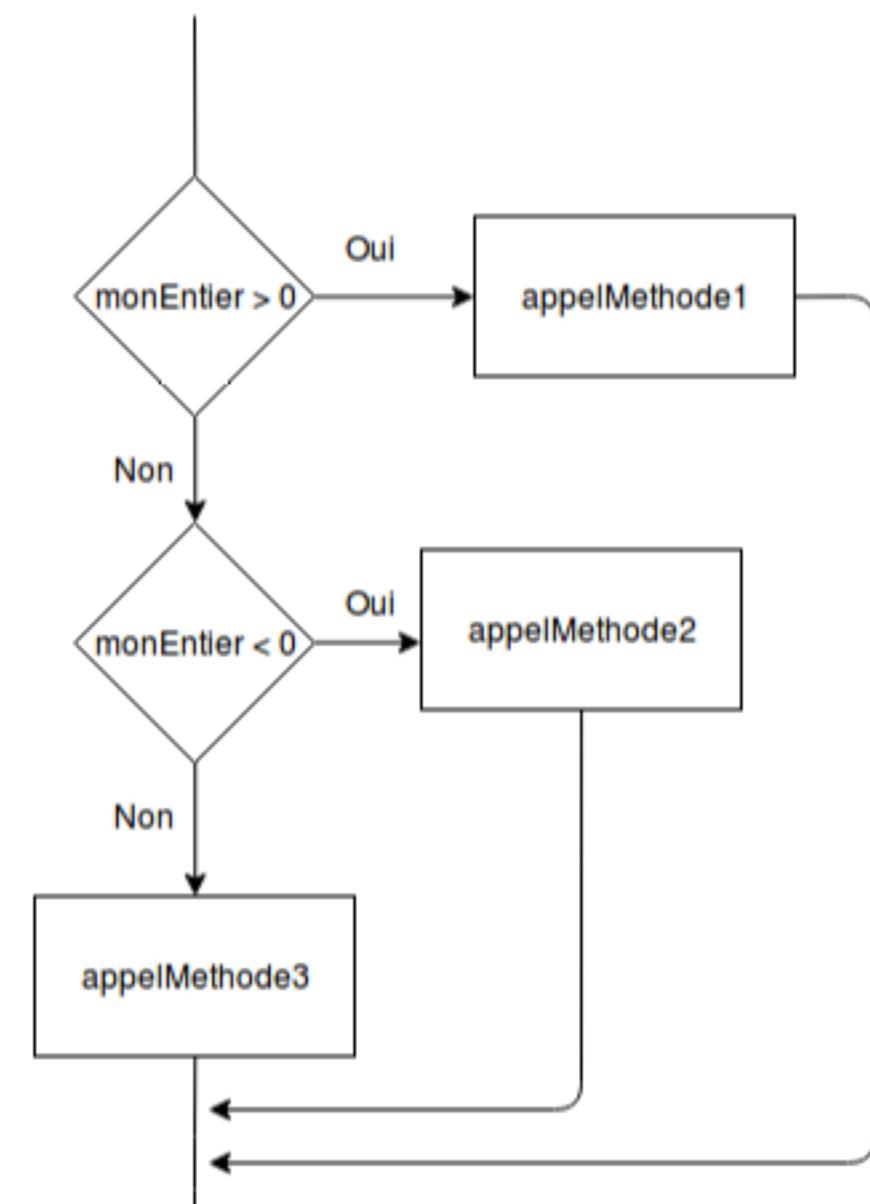
i Pour tester une condition, on utilise les mots réservés `if` et `else`.

```
boolean estStrictementPositif = false; int monEntier = 5;
if(monEntier > 0)
    estStrictementPositif = true;
    System.out.println(monEntier + " est supérieur ou inférieur à 0"); //Attention, exécuté systématiquement !

if(estStrictementPositif) // équivalent à if(estStrictementPositif == true)
{
    System.out.println(monEntier + " est strictement supérieur à 0");
    appelMethode1(monEntier); //Deux instructions, donc nécessité d'utiliser un bloc { }
}

else if (monEntier < 0) //Plusieurs else if peuvent être enchaînés pour tester plusieurs cas
{
    System.out.println(monEntier + " est strictement inférieur à 0");
    appelMethode2(monEntier);
}

else { //Si aucune des conditions précédentes n'est remplie
    System.out.println(monEntier + " est égal à 0");
    appelMethode3(monEntier);
}
```



⚠ Il est recommandé de systématiquement utiliser un bloc { } même lorsqu'il n'y a qu'une instruction, afin d'éviter des erreurs lors d'ajout d'instructions.

