what's this !

Java *est un langage de programmation et une plate-forme informatique qui ont été créés par Sun Microsystems en 1995. Beaucoup d'applications et de sites Web ne fonctionnent pas si Java n'est pas installé et leur nombre ne cesse de croître chaque jour. Java est rapide, sécurisé et fiable. Des ordinateurs portables aux centres de données, des consoles de jeux aux super ordinateurs scientifiques, des téléphones portables à Internet, la technologie Java est présente sur tous les fronts !*

Pourquoi utiliser Java ?

- Java fonctionne sur différentes plates-formes (Windows, Mac, Linux, Raspberry Pi, etc.)
- C'est l'un des langages de programmation les plus populaires au monde
- Il est facile à apprendre et simple à utiliser
- Il est open-source et gratuit
- Il est sécurisé, rapide et puissant
- Il a un énorme support communautaire (des dizaines de millions de développeurs)
- Java est un langage orienté objet qui donne une structure claire aux programmes et permet de réutiliser le code, réduisant ainsi les coûts de développement
- Comme Java est proche de C++ et C# , il est facile pour les programmeurs de passer à Java ou vice versa

Installation Java

Certains PC peuvent avoir Java déjà installé.

Pour vérifier si Java est installé sur un PC Windows, recherchez Java dans la barre de démarrage ou tapez ce qui suit dans l'invite de commande (cmd.exe) :

java –version

Si Java est installé, vous verrez quelque chose comme ceci (selon la version) :

java version "11.0.1" 2018-10-16 LTS

Syntaxe Java

nous avons créé un fichier Java appelé Main.java, et nous avons utilisé le code suivant pour afficher "Hello World" à l'écran :

Main.java

```java
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

La méthode principale

Chaque ligne de code qui s'exécute en Java doit être à l'intérieur d'un fichier class. Dans notre exemple, nous avons nommé la classe **Main** . Une classe doit toujours commencer par une première lettre majuscule.

Remarque : Java est sensible à la casse : "MyClass" et "myclass" ont des significations différentes.

Le nom du fichier Java **doit correspondre** au nom de la classe. Lors de l'enregistrement du fichier, enregistrez-le en utilisant le nom de la classe et ajoutez ".java" à la fin du nom de fichier. Pour exécuter l'exemple ci-dessus sur votre ordinateur, assurez-vous que Java est correctement installé, La sortie doit être :

public static void main(String[] args)

La méthode main() est requise et vous la verrez dans chaque programme Java .

Tout code à l'intérieur de la méthode **main()** sera exécuté. Ne vous souciez pas des mots-clés avant et après main. Vous apprendrez à les connaître petit à petit en lisant ce cours.

Pour l'instant, rappelez-vous simplement que chaque programme Java a un nom de la class qui doit correspondre au nom de fichier et que chaque programme doit contenir la méthode main().

System.out.println()

À l'intérieur de la méthode main(), nous pouvons utiliser la méthode println() pour imprimer une ligne de texte à l'écran :

Main.java

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

Remarque :

Les accolades **{}** marquent le début et la fin d'un bloc de code.

System est une classe Java intégrée qui contient des membres utiles, tels que out, qui est l'abréviation de "output". La println() méthode, abréviation de "print line", est utilisée pour imprimer une valeur à l'écran (ou dans un fichier).

Ne vous inquiétez pas trop de System, out et println(). Sachez simplement que vous en avez besoin ensemble pour imprimer des choses à l'écran.

Notez également que chaque instruction de code doit se terminer par un point-virgule (;).

Java Output

Vous avez appris du chapitre précédent que vous pouvez utiliser la méthode println() pour afficher des valeurs ou imprimer du texte en Java :

Example

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

Vous pouvez ajouter autant des méthodes println() que vous le souhaitez. Notez qu'il ajoutera une nouvelle ligne pour chaque méthode :

Java Output

Vous pouvez également générer des nombres et effectuer des calculs mathématiques :

Example

```
System.out.println(3 + 3);
```

Notez que nous n'utilisons pas de guillemets doubles (" ") à l'intérieur println() pour afficher des nombres

Il existe également une méthode print() similaire à println().

La seule différence est qu'il n'insère pas de nouvelle ligne à la fin de la sortie :

Commentaires Java

Les commentaires peuvent être utilisés pour expliquer le code Java et pour le rendre plus lisible. Il peut également être utilisé pour empêcher l'exécution lors du test d'un code alternatif.

Les commentaires sur une seule ligne commencent par deux barres obliques (//).

Tout texte entre // et la fin de la ligne est ignoré par Java (ne sera pas exécuté).

Example

```
// This is a comment
System.out.println("Hello World");
```

Commentaires multi-lignes Java

Les commentaires multilignes commencent par /* et se terminent par */.

Tout texte entre /* et */ sera ignoré par Java.

Cet exemple utilise un commentaire multiligne (un bloc de commentaire) pour expliquer le code :

Example

```
/* The code below will print the words Hello World
to the screen, and it is amazing */
System.out.println("Hello World");
```

Variables Java

Les variables sont des conteneurs pour stocker des valeurs de données.

En Java, il existe différents **types** de variables, par exemple :

- **String** stocke du texte, tel que "Bonjour". Les valeurs de chaîne sont entourées de guillemets doubles
- **int :** stocke des nombres entiers (nombres entiers), sans décimales, comme 123 ou -123
- **float :** stocke les nombres à virgule flottante, avec des décimales, telles que 19,99 ou -19,99
- **char** : stocke des caractères uniques, tels que 'a' ou 'B'. Les valeurs char sont entourées de guillemets simples
- **boolean :** stocke les valeurs avec deux états : vrai ou faux

Déclarer (créer) des variables

Pour créer une variable, vous devez spécifier le type et lui affecter une valeur :

Syntax

```
type variableName = value;
```

T*ype* est l'un des types de Java (tel que int ou String) et *variableName* est le nom de la variable (tel que **x** ou **name**). Le **signe égal** est utilisé pour attribuer des valeurs à la variable.

Exemple

Créez une variable appelée **name** de type **String** et affectez-lui la valeur " **John** " :

```
String name = "John";
System.out.println(name);
```

Variables finales

Si vous ne voulez pas que d'autres (ou vous-même) écrasent les valeurs existantes, utilisez le mot- finalclé (cela déclarera la variable comme "finale" ou "constante", ce qui signifie non modifiable et en lecture seule) :

Example

```
final int myNum = 15;
myNum = 20;  // will generate an error: cannot assign a value to a final variable
```

Autres types

Une démonstration de la façon de déclarer des variables d'autres types :

Example

```
int myNum = 5;
float myFloatNum = 5.99f;
char myLetter = 'D';
boolean myBool = true;
String myText = "Hello";
```

Variables d'affichage

La méthode println() est souvent utilisée pour afficher des variables.

Pour combiner à la fois du texte et une variable, utilisez le + caractère :

Example

```
String name = "John";
System.out.println("Hello " + name);
```

Variables d'affichage

Vous pouvez également utiliser le + caractère pour ajouter une variable à une autre variable :

Example

```
String firstName = "John ";
String lastName = "Doe";
String fullName = firstName + lastName;
System.out.println(fullName);
```

Déclarer plusieurs variables

Pour déclarer plusieurs variables du **même type** , vous pouvez utiliser une liste séparée par des virgules :

Example

Instead of writing:

```
int x = 5;
int y = 6;
int z = 50;
System.out.println(x + y + z);
```

You can simply write:

```
int x = 5, y = 6, z = 50;
System.out.println(x + y + z);
```

Une valeur à plusieurs variables

Vous pouvez également affecter la **même valeur** à plusieurs variables sur une seule ligne :

Example

```
int x, y, z;
x = y = z = 50;
System.out.println(x + y + z);
```

Identifiants

Toutes les variables Java doivent être **identifiées** par **des noms uniques** .

Ces noms uniques sont appelés **identificateurs** .

Les identifiants peuvent être des noms courts (comme x et y) ou des noms plus descriptifs (age, sum, totalVolume).

Remarque : Il est recommandé d'utiliser des noms descriptifs afin de créer un code compréhensible et maintenable :

Types de données Java

Comme expliqué dans le chapitre précédent, une variable en Java doit être un type de données spécifié :

Example

