Open classrooms : java

Déclarer une variable

Variable : Association d'un nom à une valeur. Elle est déclarée via un mot-clé et peut-être de type texte, nombre, booléen, etc.

Une variable est un outil contenant une donnée, par un exemple un mot ou un chiffre, et qui va être utilisée par un programme

Une donnée placée dans une variable s'appelle une valeur, et chaque boite contient une valeur

Boucle : Répétions d'un bloc de code tant que la condition spécifiée est valide

Fonction : Ensemble d'instructions qui effectuent une tâche. Une fonction peut être appelée plusieurs fois dans un programme.

Tableau (array) : Bloc de mémoire où de même type sont rangées côte à côte

Programmation orientée objet : Modèle qui consiste à définir et faire interagir des éléments appelés « objet »

Objet : Représentation qui se rattache au monde physique : un livre, une page de livre, une lettre.

Bonne pratiques	Erreurs Classique
- Nommer les variables, les classes et les fonctions de manière explicite	- Faure des fonctions trop longues qui font trop de choses
- Indenter son code pour ne pas se perdre	- Utiliser le mauvais type de variable
- Contrôler l'accès aux variables et aux fonctions	- Créer une boucle infinie : si la condition est toujours vraie, le programme ne s'arrêtera jamais
- Initialiser une variable au moment de la déclaration, si sa valeur est connue.	- Ne pas gérer les exceptions qui peuvent survenir lors de l'exécution du programme

On ne crée pas une variable mais on la déclare.

Une affectation normale : epargne = epargne + 100 ;

Un raccourci : epargne += 100 ;

Variantes courtes :

- +=
- -=
- *=
- /=

3 variables à connaitre :

- String = texte ;
- int ; nombre entier ;
- double = nombre en virgule flottante.

Attention String prend une MAJ alors que int et double non

Une variable avec une valeur qui ne change pas → une constante

<u>Une constante</u> : Type, non et valeur

Une fois défini, la valeur d'une constante ne peut plus être modifiée.

Utiles pour deux raisons **:**

- Elles permettent aux programmes d'être plus rapide
- Certaine valeurs ne change pas (intentionnellement ou par accident) (exemple jour dans la semaine, ou dans l'année)

le nom d'une constante en java est toujours en maj, ça permet de mieux les reconnaitre !

ex : final int NBREDEJOURDANSLASEMAINE = 7 ;

final string MONPLATPREFERE = " Pizza " ;

// note personnel : quand on met final = devient une constante

Si on modifie une constante = message erreur

En résumé :

- Une déclaration de variable est composée de trois éléments : type, nom et valeur
- Les valeurs des variables peuvent être modifiées
- Les variables à valeurs constantes sont appelées constantes
- Le nom des variables doit respecter les conventions de dénomination courantes

Une division avec int le résultat est arrondi exemple

Int a = 5 ;

Int b = 2 ;

Int c = a/b ;

Résultat 2

Si on veut avoir la réponse exacte : il faut avoir le résultat en float càd = float c ……. Et une des variables en float ex float b = 2 ;

Booléen : C'est que True / False

- Boolean isCodingJava = fale ;
  // je peux modifier la valeur
- isCodingJava = true ;
  // ou je peux aussi la modifier avec un point d'exclamation pour une inversion (non logique)
- isCodingJava = !isCodingJava ;

String : pour les caractères

String nomVariable = " texte " ;

Avec le signe + nous pouvons ajouter plusieurs variables → concaténation

Résumé :

- vous pouvez effectuer des opérations numériques sur des nombres du même type
- pour utiliser ensemble des nombres de types différentes dans les opérations, utilisez « Cast » pour qu'ils se comportent comme le type nécessaire
- les strings peuvent être mis bout à bout → concaténation

Déclarer la fonction principale java :

Fonction main : public static void main (String[] args){*ici pour entrer les variables*}

// commentaire pour autre développeur ou personnel

Méthode classe String :

Exemple : String exemple = « hello » ;

Exemple = exemple.toUpperCase() ; → ici il va transformer le hello en HELLO (en maj)

On peut remplacer une parti du mot avec la méthode (.replace)

Exemple = exemple.replace (« HELL », « YEAH ») ;

Résultat : YEAHO (car pas dit de remplacer le o

Porter fonction

Les variables ont une porté (SCOPE)

La porté des variables en java dépend de la où on les déclare

Si on a plusieurs public class la variable est reconnu que dans les guillemets mais pas à l'extérieur de celles-ci

Une variable peut être local ou globale tout dépend de l'endroit ou elle est déclarée.

Une variable global est disponible pour toutes les classes alors que la locale que pour une classe (dans la classe ou elle est déclarée.

Public vs private :

Public : toutes les classes peuvent utilisés les variables (Attention dépend ou elle est déclarée)

Private : reste privé càd les autres classes ne peuvent plus « avoir accès » à la variable.

Il y a aussi le « protected » et « package-protected »

Un fichier de code = un fichier source

Framework = un ensemble de fonctionnalités regroupées par contexte particulier.

