Présentation de l'UE

Objectifs :

- Enseigner les principes fondamentaux de la programmation impérative
- En Java mais sans l'aspect orienté objet (car Java est bcp utilisé et similaire à d'autres langages de programmation)
- Avoir acquis les notions sur lesquelles repose le reste du cursus

> 🔥 **Différence programmation impérative et programmation orientée objet**
>
> programmation impérative : suite d'instructions avec des types itératifs et des types + compliqués (chaînes, tableaux etc)
> la façon dont les types + compliqués sont élaborés est assez simple et pas forcément optimisée
> => petit nombre d'outils à disposition dans la programmation impérative --> limites
>
> programmation orientée objet : pour des programmes qui manipulent + de choses, fournit un cadre qui permet d'automatiser des contraintes
> => + compliqué à mettre en oeuvre mais pour des projets d'une ampleur un peu plus grande

Finalité de l'UE INF1
➡️ assimiler les principes de la programmation impérative

Comment ❓

- programme syntaxiquement correct
- qui fait ce qu'on attend
- d'une manière raisonnable (optimisation)

Outils pour faire un programme qui fonctionne :

- feuille pour réfléchir avant
- éditeur / IDE : *Exemples : vscodium, eclipse (+ compliqué)*

Qu'est-ce que l'informatique ?

L'informatique est la ""**science** du traitement de **l'information**"

- science : principes, pas un art, progrès, on ne connaît pas tout

Représentation de l'information en mémoire

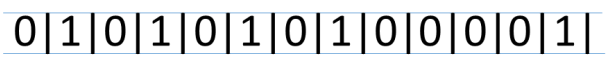
On ne peut agir qu'en jouant avec du courant électrique : on distingue 2 états = courant/pas de courant = **binaire**

(dans les années 70 : ordinateurs à 3 états : pas, un peu, beaucoup)

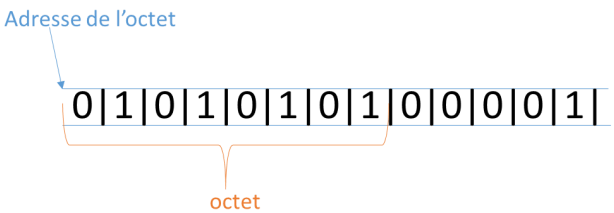
Le + **petit élément d'information** = binary unit (unité d'information binaire) = **bit** qui vaut soit 0 soit 1

A partir des bits, on peut faire des paquets de bits => **8 bits (octet)** = 1 **byte**

Mémoire d'un ordinateur peut être représentée comme :



On peut ensuite faire des paquets :



(on attribue une adresse à l'octet pour savoir où il se situe dans la mémoire)

1024 octet = 1 Ko (Kilo-octet : K $\neq$ k = kilo = 10^3)
2^{10} = 1024
1024^2 = 1 Mo
1024^3 = 1 Go
1024^4 = 1 To (ordre de grandeur des disques durs actuels)

En base 10	En base 2
0	0
1	1
2	10
3	11
4	100
...	
9	1001
10	1010
11	1011
...	
21	10101
...	
99	1100011
100	1100100
...	
255	11111111

Avec 8 chiffres, on peut stocker 255 valeurs (ce qui est assez pour stocker des chiffres, des minuscules, majuscules etc.) donc historiquement on a gardé 2^3 = 8 bits
Pour convertir de la base 10 à 2 : site

Mémoire stocke l'information, microprocesseur agit pour calculer dans la mémoire --> on arrive à la notion de programme pour que la mémoire et le microprocesseur interagissent ensemble

Notion de programme

code = suite d'instruction qui fait quelque chose
algorithme = suite d'actions pour résoudre correctement un problème => général

=> pas de code buggé (juste une suite d'instruction) mais il y a des algorithmes buggés (qui ne résolvent pas correctement un problème)

l'algorithme est ensuite décliné en **programme** = suite d'instructions (dans un langage particulier) conformes à un algorithme

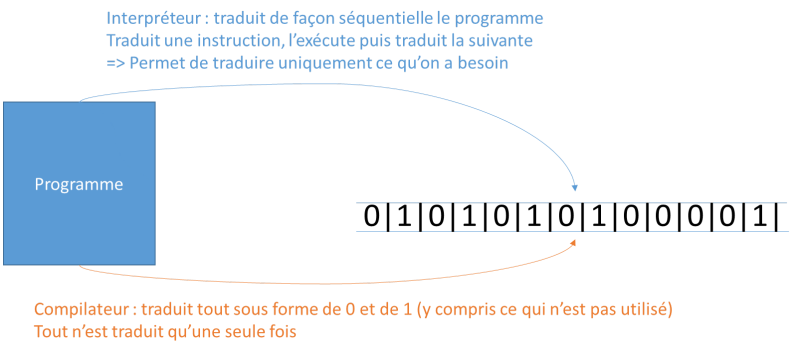
Ce qui est compliqué --> faire un algorithme
Passer d'un algorithme à un programme --> + technique (connaître le langage de programmation etc)