```
int myNum = 5;               // Integer (whole number)
float myFloatNum = 5.99f;    // Floating point number
char myLetter = 'D';         // Character
boolean myBool = true;       // Boolean
String myText = "Hello";     // String
```

Types de données primitifs

Un type de données primitif spécifie la taille et le type des valeurs de variable, et il n'a pas de méthodes supplémentaires.Il existe huit types de données primitifs en Java :

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
`boolean`	1 bit	Stores true or false values
`char`	2 bytes	Stores a single character/letter or ASCII values

Types de données non primitifs

Les types de données non primitifs sont appelés **types de référence** car ils font référence à des objets.

La principale différence entre les types de données **primitifs** et **non primitifs** est :

- Les types primitifs sont prédéfinis (déjà définis) en Java. Les types non primitifs sont créés par le programmeur et ne sont pas définis par Java (sauf pour String).
- Les types non primitifs peuvent être utilisés pour appeler des méthodes afin d'effectuer certaines opérations, contrairement aux types primitifs.
- Un type primitif a toujours une valeur, tandis que les types non primitifs peuvent être null.
- Un type primitif commence par une lettre minuscule, tandis que les types non primitifs commencent par une lettre majuscule.
- La taille d'un type primitif dépend du type de données, tandis que les types non primitifs ont tous la même taille

  Des exemples de types non primitifs sont Strings , Arrays , Classes , Interface , etc. Vous en apprendrez plus à ce sujet dans un chapitre ultérieur.

Java Type Casting

La conversion de type consiste à attribuer une valeur d'un type de données primitif à un autre type.

En Java, il existe deux types de casting :

- **Élargissement du casting** (automatiquement) - conversion d'un type plus petit en un type plus grand
  byte-> short-> char-> int-> long-> float->double

- Narrowing Casting (manuellement) - conversion d'un type plus grand en un type de taille plus petite
  double-> float-> long-> int-> char-> short->byte

Élargissement de la coulée

L'élargissement de la coulée se fait automatiquement lors du passage d'un type de taille plus petite à un type de taille plus grande :

Example

```
public class Main {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }
}
```

Rétrécissement de la coulée

Le casting de restriction doit être effectué manuellement en plaçant le type entre parenthèses devant la valeur :

Example

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

Opérateurs Java

Les opérateurs sont utilisés pour effectuer des opérations sur des variables et des valeurs.

Dans l'exemple ci-dessous, nous utilisons l' **opérateur** + pour additionner deux valeurs :

Example

```
int sum1 = 100 + 50;         // 150 (100 + 50)
int sum2 = sum1 + 250;       // 400 (150 + 250)
int sum3 = sum2 + sum2;      // 800 (400 + 400)
```

Java divise les opérateurs dans les groupes suivants :

- Opérateurs arithmétiques
- Opérateurs d'affectation
- Opérateurs de comparaison
- Opérateurs logiques
- Opérateurs au niveau du bit

Opérateurs arithmétiques

Les opérateurs arithmétiques sont utilisés pour effectuer des opérations mathématiques courantes.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Opérateurs d'affectation Java

L'opérateur **d'affectation d'addition** (**+=**) ajoute une valeur à une variable :

Example

```
int x = 10;
x += 5;
```

Une liste de tous les opérateurs d'affectation :

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Les opérateurs de comparaison

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Opérateurs logiques Java

Les opérateurs logiques sont utilisés pour déterminer la logique entre les variables ou les valeurs :

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Java Strings

Les chaînes sont utilisées pour stocker du texte.

Une String variable contient une collection de caractères entourés de guillemets :

Example

Create a variable of type `String` and assign it a value:

```
String greeting = "Hello";
```

Longueur de chaine

Une chaîne en Java est en fait un objet contenant des méthodes pouvant effectuer certaines opérations sur des chaînes. Par exemple, la longueur d'une chaîne peut être trouvée avec la méthode length() :

Example

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string is: " + txt.length());
```

Plus de méthodes de chaîne

Il existe de nombreuses méthodes de chaîne disponibles, par exemple toUpperCase() et toLowerCase():

Example

```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world"
```

Recherche d'un caractère dans une chaîne

La indexOf() méthode renvoie l' **index** (la position) de la première occurrence d'un texte spécifié dans une chaîne (y compris les espaces):

Example

```
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

Concaténation de chaînes Java

L'opérateur + peut être utilisé entre les chaînes pour les combiner. C'est ce qu'on appelle **la concaténation** :

Example

```
String firstName = "John";
String lastName = "Doe";
System.out.println(firstName + " " + lastName);
```

Vous pouvez également utiliser la méthode concat() pour concaténer deux chaînes :

Example

```
String firstName = "John ";
String lastName = "Doe";
System.out.println(firstName.concat(lastName));
```

Ajouter des nombres et des chaînes

AVERTISSEMENT!

Java utilise l'opérateur + pour l'addition et la concaténation.Les numéros sont ajoutés. Les chaînes sont concaténées.

Si vous additionnez deux nombres, le résultat sera un nombre :

Example

```
int x = 10;
int y = 20;
int z = x + y;  // z will be 30 (an integer/number)
```

Si vous ajoutez deux chaînes, le résultat sera une concaténation de chaînes :

Example

```
String x = "10";
String y = "20";
String z = x + y;  // z will be 1020 (a String)
```

Si vous ajoutez un nombre et une chaîne, le résultat sera une concaténation de chaînes :

Example

```
String x = "10";
int y = 20;
String z = x + y;  // z will be 1020 (a String)
```

Caractères spéciaux

Étant donné que les chaînes doivent être écrites entre guillemets, Java comprendra mal cette chaîne et générera une erreur :

```
String txt = "We are the so-called "Vikings" from the north.";
```

La solution pour éviter ce problème est d'utiliser le **caractère d'échappement antislash** .

Le caractère d'échappement barre oblique inverse (\ ) transforme les caractères spéciaux en caractères de chaîne :

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

La séquence \" insère un guillemet double dans une chaîne :

```
String txt = "We are the so-called \"Vikings\" from the north.";
```

La séquence \' insère un guillemet simple dans une chaîne :

```
String txt = "It\'s alright.";
```

Six autres séquences d'échappement sont valides en Java :

Code	Result
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed

Mathématiques Java

La classe Java Math possède de nombreuses méthodes qui vous permettent d'effectuer des tâches mathématiques sur des nombres.

Math.max(*x*,*y*)

La méthode peut être utilisée pour trouver la valeur la plus élevée de *x* et *y* : Math.max(*x*,*y*)

Example

```
Math.max(5, 10);
```

Math.sqrt(*x*)

La méthode renvoie la racine carrée de *x* : Math.sqrt(*x*)

Example

```
Math.sqrt(64);
```

Math.abs(*x*)

La méthode renvoie la valeur absolue (positive) de *x* : Math.abs(*x*)

Example

```
Math.sqrt(64);
```

Nombres aléatoires

Math.random() renvoie un nombre aléatoire entre 0,0 (inclus) et 1,0 (exclusif) :

Pour obtenir plus de contrôle sur le nombre aléatoire, par exemple vous ne voulez qu'un nombre aléatoire entre 0 et 100, vous pouvez utiliser la formule suivante :

Example

```
int randomNum = (int)(Math.random() * 101);  // 0 to 100
```

Valeurs booléennes

Très souvent, en programmation, vous aurez besoin d'un type de données qui ne peut avoir qu'une des deux valeurs, comme :

- OUI NON
- ALLUMÉ ÉTEINT
- VRAI FAUX

Pour cela, Java dispose d'un type boolean de données, qui peut prendre les valeurs true ou false.

Example

```
boolean isJavaFun = true;
boolean isFishTasty = false;
System.out.println(isJavaFun);     // Outputs true
System.out.println(isFishTasty);   // Outputs false
```

Expression booléenne

Une **expression booléenne** est une expression Java qui renvoie une valeur **booléenne** : true ou false.

Vous pouvez utiliser un opérateur de comparaison, tel que l'opérateur **supérieur à** (>) pour savoir si une expression (ou une variable) est vraie :

Example

```
int x = 10;
int y = 9;
System.out.println(x > y); // returns true, because 10 is higher than 9
```

Java Conditions and If Statements

Java prend en charge les conditions logiques habituelles des mathématiques :

- Inférieur à : a < b
- Inférieur ou égal à : a <= b
- Supérieur à : a > b
- Supérieur ou égal à : a >= b
- Égal à a == b
- Différent de : a != b

Vous pouvez utiliser ces conditions pour effectuer différentes actions pour différentes décisions.

Java contient les instructions conditionnelles suivantes :

- Utiliser **if** pour spécifier un bloc de code à exécuter, si une condition spécifiée est vraie
- Sert **else** à spécifier un bloc de code à exécuter, si la même condition est fausse
- Permet **else if** de spécifier une nouvelle condition à tester, si la première condition est fausse
- Utilisez **switch** pour spécifier de nombreux blocs de code alternatifs à exécuter

L'instruction if

Utilisez l' if instruction pour spécifier un bloc de code Java à exécuter si une condition est true.

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Dans l'exemple ci-dessous, nous testons deux valeurs pour savoir si 20 est supérieur à 18. Si la condition est true, imprimez du texte :

```
if (20 > 18) {
  System.out.println("20 is greater than 18");
}
```

La déclaration else

Utilisez l' else instruction pour spécifier un bloc de code à exécuter si la condition est false.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Dans l'exemple le temps (20) est supérieur à 18, donc la condition est false. Pour cette raison, nous passons à la else condition et imprimons à l'écran "Bonsoir". Si l'heure était inférieure à 18, le programme imprimerait "Bonne journée".

Example

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

L'instruction else if

Utilisez l' else if instruction pour spécifier une nouvelle condition si la première condition est false.

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

Dans l'exemple, le temps (22) est supérieur à 10, donc la **première condition** est false. La condition suivante, dans la else **if** déclaration, est également false, nous passons donc à la **else** condition puisque **condition1** et **condition2** sont les deux false, et affichons à l'écran "Bonsoir".

Example

```
int time = 22;
if (time < 10) {
  System.out.println("Good morning.");
} else if (time < 20) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

Short Hand If...Else

Il existe également un raccourci if else , appelé **opérateur ternaire** car il se compose de trois opérandes.

