



Cursus SALESFORCE

M2I Formations 2021

Christian Lisangola

MODULE JAVA

Les fondamentaux du Langage Java

1. Préparation de l'environnement

Installer les outils nécessaires



Pour cette formation nous allons utiliser comme EDI : **IntelliJ de JetBrains** dont les instructions d'installation se trouve ici :

<https://www.jetbrains.com/help/idea/installation-guide.html>

Les participant peuvent aussi utiliser : <https://replit.com/~>

2.

Contenu

MODULE JAVA

1. Bases de la programmation en JAVA

Introduction

1.1.



Travail en équipe



You devez faire quelques recherches sur le langage Java, avec un focus sur les points suivant:

- Particularités du langage
- JVM
- Bytecode
- Que veut dire “Write once, runs everywhere”
- Positionnement sur le marché

Durée : 30 minutes

Premier programme

Pour écrire un programme Java, il nous faut un environnement de développement.Dans cette formations nous utiliserons IntelliJ.

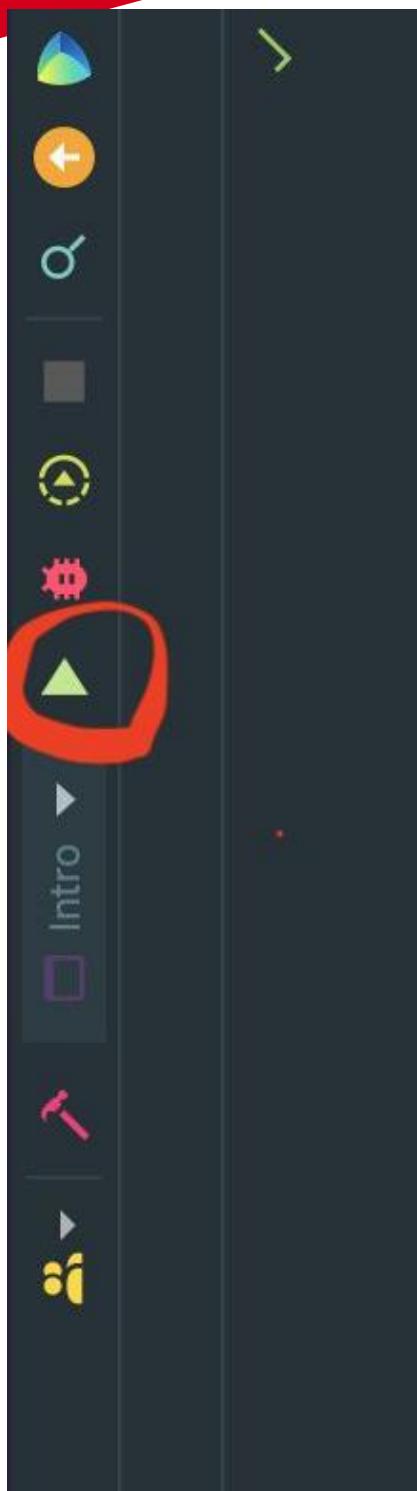
Voici notre premier programme qui affiche “Hello, world !”.

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

Compilation et Execution



Avant d'exécuter ce programme, il faut le compiler. IntelliJ nous permet de compiler et exécuter le code en appuyant sur le bouton vert au coin supérieur droit.



Compilation et Execution



Une autre manière de compiler et exécuter est de faire un clique droit dans le fichier et cliquer sur “Run”.

```
package Introduction;

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

The screenshot shows an IDE's context menu for a Java file named "Intro.java". The menu includes options like "Show Context Actions", "Paste", "Copy / Paste Special", "Column Selection Mode", "Refactor", "Folding", "Analyze", "Go To", "Generate...", and "Run 'Intro.main()'" which is highlighted with a red arrow. Other visible items include "Debug 'Intro.main()'", "Run 'Intro.main()'" with Coverage, "Modify Run Configuration...", "Open In", "Local History", "Compare with Clipboard", "Create Gist...", and "Add BOM".

En cas d'erreur



En cas d'erreur, le programme ne va pas être compilé et va afficher un message avec des indications sur la raison et la ligne de code où elle est survenue. La meilleure façon de corriger vos erreurs est de le faire dans leur ordre d'apparition dans le fichier. En effet, une erreur peut souvent entraîner d'autres plus loin dans le fichier.

