JAVA IMPÉRATIF

INTRODUCTION

OBJECTIFS DE CETTE SESSION

Comment utiliser Java comme un langage impératif ?

- **Pas de notion d'objet**
- Être capable de développer une application faisant appel à une série d'instructions évaluées les unes après les autres
- Être capable d'utiliser des **fonctions** issues d'une bibliothèque externe
- Être capable d'écrire et d'utiliser ses propres **fonctions**

POINTS ABORDÉS DANS CETTE SESSION

- Les impératifs dans un fichier Java
- Création de fonctions
- Introduction aux différents **types** en Java
- Introduction aux opérations de base
- Introduction aux instructions de **flow control**
- Introduction à la fonction main

CRÉER UN FICHIER JAVA

RETOUR SUR LE “HELLO WORLD”

Fichier **App.java**:

```java
import java.lang.System;
public class App {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

- Premières lignes : les importations de bibliothèques
- Correspondance `public class XXX` / nom de fichier

DÉFINITION D'UNE FONCTION

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

- `public static` : élément à insérer avant la déclaration d'une fonction
- `void` : type de retour
- `main` : nom de la fonction
- `String[] args` : type et nom des arguments

```
public static TYPE_RETOUR nom(TYPE1 arg1, ...){
  Instruction1 ;
  Instruction2 ;
  ...
}
```

- Corps de la fonction entre les accolades
- Une fonction ne peut retourner qu'un seul objet
- ou rien (type `void`)

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

- `System.out.println` : instruction pour afficher quelque chose dans le terminal
- `"Hello, World!"` : argument passé à la fonction `println` (ici une chaîne de caractères)

QUELQUES RÈGLES SUR LES INSTRUCTIONS :

```
Instruction ; // Ceci est un commentaire
/*
Ceci est un bloc de commentaires
Qui peut s'étendre sur plusieurs lignes
*/
Instruction_suivante ;
```

- Se termine toujours par un point-virgule
- Les commentaires sont avec les symboles / /

LES TYPES JAVA

TYPES DE BASE

Java est un langage fortement typé.

8 types de base :

- **byte, short, int, long**
- **float, double**
- **boolean**
- **char**

LES DIFFÉRENTS ENTIERS

- **byte** : entier signé sur 8 bits $[-128, 127]$
- **short** : entier signé sur 16 bits $[-32768, 32767]$
- **int** : entier signé sur 32 bits $[-2^{31}, 2^{31}-1]$
- **long** : entier signé sur 64 bits $[-2^{63}, 2^{63}-1]$

LES ENTIERS LITTÉRAUX

- Par défaut des **int** (ex. : 1, 2, 3,…)
- **byte**, **short**, **int** et **long** peuvent être initialisés avec un entier littéral
- Les **long** avec des valeurs supérieures à 2^{31} peuvent être initialisés avec des littéraux finissant par “L” (ex. : 1000000000L)
- Possibilité d’ajouter des underscores "_" pour la lisibilité (ex. : 1_100)

LES DIFFÉRENTES BASES D'ENTIERS

```
// Le nombre 26 en décimal
int decVal = 26;
//  Le nombre 26, en hexadécimal
int hexVal = 0x1a;
// Le nombre 26, en binaire
int binVal = 0b11010;
```

LES NOMBRES FLOTTANTS

- **float** : Nombres flottants simple précision codés sur 32 bits
- **double** : Nombres flottants double précision codés sur 64 bits.

Exemples :

```
double a = 1 ;
double b = 1.0 ;
double c = 1.3e3 ;
```

ATTENTION AUX EXPRESSIONS LITTÉRALES

```
double a = 3 ;
double b = 2 ;
double c = a/b ;

double d = 3/2 ; // --> Renvoie 1 !!!
```

LES BOOLÉENS

- **boolean** : ne peut valoir que **true** or **false**
- À utiliser dans les tests logiques
- “*Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn’t something that’s precisely defined.*”

LES CARACTÈRES

- **char** : caractère unicode codé sur 16 bits
- Va de *u0000* à *uFFFF*

Exemple :

```
char a = 0 ;
char capitalC = 'C' ;
(int)capitalC ; // Conversion de capitalC en en
```

ÉCHAPPER UN SYMBOLE

- Peut-on utiliser les caractères " et ' ?
- Les *échapper* avec un backslash \
- Un caractère utile : le retour à la ligne \n