Il peut être utilisé pour remplacer plusieurs lignes de code par une seule ligne, et est le plus souvent utilisé pour remplacer de simples instructions if else :

Syntax

```
variable = (condition) ? expressionTrue :  expressionFalse;
```

```java
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

Vous pouvez simplement écrire :

Example

```java
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

Java Switch Statements

Utilisez l'instruction switch pour sélectionner l'un des nombreux blocs de code à exécuter.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Java Switch Statements

Voilà comment cela fonctionne:

- L'expression **switch** est évaluée une fois.
- La valeur de l'expression est comparée aux valeurs de chaque case.
- S'il y a correspondance, le bloc de code associé est exécuté.
- Les mots clés **break** et **default** sont facultatifs et seront décrits plus loin dans ce chapitre

Le mot-clé default spécifie du code à exécuter s'il n'y a pas de correspondance de casse :

Example

```
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

Java While Loop

Les boucles peuvent exécuter un bloc de code tant qu'une condition spécifiée est atteinte.

Les boucles sont pratiques car elles permettent de gagner du temps, de réduire les erreurs et de rendre le code plus lisible.

Java While Boucle

La boucle while parcourt un bloc de code tant qu'une condition spécifiée est **true** :

Syntax

```
while (condition) {
  // code block to be executed
}
```

Dans l'exemple ci-dessous, le code de la boucle s'exécutera, encore et encore, tant qu'une variable (i) est inférieure à 5 :

Example

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

The Do / While Loop

La boucle do / while est une variante de la boucle while. Cette boucle exécutera le bloc de code une fois, avant de vérifier si la condition est vraie, puis elle répétera la boucle tant que la condition est vraie.

Syntax

```
do {
  // code block to be executed
}
while (condition);
```

L'exemple ci-dessous utilise une boucle do / while. La boucle sera toujours exécutée au moins une fois, même si la condition est fausse, car le bloc de code est exécuté avant que la condition ne soit testée :

Example

```
int i = 0;
do {
  System.out.println(i);
  i++;
}
while (i < 5);
```

Java For Loop

Lorsque vous savez exactement combien de fois vous voulez parcourir un bloc de code, utilisez la boucle for au lieu d'une boucle while :

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

L' instruction 1 est exécutée (une fois) avant l'exécution du bloc de code.

L' instruction 2 définit la condition d'exécution du bloc de code.

L' instruction 3 est exécutée (à chaque fois) après l'exécution du bloc de code.

L'exemple ci-dessous imprimera les chiffres de 0 à 4 :

Example

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

Java For Each Loop

Il existe également une boucle " **for-each ",** qui est utilisée exclusivement pour parcourir les éléments d'un tableau :

Syntax

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

L'exemple suivant affiche tous les éléments du tableau **cars** , à l'aide d'une boucle " **for-each " :**

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

Java Break and Continue

Vous avez déjà vu l'instruction break utilisée dans un chapitre précédent de ce didacticiel. Il a été utilisé pour "sauter" d'une déclaration switch.

L'instruction break peut également être utilisée pour sortir d'une **boucle** .

Cet exemple arrête la boucle lorsque **i** est égal à 4 :

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  System.out.println(i);
}
```

Java Continuer

L'nterrompt continue une itération (dans la boucle), si une condition spécifiée se produit, et continue avec l'itération suivante dans la boucle.

Cet exemple ignore la valeur de 4 :

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  System.out.println(i);
}
```

Java Arrays

Les tableaux sont utilisés pour stocker plusieurs valeurs dans une seule variable, au lieu de déclarer des variables distinctes pour chaque valeur.

Pour déclarer un tableau, définissez le type de la variable **entre crochets** :

```
String[] cars;
```

Nous avons maintenant déclaré une variable qui contient un tableau de chaînes. Pour y insérer des valeurs, nous pouvons utiliser un tableau littéral - placez les valeurs dans une liste séparée par des virgules, à l'intérieur d'accolades :

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

Pour créer un tableau d'entiers, vous pouvez écrire :

```
int[] myNum = {10, 20, 30, 40};
```

Accéder aux éléments d'un tableau

Vous accédez à un élément de tableau en vous référant au numéro d'index.

Cette instruction accède à la valeur du premier élément dans cars

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
```

Modifier un élément de tableau

Pour modifier la valeur d'un élément spécifique, reportez-vous au numéro d'index :

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo
```

Longueur du tableau

Pour connaître le nombre d'éléments d'un tableau, utilisez la propriété length:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4
```

Java Arrays Loop

Vous pouvez parcourir les éléments du tableau avec la for boucle et utiliser la propriété length pour spécifier combien de fois la boucle doit s'exécuter.

L'exemple suivant affiche tous les éléments du tableau **cars :**

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

Boucle dans un tableau avec For-Each

Il existe également une boucle " **for-each ",** qui est utilisée exclusivement pour parcourir les éléments des tableaux :

Syntax

```
for (type variable : arrayname) {
  ...
}
```

L'exemple suivant affiche tous les éléments du tableau **cars** , à l'aide d'une boucle " **for-each " :**

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

Tableaux multi dimensionnels Java

Un tableau multi dimensionnel est un tableau de tableaux.

Pour créer un tableau à deux dimensions, ajoutez chaque tableau dans son propre ensemble d' **accolades** :

Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
int x = myNumbers[1][2];
System.out.println(x); // Outputs 7
```

On peut aussi utiliser un à l'intérieur **boucle for** d'un autre **boucle for** pour obtenir les éléments d'un tableau à deux dimensions (il faut encore pointer sur les deux index) :

Example

```
public class Main {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    for (int i = 0; i < myNumbers.length; ++i) {
      for(int j = 0; j < myNumbers[i].length; ++j) {
        System.out.println(myNumbers[i][j]);
      }
    }
  }
}
```

Java POO

Programmation Orientée Objet .

La programmation procédurale consiste à écrire des procédures ou des méthodes qui effectuent des opérations sur les données, tandis que la programmation orientée objet consiste à créer des objets contenant à la fois des données et des méthodes.

La programmation orientée objet présente plusieurs avantages par rapport à la programmation procédurale :

- La POO est plus rapide et plus facile à exécuter
- La POO fournit une structure claire pour les programmes
- La POO aide à garder le code Java DRY "Ne vous répétez pas", et facilite la maintenance, la modification et le débogage du code
- La POO permet de créer des applications entièrement réutilisables avec moins de code et un temps de développement plus court

Java - Que sont les classes et les objets ?

Les classes et les objets sont les deux principaux aspects de la programmation orientée objet.

Regardez l'illustration suivante pour voir la différence entre la classe et les objets :

Classes/Objets Java

Tout en Java est associé à des classes et des objets, ainsi qu'à ses attributs et méthodes. Par exemple : dans la vraie vie, une voiture est un objet. La voiture a des **attributs** , tels que le poids et la couleur, et **des méthodes** , telles que la conduite et le freinage.

Une classe est comme un constructeur d'objets ou un "plan" pour créer des objets.

Créer une classe

Pour créer une classe, utilisez le mot clé class:

Main.java

Create a class named "`Main`" with a variable x:

```
public class Main {
  int x = 5;
}
```

Créer un objet

En Java, un objet est créé à partir d'une classe.

Nous avons déjà créé la classe nommée, nous pouvons donc maintenant l'utiliser pour créer des objets.

Pour créer un objet de Main, spécifiez le nom de la classe, suivi du nom de l'objet, et utilisez le mot-clé **new**.

Example

Create an object called "`myObj`" and print the value of x:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Objets multiples

Vous pouvez créer plusieurs objets d'une même classe :

Example

Create two objects of `Main`:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

Utilisation de plusieurs classes

Vous pouvez également créer un objet d'une classe et y accéder dans une autre classe. Ceci est souvent utilisé pour une meilleure organisation des classes (une classe a tous les attributs et méthodes, tandis que l'autre classe contient la méthode **main()** (code à exécuter)).

N'oubliez pas que le nom du fichier Java doit correspondre au nom de la classe. Dans cet exemple, nous avons créé deux fichiers dans le même répertoire/dossier :

Main.java

Second.java

Main.java

```
public class Main {
  int x = 5;
}
```

Second.java

```
class Second {
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Java Class Attributes