Si on ne spécifie pas la classe, il prendra la classe private par défaut

Boucle

Une boucle sert à ne pas se répéter (copier-coller par exemple)

Les boucles = for

For(int i = 0 ; i <=1000 ; i++){

Attention aux conditions pour ne pas faire une boucle infinie

Donc ça donne quoi

Exemple :

For (int i=0 ; i<5 ;i++{

System.out.println(« clap your hands!");

}

Cette ligne de commande donne ça :

1. **System.out.println(« clap your hands!");**
2. **System.out.println(« clap your hands!");**
3. **System.out.println(« clap your hands!");**
4. **System.out.println(« clap your hands!");**
5. **System.out.println(« clap your hands!");**

La boucle for est utilisé lorsqu'on sait combien de fois on veut la répéter

Répéter une boucle jusqu'à atteindre une condition

1. While ;
    - While (logicalExpression){
    - //liste de déclarations
    - }
2. Do …….. while
    a. Do{
    b. //instructions
    c. }while ((logicalExpresssion) ;

On peut interrompre une boucle avec « break »

Affectation conditionnelles

+ vidéo à regarder

Opérateurs de comparaison

== : égal

!= : non égal

< : inférieur à

<= : inférieur ou égal à

> : Supérieur

>= : Supérieur ou égal

Opérateurs logiques :

&& : Et logique

|| (double bar) : ou logique

! : non logique

Exemple :

```
true && true // -> true
true && false // -> false
false && false // -> false

true || false // -> true
true || true // -> true
false || false // -> false

!true // -> false
!false // -> true
```

```java
true && true && true // -> true
true && true && false // -> false

true || false || false// -> true
false || false || false// -> false
```

Comme pour les opérateurs numériques, les opérateurs logiques respectent la priorité des opérations : l'opérateur d'inversion `!` vient en premier, puis l'opérateur ET `&&` et enfin, l'opérateur OU `||`. Par exemple :

```java
false || true && true // -> true

!false && true || false // -> true
```

Comme pour les opérateurs numériques, utilisez les parenthèses `()` pour changer l'ordre :

```java
(true && false) || true // -> true
```

⚠ on peut mettre des parenthèses pour changer l'ordre

exemple:

1. (true && false) || true → true
2. !(true && false || !true) → True

Args ???

Switch :

ensemble de conditions à remplir ou elle peut être true.

⚠ chaque fin de conditions mettre : break;

- switch rend l'intention plus claire que if/else

Enumération :

- liste de car prédéfinis

exemple :

enum Direction {
north, east, South, west;
}

Exemple 1 :

```java
public class myDirection {

    /** listez toutes les directions possibles */
    enum Direction {
        north, east, south, west;
    }

    /** trouvez le nord */
    public static void main(String[] args) {
        Direction direction = Direction.north;
        switch (direction) {
            case north:
                System.out.println("You are heading north");
                break;
            case east:
                System.out.println("You are heading east");
                break;
            case south:
                System.out.println("You are heading south");
                break;
            case west:
                System.out.println("You are heading west");
                break;
        }
    }
}
```

Exemple 2 :

```java
public static void main(String[] args) {
    Direction direction = Direction.north;
    switch (direction) {
        case north:
            System.out.println("You are heading north");
            break;
        default:
            System.out.println("You are lost!");
    }
}
```

2eme partie : Poo

- class NomClasse { .

  int pixels;

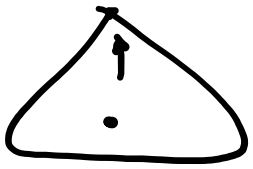
  String nom;

  Marque marque

}

class Marque {

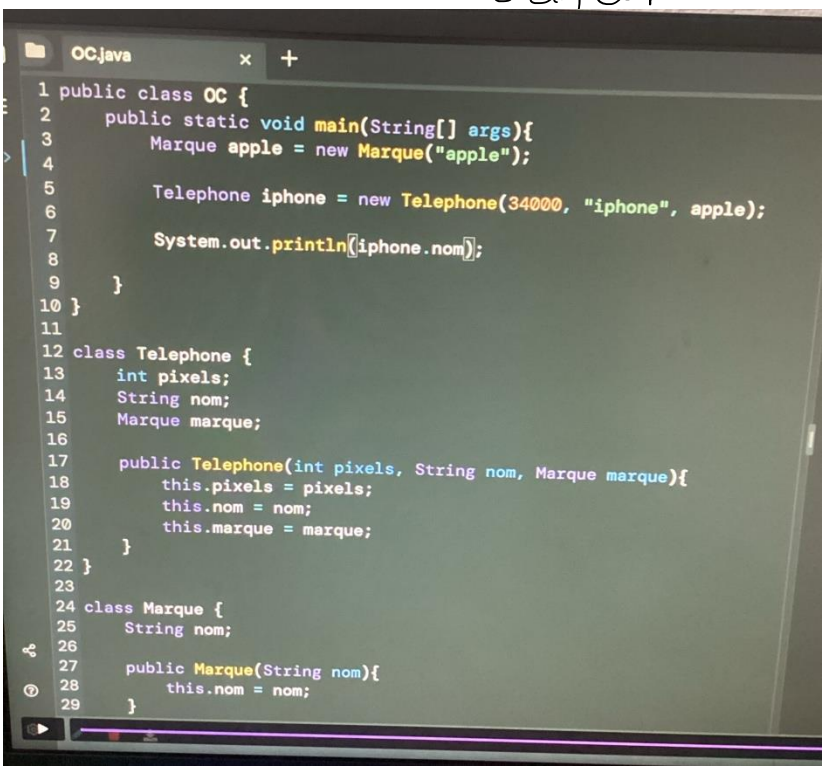
  String nom;



2 majuscules : class NomClasse

- Concevoir une classe :

  Exemple : - titre
  - auteur
  - nbre de page
  - Éditeur



```java
public class OC {
    public static void main(String[] args){
        Marque apple = new Marque("apple");

        Telephone iphone = new Telephone(34000, "iphone", apple);

        System.out.println(iphone.nom);
    }
}

class Telephone {
    int pixels;
    String nom;
    Marque marque;

    public Telephone(int pixels, String nom, Marque marque){
        this.pixels = pixels;
        this.nom = nom;
        this.marque = marque;
    }
}

class Marque {
    String nom;

    public Marque(String nom){
        this.nom = nom;
    }
```

On peut avoir plusieurs class

Dans la class mettre :

public nom (reprendre les variables)

enumérer les variables : this.nom = nom

Pour utiliser une class → créer un objet concret

exemple livre: Alice au pays des merveilles → Instance

processus est appelé: instanciation ou initialisation

créer des instances de class:

```
class Nom {
    string title
    int page
    double ...
    etc.

    Nom ( String title, int page, String Author ) {
```

une fois les variables citées reprendre nom de class

Re citer

```
        this.title = title;
        this.page = page;
        this.Author = Author;
    }
}
```