```
char guillemet = '\"' ;
char apostrophe = '\'';
char backslash = '\\' ;
char retourLigne = '\n';
```

LES TABLEAUX

Les tableaux (**array**) permettent de stocker un nombre **connu** d'objets en mémoire.

Ils peuvent être de n'importe quel type.

- `int[]`
- `double[]`
- `char[]`
- `...`

Déclaration :

```
int[] tableau ; // Déclaration d'un tableau d'enti
```

Initialisation avec le mot-clé new

```
tableau = new int[N] ; // N est un entier
```

Accès : avec l'opérateur []

```
    tableau[i] ; // i < N
```

Une autre façon d'initialiser un tableau lorsqu'on connaît les éléments :

```
int[] tableau2= {3,4,5,6} ;
```

Les tableaux peuvent être modifiés directement :

```
tableau2[0] = 0 ; // tableau2 vaut donc {0,4,5,6}
```

Comment connaître la taille d’un tableau ?

- Rappel : le type d’un tableau d’entier est int[] : la taille n’est pas mentionnée.
- **.length** après le nom du tableau

```
int N = tableau.length ;
```

Il est possible de créer des tableaux sur plusieurs dimensions. Il s'agit alors d'un tableau de tableau.

```
int[][] tab1 = {{1 , 2 , 3},{4 , 5 , 6}} ;
int[][] tab2 = {{1,2,3}, {4,5}}; // On a touj
```

LA CLASSE STRING

La classe **String** permet de manipuler les chaînes de caractère.

```
String chaine = "Hello World"; // Déclaration

String chaineSurPlusieurLignes =
"""
  "Hello" avec des guillemets !!!
  World
""";
```

Deux opérations utiles…

- **length**() : donne la longueur de la chaîne

```
String s1 = "Hello"
int N = s1.length(); // Une syntaxe un peu ét
```

- Opérateur **+** : permet de concaténer deux chaînes

```
String s2 = " World";
System.out.print(s1+s2); // Affiche Hello World
```

LES INSTRUCTIONS DE BASE EN JAVA

DÉCLARATION / AFFECTATION

- Déclaration : fournir le nom du type et de la variable

```
int  i ;
```

- Affectation : avec le signe =

```
i = 1 ;
```

- Combinaison déclaration/affectation :

```
int j = 2 ;
```

OPÉRATIONS MATHÉMATIQUES

- Opérations mathématiques de base **+** , **-** , ***** , **/** (avec les priorités mathématiques habituelles)
- Opérations d'incrémentation/décrémentation **++**/**--**

```
i ++ ; // Équivalent à i = i + 1 ;
i -- ; // Équivalent à i = i-1 ;
i += 3 ; // Équivalent à i = i + 3 ;
i *= 3 ; // Équivalent à i = i*3 ;
```

Reste de la division entière %

```
int i = 11 % 3 ; // i vaut 2
```

TESTS ET LOGIQUE BOOLÉENNE

- Test d'égalité == ou de non égalité !=

```
if (i==3){
  // Instructions si i vaut 3
}else{
  // Instructions sinon
}
```

- Tests comparatifs <=, <, >= et >

```
if (i>=3) ...
```

- Opération de négation !

```
boolean b = !true ; // b = false donc...
boolean b2 = i!=3 ;
//b2 = true ou false en fonction du test
```

- Le ET et OU logique : **&&** et ||

```
boolean b = true && false ; // false !
boolean b = true || false ; // true !
```

OPÉRATEUR TERNAIRE " ? "

Si la condition vaut **true**, alors on retourne val1, sinon on retourne val2.

```
condition ? val1 : val2 ;
```

Exemple

```
int note = 15 ;
char grade = (note >= 16) ? 'A':'B';
```

EXEMPLE

Écrire en **une instruction** une fonction qui prend en entrée une note entre 0 et 20 et qui renvoie la lettre associée (**char**) en fonction de la répartition suivante :

- [20 , 15] : A]15 , 10] : B
-]10 , 5] : C]5, 0] : D

```
int n = 7 ; // La note en chiffre
char l = n>=15 ? 'A':n>=10 ? 'B':n>= 5 ? 'C':'D';
```

LE CONTRÔLE DE FLUX EN JAVA

Il s’agit ici de définir l’ordre d’exécution des instructions.

Par défaut, au sein d’une méthode, les instructions sont exécutées **les une après les autres**.

LE MOT-CLÉ RETURN

Le mot-clé **return** permet d'interrompre définitivement l'exécution d'une méthode et de retourner la valeur précisée après le mot clé.