Dans le chapitre précédent, nous avons utilisé le terme "variable" pour x dans l'exemple (comme indiqué ci-dessous). C'est en fait un **attribut** de la classe. Ou vous pourriez dire que les attributs de classe sont des variables au sein d'une classe :

Example

Create a class called "`Main`" with two attributes: `x` and `y`:

```
public class Main {
  int x = 5;
  int y = 3;
}
```

Accéder aux attributs

Vous pouvez accéder aux attributs en créant un objet de la classe et en utilisant la syntaxe à points (`.`) :

L'exemple suivant créera un objet de la classe Main, avec le nom myObj. Nous utilisons l' x attribut sur l'objet pour imprimer sa valeur :

Example

Create an object called " `myObj` " and print the value of `x` :

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Modifier les attributs

Vous pouvez également modifier les valeurs d'attribut :

Example

Set the value of `x` to 40:

```
public class Main {
  int x;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

Ou remplacez les valeurs existantes :

Example

Change the value of `x` to 25:

```
public class Main {
  int x = 10;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // x is now 25
    System.out.println(myObj.x);
  }
}
```

Si vous ne souhaitez pas pouvoir remplacer les valeurs existantes, déclarez l'attribut comme final :

Le mot-clé final est utile lorsque vous souhaitez qu'une variable stocke toujours la même valeur, comme PI (3.14159...).

Example

```
public class Main {
  final int x = 10;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // will generate an error: cannot assign a value to a final variable
    System.out.println(myObj.x);
  }
}
```

Objets multiples

Si vous créez plusieurs objets d'une même classe, vous pouvez modifier les valeurs d'attribut d'un objet, sans affecter les valeurs d'attribut de l'autre :

Example

Change the value of `x` to 25 in `myObj2`, and leave `x` in `myObj1` unchanged:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    myObj2.x = 25;
    System.out.println(myObj1.x);  // Outputs 5
    System.out.println(myObj2.x);  // Outputs 25
  }
}
```

Attributs multiples

Vous pouvez spécifier autant d'attributs que vous le souhaitez :

Example

```
public class Main {
  String fname = "John";
  String lname = "Doe";
  int age = 24;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }
}
```

Méthodes de classe Java

Vous avez appris dans le chapitre sur les méthodes Java que les méthodes sont déclarées dans une classe et qu'elles sont utilisées pour effectuer certaines actions :

Créez une méthode nommée myMethod() dans Main :

```
public class Main {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

myMethod() imprime un texte (l'action), lorsqu'elle est **appelée** . Pour appeler une méthode, écrivez le nom de la méthode suivi de deux parenthèses **()** et d'un point-virgule **;**

A l' intérieur main, appelez myMethod():

```
public class Main {
  static void myMethod() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "Hello World!"
```

Statique vs non statique

Vous verrez souvent des programmes Java qui ont des attributs et des méthodes soit statique, public

Dans l'exemple ci-dessus, nous avons créé une static méthode, ce qui signifie qu'elle est accessible sans créer d'objet de la classe, contrairement à public, qui n'est accessible que par des objets :

Un exemple pour illustrer les différences entre les **méthodes** statique et public

```java
public class Main {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would compile an error

    Main myObj = new Main(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method on the object
  }
}
```

Méthodes d'accès avec un objet

Créez un objet Voiture nommé myCar. Appelez les méthodes fullThrottle() et sur l' objet et exécutez le programme : myCar.speed()

```java
// Create a Main class
public class Main {

  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }

  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Main myCar = new Main();   // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
}

// The car is going as fast as it can!
// Max speed is: 200
```

Exemple expliqué

) Nous avons créé une Main classe personnalisée avec le mot-clé class.

2) Nous avons créé les méthodes fullThrottle() et speed() dans la classe **Main**.

3) La méthode fullThrottle() et la méthode speed() impriment du texte lorsqu'elles sont appelées.

4) La méthode speed() accepte un **int** paramètre appelé maxSpeed, nous l'utiliserons en **8)** .

5) Afin d'utiliser la classe **Main** et ses méthodes, nous devons créer un **objet** de la Main classe.

6) Ensuite, allez à la méthode main(), que vous savez maintenant être une méthode Java intégrée qui exécute votre programme (tout code à l'intérieur de main est exécuté).

7) En utilisant le mot-clé **new**, nous avons créé un objet avec le nom myCar.

8) Ensuite, nous appelons les méthodes fullThrottle() et sur l' objet, et exécutons le programme en utilisant le nom de l'objet ( ), suivi d'un point ( ), suivi du nom de la méthode (et). Notez que nous ajoutons un paramètre de **200** à l'intérieur de la méthode speed() myCarmyCar. fullThrottle() ; speed(200) ; int speed()

Rappelez-vous que…

Le point (.) est utilisé pour accéder aux attributs et aux méthodes de l'objet.

Pour appeler une méthode en Java, écrivez le nom de la méthode suivi d'un ensemble de parenthèses **( )** , suivi d'un point-virgule (;) .

Une classe doit avoir un nom de fichier correspondant (Main et **Main.java**).

Utilisation de plusieurs classes

Comme nous l'avons précisé dans le chapitre Classes , il est de bonne pratique de créer un objet d'une classe et d'y accéder dans une autre classe.

N'oubliez pas que le nom du fichier Java doit correspondre au nom de la classe. Dans cet exemple, nous avons créé deux fichiers dans le même répertoire :

Main.java

Second.java

```java
public class Main {
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
}
```

```java
class Second {
  public static void main(String[] args) {
    Main myCar = new Main();     // Create a myCar object
    myCar.fullThrottle();       // Call the fullThrottle() method
    myCar.speed(200);           // Call the speed() method
  }
}
```

Constructeurs Java

Un constructeur en Java est une **méthode spéciale** utilisée pour initialiser des objets. Le constructeur est appelé lorsqu'un objet d'une classe est créé. Il peut être utilisé pour définir des valeurs initiales pour les attributs d'objet :

```
// Create a Main class
public class Main {
  int x;  // Create a class attribute

  // Create a class constructor for the Main class
  public Main() {
    x = 5;  // Set the initial value for the class attribute x
  }

  public static void main(String[] args) {
    Main myObj = new Main(); // Create an object of class Main (This will call the constructor)
    System.out.println(myObj.x); // Print the value of x
  }
}

// Outputs 5
```

Les notes

Notez que le nom du constructeur doit **correspondre au nom de la classe** et qu'il ne peut pas avoir de **type de retour** (comme void).

Notez également que le constructeur est appelé lorsque l'objet est créé.

Toutes les classes ont des constructeurs par défaut : si vous ne créez pas vous-même un constructeur de classe, Java en crée un pour vous. Cependant, vous ne pouvez pas définir de valeurs initiales pour les attributs d'objet.

Paramètres du constructeur

Les constructeurs peuvent également prendre des paramètres, qui sont utilisés pour initialiser les attributs.

L'exemple suivant ajoute un int y paramètre au constructeur. À l'intérieur du constructeur, nous définissons x sur y (x = y). Lorsque nous appelons le constructeur, nous passons un paramètre au constructeur (5), qui définira la valeur de x à 5 :

Paramètres du constructeur

Example

```
public class Main {
  int x;

  public Main(int y) {
    x = y;
  }

  public static void main(String[] args) {
    Main myObj = new Main(5);
    System.out.println(myObj.x);
  }
}

// Outputs 5
```

Vous pouvez avoir autant de paramètres que vous le souhaitez :

```
public class Main {
  int modelYear;
  String modelName;

  public Main(int year, String name) {
    modelYear = year;
    modelName = name;
  }

  public static void main(String[] args) {
    Main myCar = new Main(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}

// Outputs 1969 Mustang
```

Modificateurs Java

À présent, vous connaissez bien le mot-clé public qui apparaît dans presque tous nos exemples :