Conditions multiples

❶ Lorsqu'on teste plusieurs conditions, la notation `if else` peut être assez lourde. On peut alors utiliser l'instruction `switch` si les tests portent sur des types entiers (et depuis Java 7, les chaînes de caractère).

```
int position = 1;
switch (position) {
    case 1:
    case 2:
    case 3:
        System.out.println("Sur le podium, bien joué !");
        break;
    case 4:
        System.out.println("Quatrième, aux portes du podium !");
        break;
    case 5:
        System.out.print("Pas si mal, cinquième ou ");
    case 6:
        System.out.println("sixième... Bel effort !");
        break;
    default:
        System.out.println("Loin derrière, merci d'avoir participé !");
        break;
}
```

Lorsque `position` vaut 1, 2 ou 3, 'Sur le podium, bien joué !' est affiché

Lorsque `position` vaut 4, 'Quatrième, aux portes du podium !' est affiché et l'instruction `break` fait sortir du bloc `switch`

Lorsque `position` vaut 5, 'Pas si mal, cinquième ou sixième... Bel effort !' est affiché car il n'y a pas d'instruction `break` après la première instruction. Lorsque `position` vaut 6, seul 'sixième... Bel effort !' sera affiché et l'instruction `break` fait sortir du bloc `switch`

Lorsque `position` n'est pas compris entre 1 et 6, 'Loin derrière, merci d'avoir participé !' est affiché, c'est le comportement par défaut lorsqu'aucune condition précédemment évaluée n'est vraie

⚠ Attention, l'utilisation du `switch` peut être piégeuse en cas d'oubli de `break` dans chaque `case`.

Boucles **while**

❶ Pour effectuer un traitement tant qu'une condition n'est pas vérifiée, on peut utiliser une boucle **while**.

```
int[] tableauFibonacci = {0, 0, 1, 2, 3, 5, 8, 13, 21, 34, 55};  
int index = 0; int limite = 5; int taille = tableauFibonacci.length;  
System.out.println("Affichage des " + limite + " premiers éléments de la suite de Fibonacci :");
```

```
while(index < limite && index < taille)  
{  
    System.out.println(tableauFibonacci[index++]);  
}  
// Si la condition n'est pas vérifiée d'entrée  
// rien n'est affiché
```

OU

```
do  
{  
    System.out.println(tableauFibonacci[index++]);  
}  
while(index < limite && index < taille);  
//Le premier élément de la suite sera affiché  
//même si la condition n'est pas vérifiée
```

⚠ Lors de l'utilisation d'un **do while**, ne pas oublier le **;** à la fin du while !

Boucles `for`

❶ Pour effectuer un traitement tant qu'une condition n'est pas vérifiée, on peut aussi utiliser une boucle `for`.

```
int[] tableauFibonacci = {0, 0, 1, 2, 3, 5, 8, 13, 21, 34, 55};  
System.out.println("Affichage de tous les éléments de la suite de Fibonacci :");
```

```
//for(initialisation ; condition ; incrémentation)  
for(int i = 0; i < tableauFibonacci.length; i++)  
{  
    //A chaque itération, l'index i est  
    //incrémenté tant qu'il ne dépasse  
    //pas la taille du tableau  
    System.out.println(tableauFibonacci[i]);  
}
```

ou

```
//for(element : collection)  
for(int element : tableauFibonacci)  
{  
    //A chaque itération, element contient  
    //l'élément suivant du tableau jusqu'à  
    //la fin  
    System.out.println(element);  
}
```