```
✗ ② 1-premier-pas: build failed At 10/16/21 sec, 149 ms  /Users/christianlisangola/Documents/Projects/formations/noe-salesforce/java-prep/src/Introduction/Intro.java:5:45
    > `;` expected
      |
      ✗ ;` expected :5
      |
```

Variables

1.2.

Déclaration des variables

Voici la syntaxe de déclaration des variables en JAVA :

type identificateur;

type identificateur=valeur_initiale;

Ici, l'**initialisation** est un moyen de fournir une valeur initial à une variable au traverse de l'affectation, ce qui est particulièrement important en Java, car si on essay par exemple d'afficher le contenu d'une variable non initialisé, le **compilateur Java donnera une erreur.**

Déclaration des variables

```
int age=44;  
String nom="Lisangola", prenom="Christian";  
double poids=80.1;
```

Nom des variables

Il existe un certain nombre de règles à respecter concernant le choix du nom des variables :

- le nom peut être constitué uniquement de **lettres**, de **chiffres** et des **deux symboles** « _ » et « \$ » ;
- le premier caractère est nécessairement une **lettre** ou un **symbole** ;
- le nom ne doit pas être un **mot-clé réservé par le langage Java** (par exemple « **if** »);
- les **majuscules** et les **minuscules** sont autorisées mais ne sont pas équivalentes ; les noms **age** et **Age** désignent deux variables différentes en Java.

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

Types de données



Les trois types élémentaires fondamentaux en Java sont :

- int**, pour les valeurs entières (integer en anglais) ;
- double**, pour les nombres à virgule ;
- char**, pour les caractères;
- boolean**, pour les caractères;

1.3.

Variables :
Lecture et
écriture

Variables : Lecture/Ecriture

```
int nombre1=45;
int nombre2=50;
int somme=nombre1+nombre2;
double rapport=(double)nombre1/(double)nombre2;
System.out.printf("%d + %d = %d\n", nombre1, nombre2, somme);
System.out.printf("%d / %d = %f\n", nombre1, nombre2, rapport);
```

Variables : Lecture/Ecriture

Lecture à partir du clavier.

- keyb.nextInt() : Pour lire des entiers
- keyb.nextDouble() : Pour lire des réels
- keyb.nextLine() : Pour lire chaque caractère saisi par l'utilisateur
- keyb.next().charAt(0) : Pour lire un caractère
- keyb.nextBoolean() : Pour lire un booléen

```
var keyb=new Scanner(System.in);  
System.out.print("Votre nom : ");  
String nom=keyb.nextLine();  
System.out.print("Votre genre : ");|  
char genre=keyb.next().charAt(0);  
System.out.print("Votre age : ");  
int age=keyb.nextInt();  
System.out.print("Votre poids : ");  
double poids=keyb.nextDouble();  
System.out.print("Etes vous mariés : ");  
boolean estMarie=keyb.nextBoolean();  
System.out.printf("Nom : %s\nAge : %d\nGenre : %c\nPoids : %f\nEtat marital : %s\n", nom, age, genre, poids, estMarie?"Marié":"Célibataire");
```

Quelques fonction de l'API JAVA

L'API JAVA fourni un grand nombre de fonctions prêts à l'emploi, entre autres des fonctions mathématiques:

- **Math.pow(constante,puissance)** : Retourne la puissance d'un nombre
- **Math.sqrt(constant)** : Retourne la racine carré d'un nombre
- **Math.sin(constant)** : Retourne le sinus d'un nombre

Pour plus de méthodes :

<https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

MODULE JAVA

2. Structures de contrôle(1) : Branchements conditionnels

2.1.

Branchements conditionnels

Branchements conditionnels

Exemple de branchement conditionnel:

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

System.out.print("Votre age : ");

int age=keyb.nextInt();

if(age>=18){

    System.out.print("Vous êtes majeur");

} else{

    System.out.print("Vous êtes mineur");

}
```

Conditions

2.2.

Opérateurs de comparaison



Dans la leçon précédente, nous avons comparé des valeurs dans le cadre des branchements conditionnels.