```
public class Main
```

Le mot-clé public est un modificateur d'accès , ce qui signifie qu'il est utilisé pour définir le niveau d'accès pour les classes, les attributs, les méthodes et les constructeurs.

Nous divisons les modificateurs en deux groupes :

- **Modificateurs d'accès - contrôle le niveau d'accès**
- **Modificateurs non-accès - ne contrôlent pas le niveau d'accès, mais fournissent d'autres fonctionnalités**

Pour les classes , vous pouvez utiliser public ou *default* :

Public : La classe est accessible par n'importe quelle autre classe

Default :La classe n'est accessible que par les classes du même package. Ceci est utilisé lorsque vous ne spécifiez pas de modificateur. Vous en apprendrez plus sur les packages dans le chapitre Packages

Pour les attributs, les méthodes et les constructeurs

public : Le code est accessible pour toutes les classes

private : Le code n'est accessible qu'au sein de la classe déclarée

default : Le code n'est accessible que dans le même package. Ceci est utilisé lorsque vous ne spécifiez pas de modificateur. Vous en apprendrez plus sur les packages dans le chapitre Packages

protected : Le code est accessible dans le même package et sous-classes. Vous en apprendrez plus sur les sous-classes et les super-classes dans le chapitre Héritage

Modificateurs sans accès

Pour **les cours** , vous pouvez utiliser soit **final** soit **abstract**:

Final : La classe ne peut pas être héritée par d'autres classes (Vous en apprendrez plus sur l'héritage dans le chapitre Héritage)

Abstract : La classe ne peut pas être utilisée pour créer des objets (Pour accéder à une classe abstraite, elle doit être héritée d'une autre classe. Vous en apprendrez plus sur l'héritage et l'abstraction dans les chapitres Héritage et Abstraction)

Modificateurs sans accès

Pour **les attributs et les méthodes** , vous pouvez utiliser l'une des options suivantes :

Final : Les attributs et les méthodes ne peuvent pas être remplacés / modifiés

Static : Les attributs et les méthodes appartiennent à la classe, plutôt qu'à un objet

Abstract : Ne peut être utilisé que dans une classe abstraite et ne peut être utilisé que sur des méthodes. La méthode n'a pas de corps, par exemple abstract void run();. Le corps est fourni par la sous-classe (héritée de). Vous en apprendrez plus sur l'héritage et l'abstraction dans les chapitres Héritage et Abstraction

Modificateurs sans accès

suite …

transient : Les attributs et les méthodes sont ignorés lors de la sérialisation de l'objet qui les contient

synchronized : Les méthodes ne sont accessibles que par un thread à la fois

volatile : La valeur d'un attribut n'est pas mise en cache localement dans le thread et est toujours lue à partir de la "mémoire principale"

Final

Si vous ne souhaitez pas pouvoir remplacer les valeurs d'attribut existantes, déclarez les attributs comme final :

Example

```
public class Main {
  final int x = 10;
  final double PI = 3.14;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 50; // will generate an error: cannot assign a value to a final variable
    myObj.PI = 25; // will generate an error: cannot assign a value to a final variable
    System.out.println(myObj.x);
  }
}
```

Abstrait

Une méthode abstrac appartient à une classe abstract et n'a pas de corps. Le corps est fourni par la sous-classe :

```java
// Code from filename: Main.java
// abstract class
abstract class Main {
  public String fname = "John";
  public int age = 24;
  public abstract void study(); // abstract method
}

// Subclass (inherit from Main)
class Student extends Main {
  public int graduationYear = 2018;
  public void study() { // the body of the abstract method is provided here
    System.out.println("Studying all day long");
  }
}
// End code from filename: Main.java

// Code from filename: Second.java
class Second {
  public static void main(String[] args) {
    // create an object of the Student class (which inherits attributes and methods from Main)
    Student myObj = new Student();

    System.out.println("Name: " + myObj.fname);
    System.out.println("Age: " + myObj.age);
    System.out.println("Graduation Year: " + myObj.graduationYear);
    myObj.study(); // call abstract method
  }
}
```

Statique

Une méthode static signifie qu'elle est accessible sans créer d'objet de la classe, contrairement à public:

```
public class Main {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[ ] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would output an error

    Main myObj = new Main(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method
  }
}
```

Encapsulation Java

La signification d' **Encapsulation** est de s'assurer que les données "sensibles" sont cachées aux utilisateurs. Pour y parvenir, vous devez :

déclarer les variables / attributs de la classe comme private

fournir des méthodes publiques **get** et **set** private pour accéder et mettre à jour la valeur d'une variable

Obtenir et définir

Vous avez appris du chapitre précédent que **private** les variables ne sont accessibles qu'au sein d'une même classe (une classe extérieure n'y a pas accès). Cependant, il est possible d'y accéder si nous fournissons des méthodes **get** et **set publiques.**

La méthode **get** renvoie la valeur de la variable et la méthode **set** définit la valeur.

La syntaxe pour les deux est qu'ils commencent par **get** ou **set**, suivi du nom de la variable, avec la première lettre en majuscule :

Example

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

Exemple expliqué

La méthode get renvoie la valeur de la variable name.

La méthode set prend un paramètre (newName) et l'affecte à la variable **name**. Le this mot clé est utilisé pour faire référence à l'objet courant.

Cependant, comme la variable **name** est déclarée en tant que **private**, nous ne pouvons pas **y** accéder depuis l'extérieur de cette classe :

Example

```
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.name = "John";  // error
    System.out.println(myObj.name); // error
  }
}
```

Au lieu de cela, nous utilisons les méthodes **getName**() et **setName**() pour accéder et mettre à jour la variable :

Example

```
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.setName("John"); // Set the value of the name variable to "John"
    System.out.println(myObj.getName());
  }
}

// Outputs "John"
```

Pourquoi l'encapsulation ?

Meilleur contrôle des attributs de classe et des méthodes

Les attributs de classe peuvent être rendus en **lecture seule** (si vous n'utilisez que la méthode **get**) ou en **écriture seule** (si vous n'utilisez que la méthode set)

Flexible : le programmeur peut modifier une partie du code sans affecter les autres parties

Sécurité accrue des données

Java Packages

Un package en Java est utilisé pour regrouper des classes liées. Considérez-le comme **un dossier dans un répertoire de fichiers** . Nous utilisons des packages pour éviter les conflits de noms et pour écrire un code plus maintenable. Les forfaits sont divisés en deux catégories :

Packages intégrés (packages de l'API Java)

Packages définis par l'utilisateur (créez vos propres packages)

Forfaits intégrés

L'API Java est une bibliothèque de classes pré-écrites, libres d'utilisation, incluse dans l'environnement de développement Java.

La bibliothèque contient des composants pour la gestion des entrées, la programmation de la base de données et bien plus encore. La liste complète peut être trouvée sur le site Web d'Oracle : https://docs.oracle.com/javase/8/docs/api/ .

La bibliothèque est divisée en **packages** et **classes** . Cela signifie que vous pouvez soit importer une seule classe (avec ses méthodes et ses attributs), soit un package complet contenant toutes les classes appartenant au package spécifié.

Pour utiliser une classe ou un package de la bibliothèque, vous devez utiliser le mot-clé import :

Syntax

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Importer une classe

Si vous trouvez une classe que vous souhaitez utiliser, par exemple, la classe **Scanner** qui est utilisée pour obtenir l'entrée utilisateur , écrivez le code suivant :

Dans l'exemple, java.util est un package, tandis que **Scanner** est une classe du java.util package.

Example

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

Pour utiliser la Scanner classe, créez un objet de la classe et utilisez l'une des méthodes disponibles trouvées dans la Scanner documentation de la classe. Dans notre exemple, nous utiliserons la méthode nextLine() qui permet de lire une ligne complète :

Utilisation de la classe **Scanner** pour obtenir l'entrée de l'utilisateur :

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

Importer un package

Il existe de nombreux forfaits à choisir. Dans l'exemple précédent, nous avons utilisé la classe Scanner du package java.util. Ce package contient également des installations de date et d'heure, un générateur de nombres aléatoires et d'autres classes d'utilitaires.

Pour importer un package entier, terminez la phrase par un astérisque (*). L'exemple suivant importera TOUTES les classes du package java.util :

Example

```
import java.util.*;
```

User-defined Packages

Pour créer votre propre package, vous devez comprendre que Java utilise un répertoire de système de fichiers pour les stocker. Tout comme les dossiers sur votre ordinateur :

Example

```
└── root
    └── mypack
        └── MyPackageClass.java
```

Pour créer un package, utilisez le mot- packageclé :

remarque :

Le nom du package doit être écrit en minuscules pour éviter tout conflit avec les noms de classe.

MyPackageClass.java

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Héritage Java

En Java, il est possible d'hériter d'attributs et de méthodes d'une classe à une autre. Nous regroupons le "concept d'héritage" en deux catégories :

sous- classe (enfant) - la classe qui hérite d'une autre classe

superclasse (parent) - la classe héritée de

Pour hériter d'une classe, utilisez le extends mot clé.

Dans l'exemple ci-dessous, la classe Car (sous-classe) hérite des attributs et des méthodes de la Vehicleclasse (superclasse) :

```
class Vehicle {
  protected String brand = "Ford";        // Vehicle attribute
  public void honk() {                    // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";    // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from th
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

The final Keyword

Si vous ne voulez pas que d'autres classes héritent d'une classe, utilisez le mot-clé : final

```
final class Vehicle {
  ...
}

class Car extends Vehicle {
  ...
}
```

Java Polymorphism

Le polymorphisme signifie "plusieurs formes", et il se produit lorsque nous avons de nombreuses classes liées les unes aux autres par héritage.

Comme nous l'avons précisé dans le chapitre précédent; L'héritage nous permet d'hériter des attributs et des méthodes d'une autre classe. Le polymorphisme utilise ces méthodes pour effectuer différentes tâches. Cela nous permet d'effectuer une même action de différentes manières.

Par exemple

Pensez à une super classe appelée Animal qui a une méthode appelée Animal Sound(). Les sous-classes d'animaux peuvent être des cochons, des chats, des chiens, des oiseaux - et ils ont aussi leur propre implémentation d'un son d'animal (le cochon ronfle, et le chat miaule, etc.) :

Example

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Rappelez-vous du **chapitre Héritage** que nous utilisons le mot-clé extends pour hériter d'une classe.