Bonus Java 8

```
Arrays.stream(tableauFibonacci).forEach(System.out::println);
```

Interruptions

i Il est parfois nécessaire de sortir d'une boucle avant sa fin, on utilise pour cela l'instruction `break`.

```
int[] tableauFibonacci = {0, 0, 1, 2, 3, 5, 8, 13, 21, 34, 55};  
//On sort dès qu'une valeur est supérieur à 4  
for(int element : tableauFibonacci){  
    if(element > 4){  
        break;  
    }  
    //La portion de code ci-dessous ne sera pas exécutée  
    //lorsque la variable element vaudra 5  
    System.out.print(element + ", ");  
}  
//Affichage 0, 0, 1, 2, 3,
```

```
char [][] sudoku = {  
    {'2', '9', '?'}, {'8', '4', '3'}, {'6', '7', '1'}  
};  
parcours_boucle: //Etiquette parcours_boucle  
for(int i = 0; i < 3; i++){  
    for(int j = 0; j < 3; j++){  
        if(sudoku[i][j] == '?'){  
            break parcours_boucle; //On sort des deux boucles  
        }  
        System.out.print(sudoku[i][j] + ", ");  
    }  
}  
//Affichage 2, 9,
```

i Lorsque deux boucles sont imbriquées et qu'il y a un `break` dans la deuxième boucle, on ne sort que de la dernière.

i Pour quitter des boucles imbriquées, on utilise le mécanisme d'interruption étiquetée en faisant précéder le boucle externe d'un nom suivi de `:`. C'est l'étiquette. Pour sortir de cette boucle, on appelle `break nom_etiquette;`

Interruptions

i On peut vouloir également aller prématièrement au tour de boucle suivant. Pour cela on utilise l'instruction `continue`.

```
int[] tableauFibonacci = {0, 0, 1, 2, 3, 5, 8, 13, 21, 34, 55};

//On passe au tour de boucle suivant dès qu'une valeur paire
for(int element : tableauFibonacci)
{
    if(element % 2 == 0){
        continue;
    }
    //La portion de code ci-dessous ne sera pas exécutée
    //lorsque la variable element est paire
    System.out.print(element + ", ");
}

//Affichage 1, 3, 5, 13, 21, 55
```

Exceptions

Throwable ▾

❶ Ce n'est pas parce qu'un programme compile, qu'il ne comporte pas de bugs, et qu'il ne peut pas y avoir de problèmes à l'exécution. A la compilation, on parle **d'erreurs de compilation** qui empêche de compiler le programme. A l'exécution, on parle **d'exceptions** qui peuvent planter le programme lorsqu'elles ne sont pas gérées correctement.

Fautes de saisie utilisateur
Ressources inexistantes
Problèmes réseaux
Problèmes matériels
Erreurs de programmation
...

❷ On distingue la branche **Error** qui correspond aux erreurs internes au système d'exécution Java et qui ne sont pas vraiment gérables par le programmeur, et la branche **Exception**, qui se sépare elle-même en plusieurs branches, dont notamment les **RuntimeException**.

Les RuntimeExceptions

❶ Une exception `RuntimeException` survient systématiquement à cause d'une erreur de programmation. Les autres correspondent à des erreurs *externes* au programme.

```
int[] tableauFibonacci = {0, 0, 1, 2, 3};  
for(int i = 0; i < 6; i++){  
    //Lorsque i va valoir 5, une exception va être levée  
    //et va interrompre le traitement  
    System.out.print(tableauFibonacci[i] + ", ");  
}  
//Affichage : 0, 0, 1, 2, 3
```

=> Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5 at Fibonacci.main(Fibonacci.java:17)

Endroit où l'exception a été levée. Ici la classe `Fibonacci`, dans la méthode `main`, à la ligne 17

Classe de l'exception qui a été levée (Tentative d'accès à un élément du tableau en dehors des limites), ainsi ce dans ce cas précis, la valeur d'index qui a provoqué l'exception

```
int[] tableau = {5, 1, 0, 3};  
for(int i = 0; i < tableau.length; i++){  
    System.out.print(15 / tableau[i] + ", ");  
}
```

=> Exception in thread "main" java.lang.ArithmetricException: / by zero at Fibonacci.main(Fibonacci.java:17)

❷ En principe, on n'essaye pas de gérer les `RuntimeException`, on parle d'exceptions *hors-contrôle*. Le programmeur doit faire en sorte qu'elles ne surviennent pas.