ça c'est initialisé, obligatoire si non programme plante

On peut aussi déclarer plusieurs constructeurs différents.

```
class Nom {
string ... ;
   :
Nom ( String ..... , ... ) {
this ( ..... , ..... )
}
```

```
Nom (string ..., int ...) {
    this.___ = ___;
    ___. ___ = ___;
    ___. ___ = ___;
}
}
```

Exemple code créer un livre :

```
Book myBook = new Book("L'art de coder", "Raphaël B.", 425);
```

Book : Type (nom de la classe)

myBook : nom variable

new : mot-clé

Book : nom de la classe

("L'art de coder", "Raphaël B.", 425) : valeur de champs

cibler l'attribut :

```
myBook.title = "L'art de coder";
myBook.author = "Raphaël B.";
myBook.numberOfPages = myBook.numberOfPages + 10;
```

Résumé :

- Une classe est le plan d'un objet
- une classe vous permet de créer des types complexes en regroupant ses attributs, en définissant des champs.
- Pour créer un objet, vous devez déclarer une variable d'une classe et l'instancier. Utiliser le point (.) donne accès aux champs.

Spécialiser les classes avec l'héritage :

" On essaye de tout regrouper dans le même sac "

exemple :

créer une classe mère :

```
class Vehicule {
    void start () {
    System.out.println ("VROOOOOOM");
    }
}
```

Classe filles

```
class Voiture extends Vehicule {
}

class Bateau extends Vehicule {
}
```

En premier

```
class principal
Psvm (String [] args) {
Voiture voiture = new Voiture();
voiture.start();
Bateau bateau = new Bateau();
bateau.start();
```

Rappel : un champ défini comme private ne peut pas être hérité.

classe fille ne peut hériter qu'une classe mère

mais une classe mère peut être une classe fille d'une autre classe mère

Polymorphisme :

surchargé et/ou remplacé les comportements de la classe mère.

Réf classe fille

```
class voiture extends vehicule

    @override
    void start();
     super.start();
     allumerFeux();
}

    void allumerFeux {

    Sout(" Allumage feux");
```

Résumé :

- l'héritage est un concept fondamental en Java qui permet de réutiliser du code de la classe mère.
- le polymorphisme permet de "surcharger" les méthodes de la classe mère pour redéfinir leurs comportements sans changer leur signature.

Gérez les piles de données :

- tableau / array
  ↑ = ↑↳ anglais

Pour créer un tableau, il faut indiquer quel type : string, int ...

Exemple :

```
String [] tableau = new String [10];
```

type — nom variable — pour créer — reprendre son type — nbre d'éléments que je veux stocker

```
tableau [3] = "Index 3";
tableau [1] = "Index 1";
for (int i; i < tableau.length; i++) {
System.out.println ( tableau [i]);
```

⚠ Index commence à 0 et non 1

* On peut écraser une donnée. On réécris plus bas : `tableau [3] = "nouvelle donnée";`

⚠ une fois un tableau déclaré, on peut ni modifier son type, ni sa taille

Un tableau est une liste ordonnée et numéroté d'éléments du même type.

- éléments du tableau = Index

modèle :

```
int [] myArray = new int [3];
```

- `int` : Type
- `[]` : crochets
- `myArray` : nom du tableau
- `new` : mot-clé
- `[3]` : nbre d'éléments dans le tableau

Tableau crée

une fois crée :

1. Accèder à une valeur à un index donné
2. Définir une nouvelle valeur à un index donné

exemple café : je crée un tableau avec 7 éléments

pour afficher le premier jour :

```
System.out.println(nomTableau[0]);
```

Afficher toutes les valeurs :

- soit définir toutes les valeurs au moment création tableau
- Soit utiliser une boucle qui définit chaque valeur 1 par 1

pour créer un tableau avec des valeurs:

```
nomTableau = new int [] {5; 4; 7; 2; 0 ...};
```

le type : string etc.

Tableaux multidimensionnels :

exemple salle théâtre : 30 rangs de 12 places

attribuer une valeur au 6ème siège du 10ème rang

```
// création
String [][] nomTableau = new String [30] [12];
   myTheatreSeats
// Index commence à 0 (zéro)
myTheatreSeats [9] [5] = "James Logan";
```

Les collections (liste ordonnée) → la liste le plus utilisé

- Utilisez des listes si le nombre d'éléments n'est pas fixe.

les tableaux / Array:
- Taille fixe
- modifier que les valeurs existantes

Exemple collections : liste animaux, on commence à 4 ensuite on on rajoute.

Append: ajouter des éléments dans une liste mais conserve la forme d'itérable.

Avec les listes :
- accéder : à chaque élément via son index
- ajouter (append) : un nouvel élément à la fin
- insérer : un nouvel élément à un index spécifique
- Supprimer : un nouvel élément à un index spécifique

⚠ on peut changer la valeur mais pas le type → ArrayList

- On peut créer notre propre "list" → l'important est que la classe que vous sélectionnez utilise l'interface List

Note : une interface n'est rien d'autres qu'une classe définissant les signatures des méthodes que devront implémenter ttes les classes l'implémentant. C'est un contrat imposé par celui qui l'écrit à toutes les classes qui l'utiliseront.

création :

```
List<Integer> myList = new ArrayList<Integer>();
```

- `List<Integer>` : Type
- `myList` : nom variable
- `=` : mot-clé
- `ArrayList<Integer>()` : type de liste