Nous pouvons maintenant créer des objets et appeler la méthode Pig sur les deux : Dog animalSound()

```java
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

Pourquoi et quand utiliser "Héritage" et "Polymorphisme" ?

C'est utile pour la réutilisabilité du code : réutilisez les attributs et les méthodes d'une classe existante lorsque vous créez une nouvelle classe.

Java Inner Classes

En Java, il est également possible d'imbriquer des classes (une classe dans une classe). Le but des classes imbriquées est de regrouper des classes qui vont ensemble, ce qui rend votre code plus lisible et maintenable.

Pour accéder à la classe interne, créez un objet de la classe externe, puis créez un objet de la classe interne :

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

Classe intérieure privée

Contrairement à une classe "normale", une classe interne peut être private ou protected. Si vous ne voulez pas que les objets extérieurs accèdent à la classe interne, déclarez la classe comme private:

```
class OuterClass {
  int x = 10;

  private class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

Classe interne statique

Une classe interne peut également être static, ce qui signifie que vous pouvez y accéder sans créer d'objet de la classe externe :

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}

// Outputs 5
```

Remarque :

Tout comme les attributs static et les méthodes, une classe static interne n'a pas accès aux membres de la classe externe.

Accéder à la classe externe à partir de la classe interne

L'un des avantages des classes internes est qu'elles peuvent accéder aux attributs et aux méthodes de la classe externe :

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}

// Outputs 5
```

Classes abstraites et méthodes

L' abstraction de données est le processus consistant à masquer certains détails et à ne montrer que les informations essentielles à l'utilisateur.

L'abstraction peut être réalisée avec des classes abstraites ou des interfaces (sur lesquelles vous en apprendrez plus dans le chapitre suivant).

Le mot-clé abstract est un modificateur de non-accès, utilisé pour les classes et les méthodes :

Classe abstraite : est une classe restreinte qui ne peut pas être utilisée pour créer des objets (pour accéder y, elle doit être héritée d'une autre classe).

Méthode abstraite : ne peut être utilisée que dans une classe abstraite et n'a pas de corps. Le corps est fourni par la sous-classe (héritée de).

Une classe abstraite peut avoir à la fois des méthodes abstraites et régulières :

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

A partir de l'exemple ci-dessus, il n'est pas possible de créer un objet de la classe Animal :

```
Animal myObj = new Animal(); // will generate an error
```

Pour accéder à la classe abstraite, elle doit être héritée d'une autre classe. Convertissons la classe Animal que nous avons utilisée dans le chapitre Polymorphisme en une classe abstraite :

Rappelez-vous du chapitre Héritage que nous utilisons le mot- extends clé pour hériter d'une classe.

```java
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Pourquoi et quand utiliser des classes et des méthodes abstraites ?

Pour assurer la sécurité, masquez certains détails et affichez uniquement les détails importants d'un objet.

Remarque : L'abstraction peut également être réalisée avec Interfaces , dont vous en apprendrez plus dans le chapitre suivant.

Java Interface

Une autre façon de réaliser **l'abstraction** en Java consiste à utiliser des interfaces.

L'interface est une " **classe abstraite** " complètement utilisée pour regrouper des méthodes liées avec des corps vides :

Example

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

Pour accéder aux méthodes d'interface, l'interface doit être "implémentée" (un peu comme héritée) par une autre classe avec le mot-clé implements (au lieu de extends). Le corps de la méthode d'interface est fourni par la classe "implement":

```java
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Remarques sur les interfaces :

Comme **les classes abstraites** , les interfaces **ne peuvent pas** être utilisées pour créer des objets (dans l'exemple ci-dessus, il n'est pas possible de créer un objet "Animal" dans la Class MyMain)

Les méthodes d'interface n'ont pas de corps - le corps est fourni par la classe "implémenter"

Lors de l'implémentation d'une interface, vous devez surcharger toutes ses méthodes

Les méthodes d'interface sont par défaut abstract et public

Les attributs d'interface sont par défaut public, staticet final

Une interface ne peut pas contenir de constructeur (car elle ne peut pas être utilisée pour créer des objets)

Pourquoi et quand utiliser les interfaces ?

1) Pour assurer la sécurité - masquez certains détails et affichez uniquement les détails importants d'un objet (interface).

2) Java ne supporte pas "l'héritage multiple" (une classe ne peut hériter que d'une superclasse). Cependant, cela peut être réalisé avec des interfaces, car la classe peut **implémenter** plusieurs interfaces.

Remarque : Pour implémenter plusieurs interfaces, séparez les par une virgule (voir l'exemple ci-dessous).

Interfaces multiples

```
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
```

Énumérations Java

An enum est une "classe" spéciale qui représente un groupe de **constantes** (variables non modifiables, comme finalles variables).

Pour créer un enum, utilisez le mot cléenum (au lieu de class ou interface) et séparez les constantes par une virgule. Notez qu'ils doivent être en majuscules :

Example

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}
```

Vous pouvez accéder aux enum constantes avec la syntaxe à **points :**

```
Level myVar = Level.MEDIUM;
```

Enum est l'abréviation de "énumérations", ce qui signifie "spécifiquement répertorié".

Énumération à l'intérieur d'une classe

Vous pouvez également avoir un enum intérieur d'une classe :

Example

```
public class Main {
  enum Level {
    LOW,
    MEDIUM,
    HIGH
  }

  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;
    System.out.println(myVar);
  }
}
```

Les énumérations sont souvent utilisées dans **switch** les instructions pour vérifier les valeurs correspondantes :

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}

public class Main {
  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;

    switch(myVar) {
      case LOW:
        System.out.println("Low level");
        break;
      case MEDIUM:
         System.out.println("Medium level");
        break;
      case HIGH:
        System.out.println("High level");
        break;
    }
  }
```

Boucle à travers une énumération

Le type enum a une méthode values() qui renvoie un tableau de toutes les constantes enum. Cette méthode est utile lorsque vous souhaitez parcourir les constantes d'une énumération :

Example

```
for (Level myVar : Level.values()) {
  System.out.println(myVar);
}
```

Différence entre les énumérations et les classes

Un enum peut, tout comme une class, avoir des attributs et des méthodes. La seule différence est que les constantes enum sont public, static et final (inchangeables - ne peuvent pas être remplacées).

Un enum ne peut pas être utilisé pour créer des objets, et il ne peut pas étendre d'autres classes (mais il peut implémenter des interfaces).

Pourquoi et quand utiliser les énumérations ?

Utilisez des énumérations lorsque vous avez des valeurs dont vous savez qu'elles ne changeront pas, comme les jours du mois, les jours, les couleurs, le jeu de cartes, etc.

Java User Input (Scanner)

La Scanner classe est utilisée pour obtenir l'entrée de l'utilisateur, et elle se trouve dans le package java.util.

Pour utiliser la classe **Scanner**, créez un objet de la classe et utilisez l'une des méthodes disponibles trouvées dans la documentation de la classe Scanner. Dans notre exemple, nous utiliserons la méthode nextLine(), qui sert à lire les Strings :

```
import java.util.Scanner;  // Import the Scanner class

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

Types d'entrée

Dans l'exemple ci-dessus, nous avons utilisé la méthode **nextLine()**, qui est utilisée pour lire les chaînes. Pour lire d'autres types, regardez le tableau ci-dessous

nextBoolean() : Reads a boolean value from the user

nextFloat() : Reads a float value from the user

nextLine() : Lit une valeur de chaîne de l'utilisateur

nextInt() : Lit une valeur int de l'utilisateur

Dans l'exemple ci-dessous, nous utilisons différentes méthodes pour lire des données de différents types :

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter name, age and salary:");

    // String input
    String name = myObj.nextLine();

    // Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();

    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }
}
```

Remarque :

—

Si vous entrez une mauvaise entrée (par exemple, du texte dans une entrée numérique), vous obtiendrez un message d'exception / d'erreur (comme "InputMismatchException").

Vous pouvez en savoir plus sur les exceptions et la manière de gérer les erreurs dans le chapitre Exceptions .