Ceci nous permettait de formuler des conditions simples, c'est-à-dire des conditions qui comparent deux expressions grâce à un opérateur de comparaison. Les opérateurs de comparaison du langage Java sont :

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

Opérateurs de comparaison

```
int a=1,b=2;
if(a==b){
    System.out.println("a et b sont égaux");
} else {
    System.out.println("a et b sont différents");
}
```

**Comparaison
des valeurs**

```
int a=1,b=2;
if(a>b*4){
    System.out.println("a et b sont égaux");
} else{
    System.out.println("a et b sont différents");
}
```

**Comparaison
des expressions**

Opérateurs logiques



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

- **L'opérateur & (ET)** permet de vérifier si deux conditions sont simultanément vérifiées..
- **L'opérateur || (OU)** retourne true lorsqu'au moins l'une des deux conditions qu'il évalue est vraie;
- **L'opérateur ! (NON)** retourne la négation de la valeur de la condition à sa droite.

Opérateurs logiques et le type booléen

Voici les exemples d'utilisation:

```
int age=14;
if(age ≥ 10 && age ≤ 19){
    System.out.println("Vous êtes ado");
} else{
    System.out.println("Vous n'êtes pas ado");
}
```

ET(&&)

```
int nombre=9;
if(nombre ≥ 10 || nombre ≤ 19){
    System.out.println("Correct");
} else{
    System.out.println("Incorrect");
}
```

OU(||)

Le Type Booléen



```
int a=1, b=2;
boolean c=true;
boolean d=(a==b);
boolean e=(d||(a<b));
if(e){
    System.out.println("e vaut true e");
}
```

MODULE JAVA

3. Structures de contrôle(2) : Boucles ou structures itératives

Boucle “for”

3.1.



Boucle “for”



Une **boucle for** est une itération qui permet de répéter un nombre donné de fois le même bloc d'instructions.

```
for(int i=1;i <= 5;i++){  
    System.out.printf("Le carré de %d est %f\n",i,Math.pow(i,2));  
}
```

La boucle for est composé de :

- **mot-clé for suivi de la déclaration et de l'initialisation d'une variable** qui s'effectuent **une fois** avant d'entrer dans le corps de la boucle;
- **Une condition détermine si la boucle doit continuer à exécuter** son bloc d'instructions. Lorsque sa valeur avant l'exécution est fausse, on sort de la boucle;
- **Finallement une incrémentation est effectuée à la fin de chaque tour de boucle** pour changer la valeur du compteur de boucle.

Boucle “for” imbriquées

Rien ne nous empêche de placer une boucle for dans le corps d'une autre boucle for.

Voici une boucle imbriquée permettant d'afficher les tables de multiplication de 2 jusqu'à 10.

```
for(int i=2;i<=10;i++){  
    System.out.printf("\nTable de multiplication par %d\n-----\n",i);  
    for(int j=1;j<=10;j++){  
        System.out.printf("%d x %d = %d\n",j,i,j*i);  
    }  
}
```

3.2.

Boucle conditionnelles

Boucles do{...}while()



Voici la syntaxe de la boucle do...while

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

Remarquons que **si la condition ne devient jamais fausse, les instructions de la boucle seront répétées indéfiniment.** De plus, cette condition est évaluée à posteriori, c'est-à-dire après une première exécution du corps. Il existe une variante de la boucle **do...while** où la condition est évaluée a priori, c'est la boucle **while**.

Boucles while(){...}

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

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

Le principe est similaire . celui de la boucle **do...while** . la différence que la condition est testée avant d'entrer dans la boucle.

Si la condition est fausse, les instructions dans la boucle ne sont donc pas exécutées.

MODULE JAVA

4. Les tableaux

4.1.

Introduction

Types de tableaux

En Java, on dispose principalement de deux types :

- Si la taille varie au cours du temps alors nous utiliserons le type prédefini `ArrayList`;
- Si la taille du tableau ne varie pas,nous utiliserons des **tableaux de taille fixe**, que nous allons voir dans les prochaines leçons.

4.2.

Tableaux de taille fixe

Déclaration



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

int[] scores;

Ou

Int scores[];

Initialisation



Il existe deux façons en Java d'initialiser les éléments d'un tableau.

- La première est d'initialiser le tableau avec des éléments donnés au moment de sa déclaration

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

- La seconde manière est d'initialiser un tableau sans élément, puis de le remplir ailleurs dans le programme.

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

Remplissage



Attention toutefois, le premier élément d'un tableau est situé à l'index 0, et non 1 !

Ainsi le dernier élément d'un tableau de taille T est à l'index T-1.