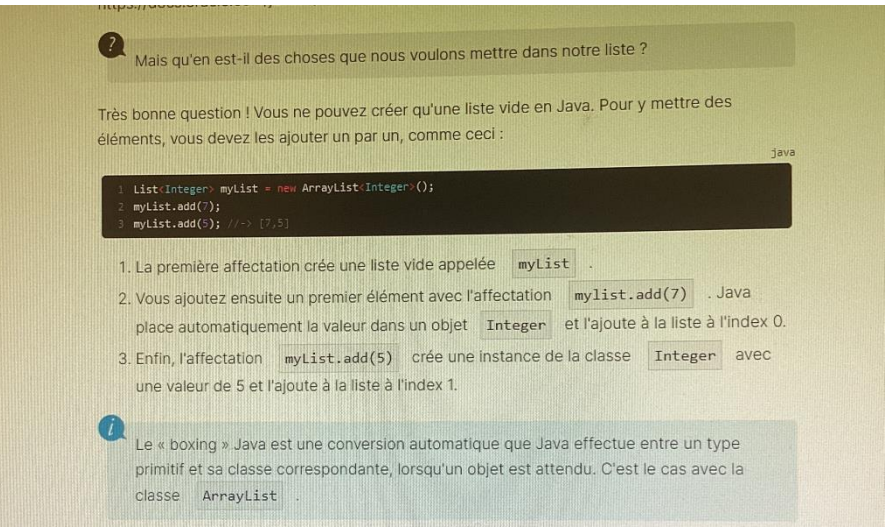
⚠ type en Maj. ils stochent que des objets

Integer au lieu de integer

Double au lieu de double

Boolean au lieu de boolean

Float au lieu de float



Mais qu'en est-il des choses que nous voulons mettre dans notre liste ?

Très bonne question ! Vous ne pouvez créer qu'une liste vide en Java. Pour y mettre des éléments, vous devez les ajouter un par un, comme ceci :

java

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(7);
myList.add(5); //-> [7,5]
```

1. La première affectation crée une liste vide appelée `myList` .
2. Vous ajoutez ensuite un premier élément avec l'affectation `mylist.add(7)` . Java place automatiquement la valeur dans un objet `Integer` et l'ajoute à la liste à l'index 0.
3. Enfin, l'affectation `myList.add(5)` crée une instance de la classe `Integer` avec une valeur de 5 et l'ajoute à la liste à l'index 1.

Le « boxing » Java est une conversion automatique que Java effectue entre un type primitif et sa classe correspondante, lorsqu'un objet est attendu. C'est le cas avec la classe `ArrayList` .

- Add() : c'est une méthode
- méthode = permet de faire des choses
- Interface : List → il y a 3 méthodes :

1. add : pour ajouter un nouvel élément à la fin du tableau
   On peut aussi insérer un nouvel élément à une position donnée en spécifiant l'index avant la valeur.

Par exemple : myList.add(1,12) → insère un entier avec la valeur 12 sur la position 1 et déplace la valeur existante sur la position 2 et ainsi de suite.

2. Set : Pour remplacer un nouvel élément sur un index spécifique. Ici vous devez fournir un nouvel élément et l'index sur lequel vous voulez qu'une nouvelle valeur soit positionnée.

3. remove : pour supprimer un élément existant de l'Index vous devez fournir l'index de l'élément que vous souhaitez supprimé. Si plus besoin du premier élément, on peut le retirer de la position 0. Cela déplacera l'élément 2 à la position 1 et la 1 à la 0 et ainsi de suite.

A quoi ça ressemble :

```
1 List<Integer> myList = new ArrayList<Integer>(); // -> []
2 myList.add(7); // -> [7]
3 myList.add(5); //-> [7, 5]
4 myList.add(1,12) //-> [7, 12, 5]
5 myList.set(0,4); // -> [4, 12, 5]
6 myList.remove(1); // removed 12 -> [4, 5]
```

Décomposons cela un petit peu, d'accord ?

- Vous pouvez voir la même opération que précédemment : ajouter 7 puis 5 à la liste.
- Puis vous insérez 12 à l'index 1. La valeur existante de l'index 1 est déplacée vers l'index 2.
- Ensuite, avec `.set()`, le premier nombre fait référence à l'index, et le second donne la valeur que vous voulez y mettre. En d'autres termes, vous demandez à votre `List` de modifier la valeur de l'index 0. Ceci transformera la valeur d'origine, 7, en nouvelle valeur, 4.
- Pour finir, avec `.remove()`, vous demandez à votre liste de supprimer toute valeur trouvée à l'index 1. Cela veut dire que vous lui demandez de supprimer 12.

Ceci vous laisse avec votre liste finale contenant deux entiers : 4 et 5. Imaginez faire cela avec des tableaux de taille fixe !

N'oubliez pas d'**utiliser les bons indices**. Si votre liste contient dix éléments, et que vous essayez d'en insérer ou supprimer 11, vous obtiendrez une erreur ! N'oubliez pas d'assurer un bon suivi.

une méthodes très importante :

- size() → permet d'obtenir le nombre d'éléments dans une liste.

```java
List<Integer> myList = new ArrayList<Integer>();
myList.add(7);
myList.add(5); //-> [7,5]
System.out.println(myList.size()); //→ 2
```

- Facilite grandement le suivi de l'emplacement des éléments de votre Liste

- Aussi possible dans les tableaux ! → myArray.length au lieu de la méthode : myListe.size()
propriété

- la méthode size() est utilisée, notamment lorsque vous avez besoin de faire une boucle sur une liste.

Utilisez une collection non ordonné - ensembles :

Un ensemble ou (Set) est une collection d'élément uniques non ordonnés. utiliser quand l'ordre n'est pas important → liste ingrédients pour une recette

Déclarez des ensembles :

Java à différentes classes pour gérer les **ensembles**

- HashSet :