Date et heure Java

Java n'a pas de la classe Date intégrée, mais nous pouvons importer le package java.time pour qu'il fonctionne avec l'API de date et d'heure. Le forfait comprend de nombreuses classes de date et d'heure. Par example:

LocalDate : Représente une date (année, mois, jour (aaaa-MM-jj))

LocalTime : Représente une heure (heure, minute, seconde et nanosecondes (HH-mm-ss-ns))

LocalDateTime : Représente à la fois une date et une heure (aaaa-MM-jj-HH-mm-ss-ns)

Afficher la date actuelle

Pour afficher la date actuelle, importez la java.time.LocalDateclasse et utilisez sa méthode now() :

Example

```
import java.time.LocalDate; // import the LocalDate class

public class Main {
  public static void main(String[] args) {
    LocalDate myObj = LocalDate.now(); // Create a date object
    System.out.println(myObj); // Display the current date
  }
}
```

Java ArrayList

La classe est un tableau ArrayList redimensionnable , qui peut être trouvé dans le package.java.util

La différence entre un tableau intégré et un ArrayListen Java, est que la taille d'un tableau ne peut pas être modifiée (si vous voulez ajouter ou supprimer des éléments d'un tableau, vous devez en créer un nouveau). Alors que des éléments peuvent être ajoutés et supprimés à ArrayList tout moment. La syntaxe est également légèrement différente :

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Ajouter des articles

La classe ArrayList a de nombreuses méthodes utiles. Par exemple, pour ajouter des éléments au ArrayList, utilisez la méthode add() :

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Accéder à un article

Pour accéder à un élément du ArrayList, utilisez la méthode get() et référez-vous au numéro d'index :

Example

```
cars.get(0);
```

Modifier un élément

Pour modifier un élément, utilisez laméthode set() et reportez-vous au numéro d'index :

Example

```
cars.set(0, "Opel");
```

Supprimer un article

Pour supprimer un élément, utilisez la méthode remove() et référez-vous au numéro d'index :

Example

```
cars.remove(0);
```

Pour supprimer tous les éléments du ArrayList, utilisez la méthode clear():

Example

```
cars.clear();
```

Taille de la liste de tableaux

Pour connaître le nombre d'éléments d'une ArrayList, utilisez la méthode size :

Example

```
cars.size();
```

Boucler dans une ArrayList

Parcourez les éléments d'un ArrayList avec une boucle **for** et utilisez la méthode size() pour spécifier combien de fois la boucle doit s'exécuter :

Example

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

Vous pouvez également parcourir une boucle ArrayList avec la boucle **for-each** :

Example

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Autres types

Les éléments d'une ArrayList sont en fait des objets. Dans les exemples ci-dessus, nous avons créé des éléments (objets) de type "String". N'oubliez pas qu'une chaîne en Java est un objet (pas un type primitif). Pour utiliser d'autres types, comme **int**, vous devez spécifier une classe wrapper équivalente : Integer. Pour les autres types primitifs, utilisez : Boolean for boolean, Character **for** char, Double **for** double, etc :

Exemple

Créez un ArrayList pour stocker des nombres (ajoutez des éléments de type Integer):

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

Trier une ArrayList

Une autre classe utile dans le package java.util est la Collections classe , qui inclut la méthode sort() de tri des listes par ordre alphabétique ou numérique :

Trier une ArrayList de chaînes :

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    Collections.sort(cars);  // Sort cars
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Trier une ArrayList d'entiers :

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

Liste liée Java

Dans le chapitre précédent, vous avez découvert la classe ArrayList. La classe LinkedList est presque identique à la ArrayList:

Example

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

ArrayList Vs LinkedList

La classe LinkedList est une collection qui peut contenir plusieurs objets du même type, tout comme le ArrayList.

La classe LinkedList a toutes les mêmes méthodes que la ArrayList classe car elles implémentent toutes les deux l' List interface. Cela signifie que vous pouvez ajouter des éléments, modifier des éléments, supprimer des éléments et effacer la liste de la même manière.

Cependant, si la classe ArrayList et la classe LinkedList peuvent être utilisées de la même manière, elles sont construites très différemment.

Comment fonctionne ArrayList

—

La classe ArrayList a un tableau régulier à l'intérieur. Lorsqu'un élément est ajouté, il est placé dans le tableau. Si le tableau n'est pas assez grand, un nouveau tableau plus grand est créé pour remplacer l'ancien et l'ancien est supprimé.

Comment fonctionne la LinkedList

Le LinkedList stocke ses articles dans des "conteneurs". La liste a un lien vers le premier conteneur et chaque conteneur a un lien vers le conteneur suivant dans la liste. Pour ajouter un élément à la liste, l'élément est placé dans un nouveau conteneur et ce conteneur est lié à l'un des autres conteneurs de la liste.

Quand utiliser

Utilisez un ArrayList pour stocker et accéder aux données, et LinkedList pour manipuler les données.

Méthodes de listes liées

Dans de nombreux cas, le ArrayList est plus efficace car il est courant d'avoir besoin d'accéder à des éléments aléatoires de la liste, mais le LinkedList fournit plusieurs méthodes pour effectuer certaines opérations plus efficacement :

addFirst() : Ajoute un élément au début de la liste.

addLast() : Ajouter un élément à la fin de la liste

removeFirst() : Supprimer un élément du début de la liste.

removeLast() : Supprimer un élément de la fin de la liste

getFirst() : Obtenir l'élément au début de la liste

getLast() : Obtenir l'élément à la fin de la liste

Java HashMap

Dans ce ArrayList chapitre, vous avez appris que les tableaux stockent les éléments sous la forme d'une collection ordonnée et que vous devez y accéder avec un numéro d'index (inttype). A HashMap cependant, stockez les éléments dans des paires " **clé** / **valeur** ", et vous pourrez y accéder par un index d'un autre type (par exemple a String).

Un objet est utilisé comme clé (index) pour un autre objet (valeur). Il peut stocker différents types : String clés et Integer valeurs, ou le même type, comme : String clés et String valeurs :

Créez un HashMap objet appelé **capital Cities** qui stockera les String **clés** et String **les valeurs** :

```
import java.util.HashMap; // import the HashMap class

HashMap<String, String> capitalCities = new HashMap<String, String>();
```

Ajouter des articles

La HashMap classe a de nombreuses méthodes utiles. Par exemple, pour y ajouter des éléments, utilisez la méthode put() :

```
// Import the HashMap class
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
    System.out.println(capitalCities);
  }
}
```

Accéder à un article

Pour accéder à une valeur dans le HashMap, utilisez la méthode get() et référez-vous à sa clé :

Example

```
capitalCities.get("England");
```

Supprimer un article

Pour supprimer un élément, utilisez la méthode remove() et référez-vous à la clé :

Example

```
capitalCities.remove("England");
```

Exceptions Java - Try...Catch

Lors de l'exécution du code Java, différentes erreurs peuvent survenir : erreurs de codage faites par le programmeur, erreurs dues à une mauvaise saisie ou autres choses imprévisibles.

Lorsqu'une erreur se produit, Java s'arrête normalement et génère un message d'erreur. Le terme technique pour cela est : Java lèvera une **exception** (lancera une erreur).

Java try and catch

L'instruction try vous permet de définir un bloc de code à tester pour les erreurs pendant son exécution.

L'instruction catch vous permet de définir un bloc de code à exécuter, si une erreur se produit dans le bloc try.

Les mots-clés try et catch vont par paires :

Syntax

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Considérez l'exemple suivant :

Cela générera une erreur, car **myNumbers[10]** n'existe pas

```
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

Si une erreur se produit, nous pouvons utiliser try...catch pour intercepter l'erreur et exécuter du code pour la gérer :

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
```

Finally

L' finally instruction vous permet d'exécuter du code, après try...catch, quel que soit le résultat :

Example

```
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

The throw keyword

L' instruction throw vous permet de créer une erreur personnalisée.

L' instruction throw est utilisée avec un **type d'exception** . Il existe de nombreux types d'exceptions disponibles en Java : ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc :

Exemple

Lancer une exception si l' **âge** est inférieur à 18 ans (imprimer "Accès refusé"). Si l'âge est de 18 ans ou plus, écrivez "Accès accordé":

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

Expressions régulières Java

Une expression régulière est une séquence de caractères qui forme un modèle de recherche. Lorsque vous recherchez des données dans un texte, vous pouvez utiliser ce modèle de recherche pour décrire ce que vous recherchez.

Une expression régulière peut être un caractère unique ou un modèle plus compliqué.

Les expressions régulières peuvent être utilisées pour effectuer tous les types d' opérations de **recherche** et de **remplacement de texte** .

Java n'a pas de classe d'expression régulière intégrée, mais nous pouvons importer le java.util.regex package pour travailler avec des expressions régulières. Le forfait comprend les cours suivants :

Classe Pattern - Définit un modèle (à utiliser dans une recherche)

Classe Matcher - Utilisé pour rechercher le modèle

Classe PatternSyntaxException - Indique une erreur de syntaxe dans un modèle d'expression régulière

Exemple

Découvrez s'il y a des occurrences du mot "w3schools" dans une phrase :

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("w3schools", Pattern.CASE_INSENSITIVE);
    Matcher matcher = pattern.matcher("Visit W3Schools!");
    boolean matchFound = matcher.find();
    if(matchFound) {
      System.out.println("Match found");
    } else {
      System.out.println("Match not found");
    }
  }
}
// Outputs Match found
```

Exemple expliqué

Dans cet exemple, le mot "w3schools" est recherché dans une phrase.

Tout d'abord, le motif est créé à l'aide de la méthode Pattern.compile(). Le premier paramètre indique quel modèle est recherché et le deuxième paramètre a un indicateur pour indiquer que la recherche doit être insensible à la casse. Le deuxième paramètre est facultatif.