De plus, si l'on essaye d'accéder à un élément qui n'est pas dans le tableau, une **exception est lancée**, ce qui se traduit pour nous par l'arrêt du programme.

4.3.

**Traitements
courants**

Affectations



scores[index]=valeur

Ex : scores[1]=23

```
Scanner keyb=new Scanner(System.in);
System.out.print("Entrez la taille du tableau : ");
int taille=keyb.nextInt();
int[] scores=new int[taille];
for(int i=0;i<scores.length;i++){
    if(i==0){
        System.out.printf("Entrez le %der score : ",(i+1));
    }else{
        System.out.printf("Entrez le %dème score : ",(i+1));
    }
    scores[i]=keyb.nextInt();
}
```

Comparaisons

4.4.

Affichage : autre méthode

Pour des types évolués, l'opérateur `a == b` teste si les variables `a` et `b` référencent le même emplacement mémoire.

Avec l'affectation `a = b`, dans le cas des types évolués, on affecte à `a` la référence de `b`.

Pour vérifier l'égalité de contenu des tableaux, il faut écrire explicitement les tests.

4.5.

**Tableaux à
plusieurs
dimensions**

Ecriture



```
int[][] tab= {  
    {1(0,0), 2(0,1) ,6(0,2), 5(0,3) },  
    {2(1,0), 4(1,1) ,9(1,2) },  
    {7(2,0), 3(2,1) }  
};
```

Exemple :

- Remplacer 6 par 11 : tab[0][2] = 11
- Remplacer 3 par 23 : tab[2][1] = 23
- Remplacer 9 par 45 : tab[1][2] = 45

MODULE JAVA

5. Les tableaux dynamiques et chaînes de caractères

5.1.

Strings

Comparaisons

5.2.

Affectation et comparaison

```

String a="Christian";
String b="Christian";
if(a==b){
    System.out.println("a et b pointent vers le même littéral");
} else{
    System.out.println("a et b ne pointent pas vers le même littéral");
}

```

a==b

```

Scanner keyb=new Scanner(System.in);
String a="Christian";
String b=keyb.nextLine();
if(a==b){
    System.out.println("a et b pointent vers le même littéral");
} else{
    System.out.println("a et b ne pointent pas vers le même littéral");
}

```

a==b

```

Scanner keyb=new Scanner(System.in);
String a="Christian";
String b=keyb.nextLine();
if(a.equals(b)){
    System.out.println("a et b pointent vers le même littéral");
} else{
    System.out.println("a et b ne pointent pas vers le même littéral");
}

```

a.equals(b)

Traitement String

5.3.

Affectation et comparaison

Il existe de nombreux traitements que nous pouvons appliquer aux objets de type String. Ces traitements s'appellent des **méthodes** (ou fonctions) en Java.

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

Autres traitements

- ❑ L'instruction **chaine.replace(char1, char2)** construit une nouvelle chaîne en remplaçant dans toute la chaîne **char1** par **char2**.

Exemple :

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

Ce code construit la nouvelle chaîne **"*b*r*c*d*b*r*"**.

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

Exemple :

```
String exemple = " anticonstitutionnel" ;  
String avecDesEtoiles = exemple.substring(4, 16) ;  
Ce code construit la nouvelle chaîne "constitution".
```

5.4.

Tableaux Dynamiques

Tableaux dynamiques



Nous avons vu en détail les tableaux de taille fixe dans les leçons précédentes.

Nous allons maintenant étudier les tableaux dynamiques.

Les tableaux dynamiques ont la particularité de pouvoir changer de taille pendant l'exécution du programme.

Déclaration et Initialisation



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

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

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

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

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

Méthodes courantes

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

Voici les principales :

- tableau.size()** renvoie la taille du tableau (de type int) ;
- tableau.get(i)** renvoie l'élément à l'indice i dans le tableau (i doit être un entier compris entre 0 et tableau.size()-1) ;
- tableau.add(valeur)** ajoute valeur à la fin de tableau ;

MODULE JAVA

6. Fonctions (Méthodes en Java)

6.1.

Introduction

Méthodes courantes



Les fonctions en programmation **sont des traitements, qui agissent sur des données.**

Dans un **langage orienté objet** comme le Java, on utilisera l'appellation **méthode** plutôt que **fonction**.

Les méthodes sont des portions de code que l'on définit à un endroit du programme et que l'on peut appeler depuis n'importe quel autre endroit de notre code.