```
Set<String> ingredients = new HashSet<String>();
```

Type — nom variable — mot-clé — Type d'ensemble

manipuler les éléments d'un ensemble :

- ajouter un nouvel élément avec une nouvelle clé
- supprimer un élément pour une clé spécifique
- compter le nombre d'éléments de l'ensemble

⚠ - Accéder ? → vu que élément non ordonnés pas de moyen simple de pointer des éléments particuliers
c'est possible mais avec d'autres moyens → apprendre avec le temps

- Add.
  - `ingredients.add("eggs");`
  - `ingredients.add("Sugar");`
  - `ingredients.add("butter");`

⚠ pas besoin de "Append" car non ordonnées

- remove : ingredients.add("salt");
  ingredients.remove("salt");

- size : System.out.println(ingredients.size());

Un dictionnaire/Map :

- Un dictionnaire est une liste d'éléments organisé en fonction d'une clé, cette clé est un terme spécifique que vous recherchez pour trouver sa définition ou sa **valeur**. C'est ce qu'on appelle une association "clé <> valeur" (key<>value)

- Exemple: lister nom + âge des amis :

Jenny	Livia	Paul
24	38	31

- toutes les clés d'un dictionnaire doivent être uniques, comme les numéros de plaque des voitures

(i) -> info Le type de clé utilisé le plus -> String

Declarez des dictionnaires:

interface : Map

classe : HashMap

code :

```
Map<String, Integer> myMap = new HashMap<String, Integer>();
```

mot-clé · T. clé · T. valeur · Type dictionnaire

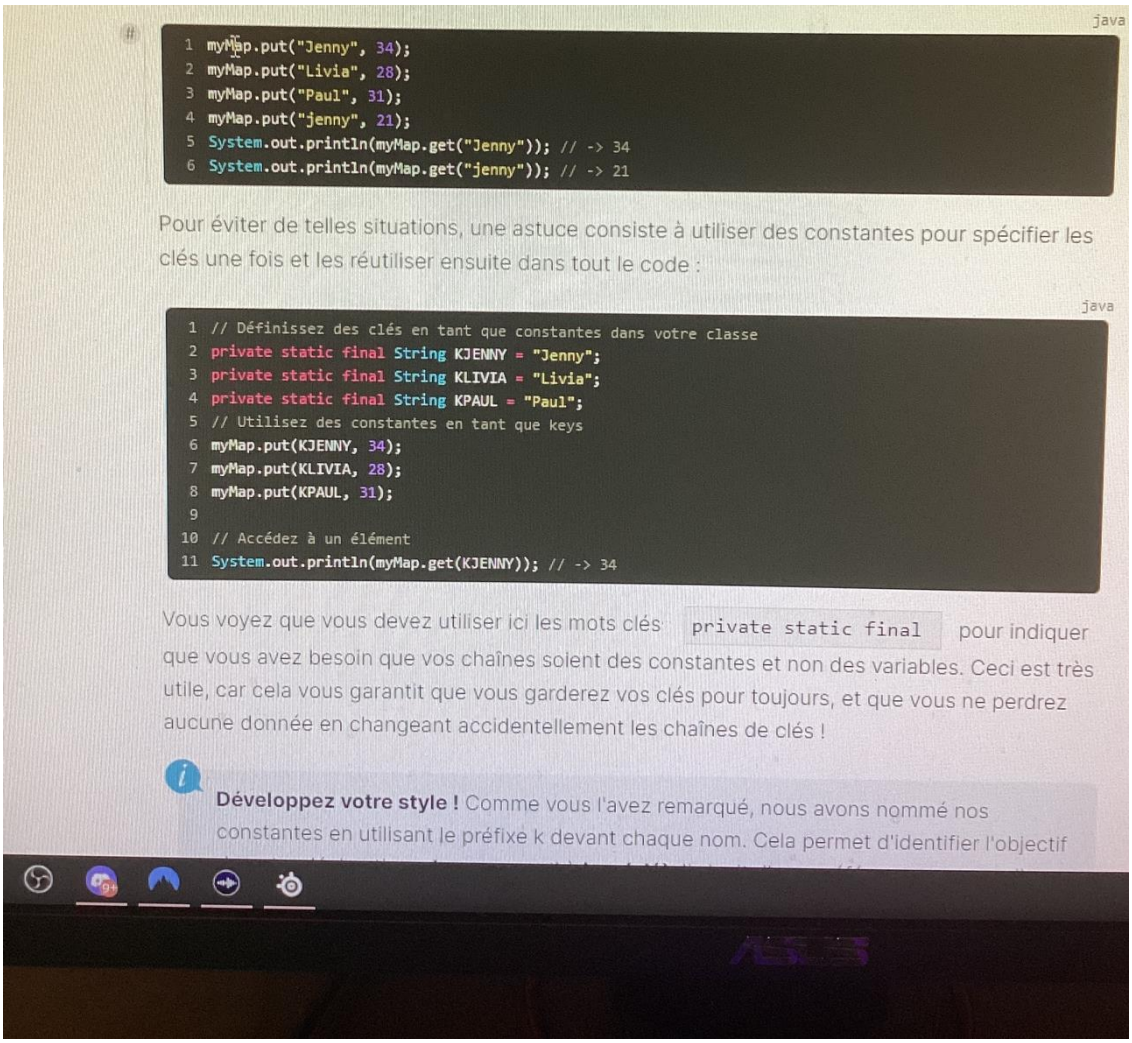
- type clé
- Type de valeur

- On le paramètre avec 2 éléments :
  1. String est le type de la clé
  2. Integer est le type de la valeur.

- Ajouter des éléments
  - Méthodes : myMap.put()

ex :
```
myMap.put("Jenny, 24");
      .put("Livia, 28");
      .put("Paul, 31");
System.out.println(myMap.get("Jenny")); //=24
```

↳ méthode : Sout → affiche le résultat
↳ expression réelle : myMap.get("Jenny") → renvoie la valeur identifiée.



java