```
int renvoie1(){
  return 1;
}
```

Ce mot clé est obligatoire pour les fonctions qui retournent autre chose que *void*

LES INSTRUCTIONS IF/ELSE

```
instruct1 ;
if (x == 4)
{
  instruct2 ;
}else
{
  instruct3 ;
}
instruct4 ;
```

- Si $x = 4$, on aura instruct1 → instruct2→ instruct4
- Si $x \neq 4$, on aura instruct1 → instruct3 → instruct4

On peut également avoir **if** sans **else**

```
if(x==4){
  instruct1 ;
}
instruct2 ;
```

- Si $x = 4$, on aura instruct1 → instruct2
- Si $x \neq 4$, on aura instruct2

LES BOUCLES FOR

```
for (initialisation ; conditionFin ; increment){
  instructions;
}
instructionsSuivante ;
```

- Réaliser des opérations un nombre défini de fois
- Parcourir un tableau / une liste
- Une fois la condition de fin réalisée, **instructionsSuivante** est exécutée

```java
for (int i=0 ; i < = 10 ; i++){
  System.out.print("On affiche le nombre ");
  System.out.println(i);
}
System.out.println("On a compté jusqu'à 10")
```

```java
for (int i=10 ; i > = 0 ; i--){
  System.out.print("On affiche le nombre ");
  System.out.println(i);
}
System.out.println("Fin du compte à rebours !")
```

for comme en Python (*foreach*)

```
int[] tab = {2,3,1,2}
for (int e:tab){
  System.out.println(e);
}
```

Attention : le contenu de la boucle ne doit pas modifier le tableau

LES BOUCLES WHILE

```
while (expressionTest) {
  instructions;
}
instructionsSuivantes ;
```

- Réaliser des opérations tant qu'une condition est réalisée
- Si **expressionTest** vaut **false** lors de sa première évaluation, on passe à **instructionsSuivantes**
- **while(true)** → boucle infinie
- Une fois que **expressionTest** est faux, **instructionsSuivantes** est exécutée

LES BOUCLES DO WHILE

```
do{
  instructions;
}while (expressionTest);
instructionsSuivantes ;
```

- Similaire à **while**
- Mais garantit que le bloc **instructions** est exécuté **au moins une fois**.
- Une fois que **expressionTest** est faux, **instructionsSuivantes** est exécutée

LE MOT CLÉ BREAK

Permet de sortir d'un bloc d'instruction **for**, **while** ou **do while** prématurément

```
String chaine = "Hello World" ;
// Recherche de la présence du caractère 'W'
boolean wPresent = false ;
for (int i = 0 ; i< chaine.length() ; i++){
  if (chaine.charAt(i) == 'W'){
    wPresent = true ;
    break ; // Il n'est plus utile de continuer le
  }
}
```

LE MOT CLÉ CONTINUE

Permet de “**sauter**” l’itération courante d’un bloc d’instruction **for**, **while** ou **do while**.

```
String chaine = "Hello world" ;
// Comptage du nombre de 'l'
int nb = 0 ;
for (int i = 0 ; i< chaine.length() ; i++){
    if (chaine.charAt(i)!='l')
      continue ; // On passe à i+1
    // On traite le caractère
    nb++;
}
```

L’INSTRUCTION SWITCH

Permet de placer le contrôle de flux à un endroit spécifique en fonction de la valeur d’une variable parmi un ensemble donné :

```
switch(variable){
  case valeur1 : instr1 ; instr2 ; //...
  case valeur2 : instr3 ; instr4 ; //...
  case valeur3 : instr5 ; instr6 ; //...
}
```

Dès qu’une des conditions est vérifiée, le code exécute **toutes les instructions suivantes**

Si variable = valeur2 alors instr3 → instr4 → instr5 → instr6

Penser à l'instruction **break** et à l'instruction **default**

```
switch(variable){
  case valeur1 : instr1 ; instr2 ; break
  case valeur2 : instr3 ; instr4 ; break
  case valeur3 : instr5 ; instr6 ; break
  ...
  default: instrDefault;
}
```

LES EXCEPTIONS

BUT

- Lors de la compilation, le compilateur vérifie surtout le respects de **contrats** (signature des fonctions).
  - ex. : on passe les bons types en arguments
- Il ne vérifie pas la cohérence des valeurs
- Le but des exceptions est de “signaler” les problèmes
- Représente un cas particulier de contrôle de flux.

EXEMPLE

```java
public class Main {
  public static void main(String[] args) {
      System.out.println(1/0);
  }
}
```

Résultat dans le terminal :

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero at
Main.main(Main.java:3)
```

QUE SE PASSE-T-IL ?