La méthode matcher() est utilisée pour rechercher le motif dans une chaîne. Elle renvoie un objet Matcher qui contient des informations sur la recherche qui a été effectuée.

La méthode find() renvoie vrai si le motif a été trouvé dans la chaîne et faux s'il n'a pas été trouvé.

Drapeaux

Les indicateurs dans la méthode compile() modifient la manière dont la recherche est effectuée. En voici quelques-uns :

- Pattern.CASE_INSENSITIVE- La casse des lettres sera ignorée lors d'une recherche.
- Pattern.LITERAL- Les caractères spéciaux du modèle n'auront aucune signification particulière et seront traités comme des caractères ordinaires lors de l'exécution d'une recherche.
- Pattern.UNICODE_CASE- Utilisez-le avec le CASE_INSENSITIVEdrapeau pour ignorer également la casse des lettres en dehors de l'alphabet anglais

Modèles d'expressions régulières

Le premier paramètre de la Pattern. la méthode compile() est le motif. Il décrit ce qui est recherché.

Les parenthèses sont utilisées pour trouver une plage de caractères :

[abc] : Trouvez un caractère parmi les options entre parenthèses

[^abc] : Trouver un caractère NON entre parenthèses

[0-9] : Trouver un caractère dans la plage de 0 à 9

Métacaractères

Les **métacaractères** sont des caractères ayant une signification particulière :

| : Trouvez une correspondance pour l'un des modèles séparés par | comme dans : chat|chien|poisson

. : Trouver une seule instance de n'importe quel personnage

^ : Trouve une correspondance comme début d'une chaîne comme dans : ^Hello

$: Trouve une correspondance à la fin de la chaîne comme dans : World$

\d : Trouver un chiffre

\s : Trouver un caractère d'espacement

Quantificateurs

Les quantificateurs définissent les quantités :

n+ : Correspond à toute chaîne contenant au moins un n.

n* : Correspond à toute chaîne contenant zéro ou plusieurs occurrences de n

n? : Correspond à toute chaîne contenant zéro ou une occurrence de n

n{x} : Correspond à toute chaîne contenant une séquence de X n

n{x,y} : Correspond à toute chaîne contenant une séquence de X à Y n

n{x,} : Correspond à toute chaîne contenant une séquence d'au moins X n

Fils Java

Les threads permettent à un programme de fonctionner plus efficacement en faisant plusieurs choses en même temps.

Les threads peuvent être utilisés pour effectuer des tâches complexes en arrière-plan sans interrompre le programme principal.

Création d'un fil

Il existe deux façons de créer un fil.

Il peut être créé en étendant la classe Thread et en remplaçant sa méthode run() :

```
public class Main extends Thread {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Une autre façon de créer un thread consiste à implémenter l' Runnableinterface :

```
public class Main implements Runnable {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Exécution de threads

Si la classe étend la classe Thread, le thread peut être exécuté en créant une instance de la classe et en appelant sa méthode start() :

```
public class Main extends Thread {
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Si la classe implémente l' Runnable interface, le thread peut être exécuté en passant une instance de la classe au Thread constructeur d'un objet, puis en appelant la méthode start() du thread :

```
public class Main implements Runnable {
  public static void main(String[] args) {
    Main obj = new Main();
    Thread thread = new Thread(obj);
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Différences entre "étendre" et "implémenter" les threads

La principale différence est que lorsqu'une classe étend la classe Thread, vous ne pouvez étendre aucune autre classe, mais en implémentant l'interface Runnable, il est également possible d'étendre une autre classe, comme : class MyClass extends OtherClass implements Runnable.

Problèmes de simultanéité

Étant donné que les threads s'exécutent en même temps que d'autres parties du programme, il n'y a aucun moyen de savoir dans quel ordre le code s'exécutera. Lorsque les threads et le programme principal lisent et écrivent les mêmes variables, les valeurs sont imprévisibles. Les problèmes qui en résultent sont appelés problèmes de concurrence.

Exemple

Un exemple de code où la valeur du **montant** variable est imprévisible :

```
public class Main extends Thread {
  public static int amount = 0;

  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println(amount);
    amount++;
    System.out.println(amount);
  }

  public void run() {
    amount++;
  }
}
```

Pour éviter les problèmes de concurrence, il est préférable de partager le moins d'attributs possible entre les threads. Si les attributs doivent être partagés, une solution possible consiste à utiliser la isAlive() méthode du thread pour vérifier si le thread a fini de s'exécuter avant d'utiliser les attributs que le thread peut modifier.

Exemple

Utilisez isAlive()pour éviter les problèmes de simultanéité :

```
public class Main extends Thread {
  public static int amount = 0;

  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    // Wait for the thread to finish
    while(thread.isAlive()) {
    System.out.println("Waiting...");
  }
  // Update amount and print its value
  System.out.println("Main: " + amount);
  amount++;
  System.out.println("Main: " + amount);
  }
  public void run() {
    amount++;
  }
```

Expressions Java Lambda

Les expressions Lambda ont été ajoutées dans Java 8.

Une expression lambda est un court bloc de code qui prend des paramètres et renvoie une valeur. Les expressions lambda sont similaires aux méthodes, mais elles n'ont pas besoin de nom et peuvent être implémentées directement dans le corps d'une méthode.

Syntaxe

L'expression lambda la plus simple contient un seul paramètre et une expression :

```
parameter -> expression
```

Pour utiliser plusieurs paramètres, placez-les entre parenthèses :

```
(parameter1, parameter2) -> expression
```

Les expressions sont limitées. Ils doivent renvoyer immédiatement une valeur et ne peuvent pas contenir de variables, d'affectations ou d'instructions telles que ifou for. Afin d'effectuer des opérations plus complexes, un bloc de code peut être utilisé avec des accolades. Si l'expression lambda doit renvoyer une valeur, le bloc de code doit contenir une return instruction.

```
(parameter1, parameter2) -> { code block }
```

Utilisation d'expressions Lambda

Les expressions lambda sont généralement transmises en tant que paramètres à une fonction :

Utilisez une expression lamba dans la ArrayList méthode forEach() de pour imprimer chaque élément de la liste :

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

Les expressions lambda peuvent être stockées dans des variables si le type de la variable est une interface qui n'a qu'une seule méthode. L'expression lambda doit avoir le même nombre de paramètres et le même type de retour que cette méthode. Java intègre plusieurs de ces types d'interfaces, telles que l' Consumer interface (qui se trouve dans le java.utilpackage) utilisée par les listes.

Utilisez l'interface de Java Consumer pour stocker une expression lambda dans une variable :

```java
import java.util.ArrayList;
import java.util.function.Consumer;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    Consumer<Integer> method = (n) -> { System.out.println(n); };
    numbers.forEach( method );
  }
}
```

Pour utiliser une expression lambda dans une méthode, la méthode doit avoir un paramètre avec une interface à méthode unique comme type. L'appel de la méthode de l'interface exécutera l'expression lambda :

```
interface StringFunction {
  String run(String str);
}

public class Main {
  public static void main(String[] args) {
    StringFunction exclaim = (s) -> s + "!";
    StringFunction ask = (s) -> s + "?";
    printFormatted("Hello", exclaim);
    printFormatted("Hello", ask);
  }
  public static void printFormatted(String str, StringFunction format) {
    String result = format.run(str);
    System.out.println(result);
  }
}
```

Java Files

La gestion des fichiers est une partie importante de toute application.

Java propose plusieurs méthodes pour créer, lire, mettre à jour et supprimer des fichiers.

La classe File du java.iopackage nous permet de travailler avec des fichiers.

Pour utiliser la classe File, créez un objet de la classe et spécifiez le nom du fichier ou le nom du répertoire :

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

La Fileclasse dispose de nombreuses méthodes utiles pour créer et obtenir des informations sur les fichiers. Par example:

canRead() : Boolean,Teste si le fichier est lisible ou non.

canWrite() : Boolean, Teste si le fichier est inscriptible ou non.

createNewFile() : Boolean, Creates an empty file

delete() : Boolean, Supprime un fichier.

exists() : Boolean , Teste si le fichier existe

getName() : String, Renvoie le nom du fichier.

getAbsolutePath() : String , Renvoie le chemin d'accès absolu du fichier.

length() : Long, Renvoie la taille du fichier en octets.

list() : String[], Renvoie un tableau des fichiers du répertoire.
mkdir() : Boolean, Crée un répertoire

Créer un fichier

Pour créer un fichier en Java, vous pouvez utiliser la méthode createNewFile(). Cette méthode renvoie une valeur booléenne : true si le fichier a été créé avec succès et false si le fichier existe déjà. Notez que la méthode est enfermée dans un try...catch bloc. Ceci est nécessaire car il lance un IOException si une erreur se produit (si le fichier ne peut pas être créé pour une raison quelconque) :

Exemple

```
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```