Les exceptions sous contrôle

❶ Les exceptions **sous contrôle** sont toutes les exceptions en dehors des `RuntimeException` (et de celles qui en héritent). Celles-ci doivent être gérées par le programmeur.

Faire gérer l'exception par le *niveau supérieur*

```
methodeRuntimeException() throws RuntimeException {  
    ...  
    throw new RuntimeException("Erreur...");  
}  
  
//On ne gère pas les RuntimeException  
methodeRuntimeException();  
  
//On signale au programmeur que cette méthode peut lancer des  
//exceptions de type IOException ou EOFException qu'il faudra gérer  
methodeAutreException() throws IOException, EOFException{  
    ...  
    throw new IOException("Erreur...");  
    ...  
    throw new EOFException("Erreur...");  
}
```

Gérer l'exception *localement*

```
try {  
    methodeAutreException();  
} catch(IOException exception) {  
    //Gérer l'exception IOException  
    ...  
} catch(EOFException exception) {  
    //Gérer l'exception EOFException  
    ...  
} finally {  
    //Dans tous les cas (exception ou pas), effectuer ce traitement  
    ...  
}
```

❶ Il est possible de créer ses propres classes d'exceptions pour représenter des erreurs métier.

Conseils sur l'utilisation des exceptions

i Les **exceptions** sont un mécanisme puissant mais il faut les utiliser intelligemment.

La gestion des exceptions ne doit pas remplacer un test simple

Ne pas gérer trop finement les exceptions

Respecter la distinction entre exceptions *hors contrôle* et exceptions *sous-contrôle*

Ne pas gérer silencieusement les exceptions

Ne pas renvoyer de *valeur par défaut* au lieu de lever une exception

Ne pas hésiter à renvoyer des exceptions aux niveaux supérieurs

⚠ Les exceptions ne doivent être utilisées que pour la gestion d'erreurs inhabituelles ou rares car c'est un mécanisme coûteux.

Chaînes

i Nous avons vu qu'un caractère pouvait être représenté par le type `char`. Une chaîne de caractère peut donc être représentée par un tableau de caractères `char[]`.

i Parce que le type `char` est contraignant à utiliser et peu pratique, on préférera utiliser la classe standard `String`.

i C'est d'ailleurs la classe `String` qui est implicitement utilisée par Java lorsqu'on écrit une constante chaîne comme `"Hello"`.

La classe **String**

⚠ Contrairement aux types primitifs, le type **String** est en fait une *classe*. Ce sont des références à des objets qui sont manipulées, pas les valeurs elles-mêmes. On utilise donc l'instruction **new** pour créer une référence vers un objet.

```
//Déclaration d'un objet de type String
String chaine; //chaine sera destinée à contenir une référence à un objet de type String

//Déclaration et initialisation d'un objet de type String
String chaine1 = ""; //référence à une chaîne vide
String chaine2 = new String(); //référence à une nouvelle chaîne vide
String chaine3 = new String("Hello"); //référence à une chaîne contenant "Hello"
String chaine4 = "Hello"; // Le compilateur crée automatiquement un objet de type String contenant "Hello"
```

Les notions d'objet et de classe seront abordées dans le cours [Java 220](#).