```java
myMap.put("Jenny", 34);
myMap.put("Livia", 28);
myMap.put("Paul", 31);
myMap.put("jenny", 21);
System.out.println(myMap.get("Jenny")); // -> 34
System.out.println(myMap.get("jenny")); // -> 21
```

Pour éviter de telles situations, une astuce consiste à utiliser des constantes pour spécifier les clés une fois et les réutiliser ensuite dans tout le code :

java

```java
// Définissez des clés en tant que constantes dans votre classe
private static final String KJENNY = "Jenny";
private static final String KLIVIA = "Livia";
private static final String KPAUL = "Paul";
// Utilisez des constantes en tant que keys
myMap.put(KJENNY, 34);
myMap.put(KLIVIA, 28);
myMap.put(KPAUL, 31);

// Accédez à un élément
System.out.println(myMap.get(KJENNY)); // -> 34
```

Vous voyez que vous devez utiliser ici les mots clés `private static final` pour indiquer que vous avez besoin que vos chaînes soient des constantes et non des variables. Ceci est très utile, car cela vous garantit que vous garderez vos clés pour toujours, et que vous ne perdrez aucune donnée en changeant accidentellement les chaînes de clés !

Développez votre style ! Comme vous l'avez remarqué, nous avons nommé nos constantes en utilisant le préfixe k devant chaque nom. Cela permet d'identifier l'objectif

Au lieu des "K" on met ce qu'on veut.

- On peut avoir des clés de type Integer

- Map < Integer, String > ---------- < Integer, String ...
- put on inverse aussi
  - myMap.put(1, "wakeup");

- Toutes les clés doivent être du même type et les valeurs aussi doivent être du même type.

Manipulez éléments dictionnaires:

- accéder à la valeur d'un élément par sa clé
- ajouter un nouvel élément avec une nouvelle clé et une valeur
- Supprimer un élément pour une clé spécifique

: ajouter : nomVariable.put("xx","yy");

- accèder : dans les "()" du sout → get("xx").
- on peut aussi modifier avec "put" → écraser l'autre valeur
- supprimer : nomVariable.remove("Jenny")
- compter : size()

Sout(nomVariable.size());

En résumé

Dans ce chapitre, vous avez appris les bases pour travailler avec des conteneurs qui stockent plusieurs éléments d'un type de données en particulier :

- conteneurs de taille fixe : **Arrays –** les éléments d'un tableau sont indexés à partir de 0 et sont accessibles à l'aide de cet index. Le nombre d'éléments ne peut pas être modifié ;
- listes ordonnées : **Lists –** les éléments d'une liste se comportent comme un tableau. Le nombre d'éléments peut évoluer en ajoutant et en supprimant des éléments ;
- listes non ordonnées : **Sets –** les éléments d'un ensemble sont stockés sans ordre particulier. Vous pouvez y accéder en les énumérant ;
- dictionnaires : **Maps –** les éléments d'un dictionnaire sont organisés par paires clé-valeur et sont accessibles à l'aide d'une clé ;
- les actions les plus courantes effectuées avec les collections sont :
    - accéder à un élément,
    - ajouter un nouvel élément,
    - supprimer un élément,
    - modifier un élément,
    - compter tous les éléments,
    - parcourir tous les éléments.

Gerez les differents types de passage paramètre

- Différence type de valeur et type de référence

exemple : calculer perimètre rectangle

```
int perimetre = 2 * (4 + 6);
```

↳ problème car tjs le mm perimetre

- créer une méthode : -> = paramètre

↳ public static void Rperimetre (int length, int width)

```
int perimetre = 2 * (length + width);

Sout (perimetre);
```

arguments

↳ Rperimetre (10, 11) => 42

↳ Rperimetre (2, 2) => 8

- Valeur de retour

1. modifier la déclaration de la fonction

pour indiquer qu'il est prévu qu'elle renvoie un résultat

2. Terminer la fonction avec → Return

```java
int sumOfSmallNumbers = sum(3,7);
System.out.println(sumOfSmallNumbers); // -> 10
int sumOfLargerNumbers = sum(sumOfSmallNumbers,999);
System.out.println(sumOfLargerNumbers); // -> 1009
```

Dans l'exemple ci-dessus, nous avons utilisé le résultat du premier calcul comme paramètre pour le suivant.

Un autre élément à comprendre et à assimiler : certaines variables stockent une valeur directement, tandis que d'autres stockent une référence. Cela a un impact sur la façon dont elles sont envoyées comme arguments à une fonction.

type valeur ou type référence

type de valeur :

Si je crée un int et que je lui donne une valeur et que j' affecte cette

valeur à une seconde variable int,
la valeur de la première sera copié sur le 2ème

Type de référence :

Les types de références sont aussi appelés
Pointeurs → ils pointent vers l'objet, mais
pas l'objet en tant que tel.

les classes → toujours type de référence

les nombres → toujours type de valeur

Si on passe un type de réf. en paramètre,
toutes les modif. que je ferai sur eux dans
les fonctions modifieront les objets originaux

les paramètres → constantes

En résumé :

- les fonctions peuvent avoir des paramètres et des valeurs de retour (return)
- une valeur de retour est le résultat d'une exécution de la fonction. Elle peut renvoyée au bloc de code qui a appelé la fonction, et être utilisé si besoin est.
- les paramètres sont les entrées d'une fonction **nécessaires à l'exécution et** l'obtention d'un résultat
- les paramètres sont des variables définies par leur nom et par leur type. Ces paramètres sont spécifiés dans la déclaration de la fonction.
- Lorsque vs appelez une fonction, vs passez des valeurs à des variables. ces valeurs sont appelées **arguments.**

Récursivité

Factorielle :

$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

$N! = N \cdot (N-1)! \rightarrow 5! = 5 \cdot (\underbrace{5-1}_{4})! \rightarrow 5! = 5 \cdot 4!$

- **créer** une fonction qui s'exécute de façon répétée
- une méthode est appellée à l'intérieur d'elle même. → récursivité
- En termes techniques, la récursion est une action qui s'initialise à l'intérieur d'elle même. Elle continue de descendre couche par couche **jusqu'à qu'une** condition soit rempli et remonte à la 1ère couche.

Résumé :

- la récursivité est une action qui s'initialise à l'intérieur d'elle même.
- les méthodes récursives servent à passer des structures en couches.

Gerez les comportements inattendus :

Gerez les erreurs :

↳ faute de frappe à utilisation incorrecte de variable
↳ il y a aussi les erreurs logiques métiers → produit résultat incorrecte.

Quand j'exécute :

- compilation ou interprétation : code vérifié et traduit en code machine. un fichier prêt à exécuter → **exécutable**
- Exécution - exécuter l'application - exécuter le fichier exécutable

Type d'erreurs : orthographes, ↳ syntaxiques
sémantique ↳ mot interdit ↳ type incorrect

Gerez les erreurs d'exécution

Apparaissent que pdt le processus d'exéchion
↳ erreur : logique du code, accès à des données qui n'existent pas ou dans un format différent.

logique : pensez que la liste était plus longue

logique métier : inverser la significations des termes : Débit / crédit

Une erreur logique = un plantage

On appelle ça un **crash**

- le débogage : regarder le programme ligne par ligne et observer l'état des variables.

2 stratégies :

1. Recherche d'erreurs à l'aide de conditions
2. Trier parti du mécanisme des exceptions

Gerer les exceptions :

try / catch → (essayer / capturer)

↳ on demande à Paire qqch. par exemple exécuter un bloc de code ou "Try" essayer de le faire. Si il y a une erreur, on la capture "catch"