Instruction “return”



L'instruction return fait deux choses :

- Elle précise la valeur qui sera fournie par la méthode en résultat.
- Elle met fin à l'exécution des instructions de la méthode.

Méthode “main”



Nous utilisons, dans tous nos programmes depuis le début, une méthode spéciale : la méthode **main**.

Par convention, tout programme Java doit avoir une méthode **main**. Cette **méthode est la première méthode qui est appelée dans notre programme**.

L’entête autorisée pour la méthode main est la suivante :

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

6.2. Appel/Invocation

Appel de la méthode

```
public static int calculerSomme(int nombre1,int nombre2){  
    return nombre1 + nombre2;  
}  
  
public static void main(String[] args) {  
    Scanner keyb=new Scanner(System.in);  
    System.out.print("Premier nombre : ");  
    int nombre1=keyb.nextInt();  
    System.out.print("Deuxième nombre : ");  
    int nombre2=keyb.nextInt();  
    int somme=calculerSomme(nombre1,nombre2);  
    System.out.printf("%d + %d = %d",nombre1,nombre2,somme);  
}
```

6.3.

Passage des arguments

Passage des arguments

Nous avons vu comment est évalué l'appel d'une méthode lorsque les arguments passés à celle-ci sont des valeurs ou des expressions. En programmation, de façon générale, on dira que :

- Un **argument est passé par valeur** si la méthode ne peut pas le modifier : la méthode crée une copie locale de cet argument.
- Un **argument est passé par référence** si la méthode peut le modifier.

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

Arguments de Type Simple

Si un argument passé à une méthode est de type simple, la méthode n'aura pas de conséquences sur sa valeur en dehors de la méthode puisque le passage par valeur crée une copie locale de cet argument.

```
public static void eleverAuCarre(int nombre) {  
    nombre=nombre*nombre;  
    System.out.println("Dans la fonction : "+nombre);  
}  
  
public static void main(String[] args) {  
    int nombre=9;  
    eleverAuCarre(nombre);  
    System.out.println("En dehors de la fonction : "+nombre);  
}
```

Arguments de Type Evolué



Pour des arguments de type évolué, qui sont pour rappel manipulés via des références, c'est la référence qui est passée à la méthode. Comme en Java les arguments sont passés par valeur, la référence de l'objet passée à la méthode est copiée.

Ainsi, si la référence est modifiée dans la méthode, cela n'aura pas d'influence sur la variable en dehors de la méthode.

```
public static void ajouterFruit(String valeur, ArrayList<String> fruits){  
    fruits.add(valeur);  
  
}  
  
public static void main(String[] args) {  
    ArrayList<String> fruits=new ArrayList<String>();  
    fruits.add("pommes");  
    fruits.add("mangues");  
    System.out.println("En dehors de la fonction avant appel : "+fruits);  
    ajouterFruit( valeur: "Raisins" ,fruits);  
    System.out.println("En dehors de la fonction après appel : "+fruits);  
}
```

Arguments de Type Evolué



Elle est constituée de son nom, de ses paramètres et de son type de retour. La syntaxe est la suivante :

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

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

Exemples :

```
static int caculerSomme(int nombre1, int nombre2);
```

Dans le cas d'une méthode sans valeur de retour, on utilisera le **type spécial void comme type de retour**.

Surcharge

6.4.

Surcharge



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

Surcharge

```
public static int somme(int a,int b){  
    return a+b;  
}  
  
public static double somme(double a,double b){  
    return a+b;  
}  
  
}
```

```
public static double somme(double a){  
    return a+a;  
}  
  
}
```

TP 29



On vous donne comme argument un tableau contenant des chaînes de directions (haut, bas, gauche, droite). Imaginez une personne debout sur une grille au point 0, 0. Pour chaque direction dans le tableau de chaînes, déplacez votre personne dans cette direction sur la grille. Retournez le point final X,Y où se trouve la personne sous la forme d'un tableau de deux entiers.

Exigences

Doit retourner un tableau de deux entiers

Exemple:

```
maMethode(["haut", "haut", "bas", "gauche", "gauche", "droite", "haut"])
```

> [-1, 2]

Exercices



<https://docs.google.com/document/d/1k38ZQ0KSxQSrdA603q1sYjGJicq-A47wOgZ04YhgKx4/edit?usp=sharing>

THE END.