Manipulation de chaînes

Opération	Méthode → retour	Exemple
Longueur	<code>length() → int</code>	<code>"Hello".length();//5</code>
Concaténation	<code>concat(String chaineAConcatener) → String</code>	<code>"Hello".concat(" World");//"Hello World"</code> <code>"Hello" + " World";//Equivalent avec l'opérateur +</code>
Egalité	<code>equals(String chaineAComparer) → boolean</code>	<code>"Hello".equals("World");//false</code> <code>"Hello" == "Hello";//true, même références</code> <code>new String("Hello") == "Hello";//false, références différentes</code> <code>"HELLO".equalsIgnoreCase("hello");//true</code>
Sous-chaînes	<code>substring(int indexDebut, int indexFin) → String</code>	<code>"Hello".substring(2, 4);//"ll"</code> <code>"Hello".substring(2, 5);//"llo"</code> <code>"Hello".substring(2); //"llo", par défaut jusqu'à la fin</code>
Suppression des espaces de début et fin	<code>trim() → String</code>	<code>" Hello World ".trim();//"Hello World"</code>
Remplacement	<code>replace(String recherche, String remplacement) → String</code>	<code>"Hello World".replace("o", "ø");//"Hello Wørlد"</code> <code>"Hello World".replace("World", "WORLD");//"Hello WORLD"</code>

Manipulation de chaînes

Opération	Méthode → retour	Exemple
Début/Fin	<code>startsWith(String)</code> et <code>endsWith(String) → boolean</code>	<code>"Hello".startsWith("He");//true</code> <code>"Hello".startsWith("he");//false</code> <code>"Hello".endsWith("o");//true</code>
Ricerca	<code>contains(String chaineARechercher) → boolean</code>	<code>"Hello".contains("el");//true</code>
Casse	<code>toUpperCase() ou toLowerCase() → String</code>	<code>"Hello".toUpperCase();//"HELLO"</code> <code>"Hello".toLowerCase();//"hello"</code>
Expression régulière	<code>matches(String regex) → boolean</code>	<code>"hello".matches("[a-z]*");//alpha minuscule : true</code> <code>"hello".matches("[0-9]*");//numérique entre 0 et 9 : false</code>
Séparation	<code>split(String regex) → String[]</code>	<code>"Hello World !".split(" ");//["Hello", "World", "!"]</code> <code>"Hello World !".split(" ", 2);//["Hello", "World !"]</code>
Recherche index	<code>indexOf(String recherche) ou lastIndexOf(String recherche) → int</code>	<code>"Hello World".indexOf("X");//-1, non trouvé</code> <code>"Hello World".indexOf("W");//6</code> <code>"Hello World".indexOf("o");//4, première occurrence</code> <code>"Hello World".lastIndexOf("o");//7, dernière occurrence</code>

La console

❶ Lorsqu'on lance un programme Java dans un terminal sous Linux, dans une invite de commande sous Windows, ou dans un IDE, une *console* peut afficher des éléments issus du programme ou demander des données à l'utilisateur.

Afficher dans la console

```
System.out.print("Hello ");
System.out.println("World !");
System.out.print("How are you ?");
//Hello World !
//How are you ?
```

Lire dans la console

```
System.out.println("Quel est votre nom ?");
Scanner scanner = new Scanner(System.in);
String ligneSaisie = scanner.nextLine();
//Je tape John suivie de entrée
System.out.println("Hello " + ligneSaisie);
//Hello John
scanner.close(); //Appeler lorsque les saisies sont terminées:
```

⚠ Suivant les programmes, et la manière dont ils sont lancés, il n'y a pas toujours de console disponible. Ainsi pour la journalisation, il faudra préférer utiliser une librairie dédiée. Pour la levée d'erreur, utiliser plutôt le mécanisme d'exceptions. L'utilisation de `System.out.print` est à éviter...