```
try {
// un peu de code
// une fonction à essayer pouvant générer une erreur
// encore du code
}
```

catch (ExceptionName e) {

// code à exécuter au cas où l'essai ne fonctionnerait pas et qu'une erreur se produirait.

En cas de problème à l'intérieur d'une méthode pdt son exécution, une erreur est générée → **throw**

catch: si aucune inscription d'interception n'est fournie le programme finit par planter.

<u>En résumé:</u>

- la compilation est le processus du code langage de programmat° en machine. les erreurs sont faciles à découvrir et doivent être résolues pour terminer le processus de compilation.
- l'exécution est le processus d'utilisation ou d'exécution de l'application. **Les erreurs d'exécution** sont les plus durs à découvrir et provoquent un crash.
- En Java, les erreurs d'exécutions génèrent des exceptions
- les exceptions sont gerées à l'aide d'une instruction try/catch
  - Try: indique qu'une fonction peut lancer une exception
  - catch: indique quel code exécuter si une exception est générée (throw)

Manipuler des fichiers

- Try catch qui nous permet de gérer les exceptions.
- un Filereader qui va lire le fichier au chemin qui lui est donné en argument
- un BufferredReader qui va utiliser le Filereader pour lire le fichier ligne par ligne.
- une condition while qui ns permet de répéter le code tant que nous ne sommes pas à la fin du fichier.

Ecrire du texte :

- le "try catch" qui ns permet de gérer les exceptions
- un "File writter" qui va préparer le fichier au chemin qui lui est donné en argument.
- un "BufferedWritter" qui va utilisé le "FileWrite" pour écrire dans le fichier.

Résumé :

- lecture contenu fichier ligne par ligne → "BufferedReader"
- Ecriture de texte dans un fichier → BufferedWriter"
- Quand on manipule des fichiers, tjs gerer les exceptions, car les fichiers sont des éléments extérieur à notre programme

Code lambda

- **obtenir** une référence vers une méthode → une closure
  ↳ Fonctionnalité Lambda

lambda: écrire du code compact et lisible
  ↳ Avantages: avoir une alternative aux appels de méthode à partir de classe anonyme

closure: interfaces fonctionnelles → une seule méthode abstraite

- ( ) → action ou
- (paramètre, ...) → une action ou
- (paramètre, ...) → { traitement, retourner qqch }

une expression lambda c'est toujours:

< liste de paramètres > → < le corps de la fonction >

- liste paramètres: indique les paramètres en entrée de l'unique méthode de l'interface fonctionnelle, ils seront sans entre ()
  et pas obligatoires d'indiquer leurs types.
- →: marqueur de l'expression Lambda. il sépare liste paramètre et le corps de la fonction.

Le corps de la fonction: entre { } ou pas, il n'y a qu'une expression courte.
→ expression unique
↳ bloc de code composé ou plusieurs instructions entourées par des accolades.

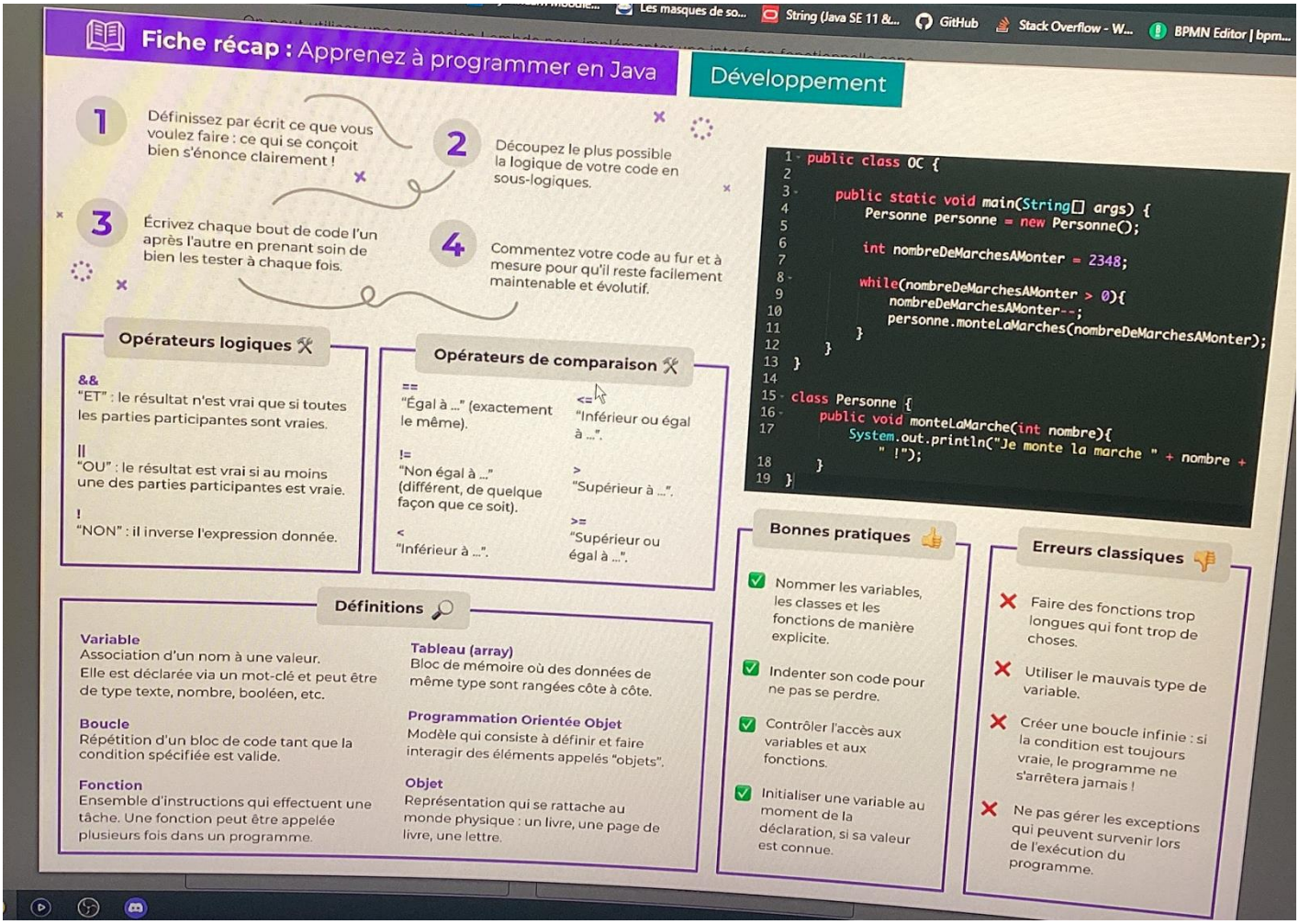
le corps de l'expression doit respecter certaines règles:

- il peut n'avoir aucune, une ou plusieurs instructions.
- Quand il en a qu'une seule, accolades non-obligatoires, et la valeur de retour est celle de l'instruction si elle en possède une.
- Quand il y a plusieurs instructions, obligatoires → accolades
- la valeur de retour et celle de la dernière expression, ou void si rien n'est retourné.

RÉSUMÉ

- une expression Lambda est une référence vers un bloc de code.
- Une interface fonctionnelle → à qu'une seule méthode abstraite
- on **peut** utiliser une expression Lambda pour implémenter une interface fonctionnelle sans déclarer de classe abstraite.

Résumé Général :



Fiche récap : Apprenez à programmer en Java

Développement

1. Définissez par écrit ce que vous voulez faire : ce qui se conçoit bien s'énonce clairement !
2. Découpez le plus possible la logique de votre code en sous-logiques.
3. Écrivez chaque bout de code l'un après l'autre en prenant soin de bien les tester à chaque fois.
4. Commentez votre code au fur et à mesure pour qu'il reste facilement maintenable et évolutif.