- Je réalise une division par zéro avec des entiers
- Il s’agit d’une erreur arithmétique détectée par la fonction qui encode la division des entiers
- Cette fonction **lève une exception** et la transmet à la méthode appelante (ici **main**)
- **main** reçoit cette exception. Par défaut, l’exécution de **main** s’arrête et le texte descriptif de l’exception s’affiche dans la console

PRINCIPE EXCEPTIONS ?

- C’est une mesure de protection pour éviter des comportements non prévus
- Lorsqu’une exception est levée, la méthode appelante la reçoit. Si elle n’est pas traitée, l’exécution de la méthode est interrompue et l’exception remontre à la méthode appelante
- Si l’exception n’est toujours pas traitée par la méthode principale (**main**), alors le programme s’arrête et le contenu de la pile d’exécution est affichée (on voit toutes les fonctions appelées)

COMMENT TRAITER LES EXCEPTIONS ?

Il faut essayer !

```
public class Main {
  public static void main(String[] args) {
      try{
        System.out.println(1/0);
      }catch(Exception e){
        System.out.println("Ce n'est pas très bien
      }
  }
}
```

```
catch(Exception e)
```

- **J’attrappe** l’exception **e** (de type **Exception**)
- Je peux alors réaliser un traitement spécifique adapté à la situation

Attraper une exception spécifique

```
public class Main {
  public static void main(String[] args) {
    try{
      System.out.println(1/0);
    }catch(ArithmeticException e){
      System.out.println("Ce n'est pas très bien
    }
  }
}
```

Traiter plusieurs exceptions

```
public class Main {
  public static void main(String[] args) {
    try{
      System.out.println(1/0);
    }catch(ArithmeticException e){
      System.out.println("C'est une exception ari
    }catch(Exception e){
      System.out.println("C'est une autre exceptio
    }
```

LE MOT-CLÉ “THROWS”

- Par défaut, les exceptions sont vérifiées par le compilateur
- Si votre méthode est susceptible de ne pas traiter un type d’exception et de le renvoyer, il faut en principe le déclarer dans la signature.

```
public static void main(String[] args) throws Arit
  System.out.println(1/0);
}
```

- Certaines exception étant très communes, les déclarer via **throws** est superflu. C'est le cas de *ArithmeticException*
- Pour d'autres exceptions (ex. *IOException*), il faut les déclarer via **throws** sous peine d'erreur de compilation.
- Liste des exceptions par défaut

LEVER UNE EXCEPTION

Utiliser le mot clé **throw**

```
public void printAge(int i){
  if (i<0){
    throw(new IllegalArgumentException("Age >0 !")
  }else{
    System.out.println("Vous avez "+i+" ans !");
  }
}
```

LA STRUCTURE D'UNE APPLICATION JAVA

LA FONCTION MAIN

Il est possible “d’exécuter” un fichier *.java* si et seulement si celui-ci contient une fonction **main** dont la signature est la suivante :

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

- **public static** : nécessaire pour les fonctions
- **void** : la fonction ne retourne rien.
- **String**[] **args** : l'argument de main est un tableau de String.

POURQUOI STRING[] ARGS?

- Identification des arguments lors d'une commande textuelle dans un terminal

```
$ ls -l *.java      (unix)
$ dir *.java        (windows)
```

Liste tous les fichiers avec l'extension java et les présente sous forme de liste

RÉCUPÉRER LES ARGUMENTS

```
$ ls -l *.java
```

- Exemple avec la ligne de commande unix
  - **ls** : nom de la commande
  - **-l** : premier argument
  - ***.java** : deuxième argument
- Du point de vue de la fonction `main` :
  - args[0] = "-l"
  - args[1] = "*.java"

POUR EXÉCUTER

Se mettre dans le répertoire contenant le fichier .class (ex. MainClass.class) issu de la compilation à l'aide de la commande **cd**.

```
$ java MainClass argument1 argument2....
```