Bonus

La récursivité

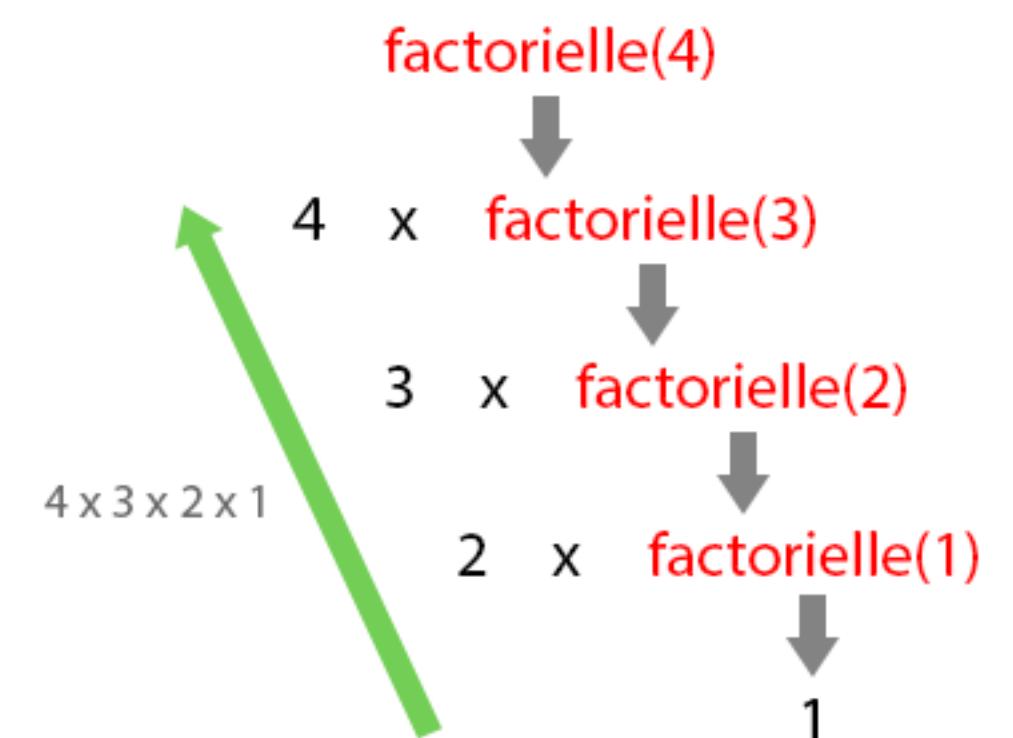
Maitriser son IDE

Les méthodes à *ellipse*

Récursivité

❶ Une fonction est dite récursive lorsqu'elle s'appelle elle-même, c'est-à-dire lorsque dans le corps de cette méthode figure un appel à cette même méthode.
Exemple le plus parlant : la fonction factorielle.

```
static int factorielle(int nombre) {  
    if (nombre == 1) {  
        return 1;  
    } else {  
        return (nombre * factorielle(nombre - 1));  
    }  
}
```



❶ A chaque nouvel appel de `factorielle`, un espace mémoire est alloué permettant de stocker les variables locales, le paramètre `nombre` et la valeur de retour. Les appels s'empilent jusqu'à arriver au `return 1` qui va dépiler les appels et compiler la valeur de retour.

Maitriser son IDE

Auto-complétion

Deboguer

Indenter automatiquement

Réorganiser les imports

Renommer une classe, une méthode, une variable

Refactorer

Générer du code

Connaître les principaux raccourcis

Méthodes à ellipses

❶ En utilisant le principe d'*ellipse*, il est possible de définir des méthodes qui peuvent être appelées avec un nombre variable de paramètres.

```
void affiche(String... textes) {  
    for(String texte : textes){  
        System.out.println(texte);  
    }  
}  
  
affiche("Hello");  
affiche("Hello", "World");  
  
void afficheObjectXFois(int nbFois, Object... objects)  
    for (int i = 0; i < nbFois, i++){  
        separeMotEspaces(objects);  
    }  
}  
  
afficheObjectXFois(5, "Hello");//Affiche 5 fois Hello  
afficheObjectXFois(1, "Hello", 50, true);//Affiche Hello 50 true
```

Un paramètre à *ellipse* est spécifié en indiquant `...` après le type du paramètre. Ici, on ne pourra spécifier que des paramètres de type `String`.

Si la méthode a d'autres paramètres que le paramètre à *ellipse*, ils doivent être spécifier avant. On peut également appeler une autre méthode comportant des paramètres à *ellipse* en passant la variable directement.

❷ Ici la variable `textes` est en fait de type `String[]`. Une variable *ellipse* doit être systématiquement le dernier paramètre dans la signature de la méthode.