```java
public class OC {

    public static void main(String[] args) {
        Personne personne = new Personne();

        int nombreDeMarchesAMonter = 2348;

        while(nombreDeMarchesAMonter > 0){
            nombreDeMarchesAMonter--;
            personne.monteLaMarches(nombreDeMarchesAMonter);
        }
    }
}

class Personne {
    public void monteLaMarche(int nombre){
        System.out.println("Je monte la marche " + nombre +
            " !");
    }
}
```

Opérateurs logiques

&&
"ET" : le résultat n'est vrai que si toutes les parties participantes sont vraies.

||
"OU" : le résultat est vrai si au moins une des parties participantes est vraie.

!
"NON" : il inverse l'expression donnée.

Opérateurs de comparaison

==
"Égal à ..." (exactement le même).

!=
"Non égal à ..." (différent, de quelque façon que ce soit).

<
"Inférieur à ...".

<=
"Inférieur ou égal à ...".

>
"Supérieur à ...".

>=
"Supérieur ou égal à ...".

Définitions

Variable
Association d'un nom à une valeur. Elle est déclarée via un mot-clé et peut être de type texte, nombre, booléen, etc.

Boucle
Répétition d'un bloc de code tant que la condition spécifiée est valide.

Fonction
Ensemble d'instructions qui effectuent une tâche. Une fonction peut être appelée plusieurs fois dans un programme.

Tableau (array)
Bloc de mémoire où des données de même type sont rangées côte à côte.

Programmation Orientée Objet
Modèle qui consiste à définir et faire interagir des éléments appelés "objets".

Objet
Représentation qui se rattache au monde physique : un livre, une page de livre, une lettre.

Bonnes pratiques

- ✅ Nommer les variables, les classes et les fonctions de manière explicite.
- ✅ Indenter son code pour ne pas se perdre.
- ✅ Contrôler l'accès aux variables et aux fonctions.
- ✅ Initialiser une variable au moment de la déclaration, si sa valeur est connue.

Erreurs classiques

- ❌ Faire des fonctions trop longues qui font trop de choses.
- ❌ Utiliser le mauvais type de variable.
- ❌ Créer une boucle infinie : si la condition est toujours vraie, le programme ne s'arrêtera jamais !
- ❌ Ne pas gérer les exceptions qui peuvent survenir lors de l'exécution du programme.