En fonction de l'algorithme => 1 langage de programmation peut être favorisé par rapport à un autre

Executer un programme

Prendre une suite d'action sous forme de 0 et de 1 et exécuter la fonction du programme

Le **langage de programmation** permet de fournir des instructions de haut niveau (pas des 0 et des 1 directement)

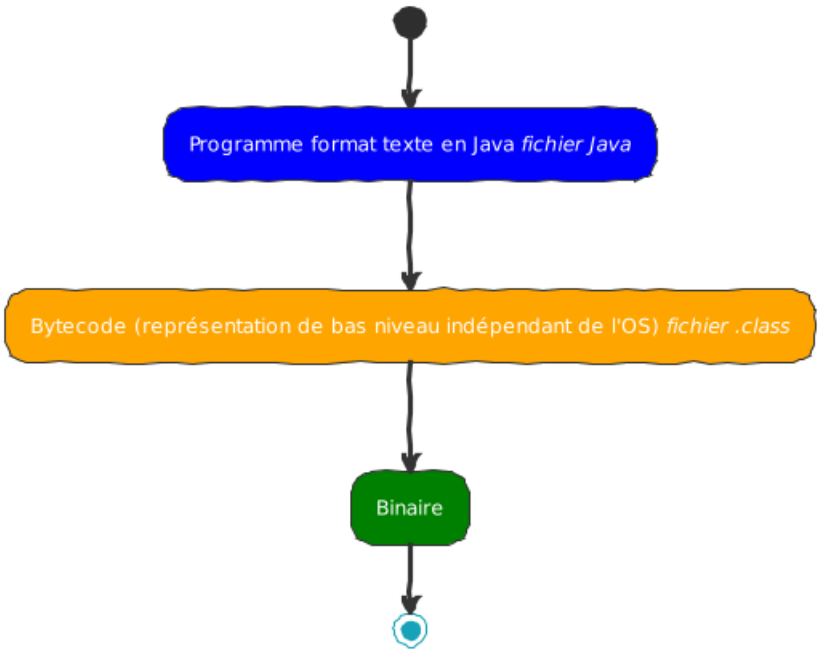


Le résultat de la traduction (**compilateur**) dépend de l'ordinateur, et notamment du système d'exploitation
Donc => le même programme doit être compilé 1 fois pour Windows, 1 fois pour MacOS

Pour **l'interpréteur**, à chaque nouvelle exécution du programme on relance la traduction, alors que ce n'est pas le cas avec le compilateur

Les langages avec un interpréteur sont beaucoup plus portables (dépend moins de l'OS) qu'avec compilateur

Java fait la **compilation et l'interprétation**



Le fichier `.class` est compris par l'ordinateur, mais on n'y touche jamais

Noms de programmes en Java commencent toujours par une majuscule
Fonction `Main` : fonction exécutée par le programme, tout ce qui se trouve dans les accolades est exécuté de manière séquentielle (d'abord une instruction, puis l'autre etc.)

Variable

variable = zone en mémoire qui contient une valeur et a un nom
Caractérisé par :

- son **nom** : `longueur_cote` ou `Longueur_Cote` par exemple
- son **type** (entier, nombre décimal, booléen)
- sa **valeur**

3 étapes pour une variable :

1. **Déclaration**
   On donne nom et type
   L'ordinateur trouve une zone mémoire et met à jour la liste des variables qu'il connaît
   Exemple : déclarer une variable 'riri' qui est un entier : `int riri;`

2. **Affectation** : mettre une valeur dans la variable
   On utilise un opérateur d'affectation `=` qui fait 2 vérifications :

- est-ce que la variable existe ?
- est-ce que le type de la variable est le même que le type de l'affectation ?
  Par exemple : `riri = 23` (asymétrique)
  La première fois qu'une affectation est effectuée : initialisation

3. **Utilisation** : remplace le nom de la variable par sa valeur

Variables booléennes

Peut prendre 2 valeurs :

- True (T, qu'on note 1)
- False (F, qu'on note 0)

Le 0/1 est purement conventionnel

Pour obtenir un booléen :

- directement
- en comparant 2 valeurs
  - vrai : même valeurs `==`
  - faux : valeurs différentes `!=`
- en comparant 2 nombres
  - mêmes comparaisons que tout à l'heure et comparaisons $< > \leq \geq$
  - Pour $\leq$, on utilise `<=`
  - Pour $\geq$, on utilise `>=`

Opérations entre booléens

- `NOT`

a	not(a)
F	T
T	F

- `AND`

a	b	a AND b
F	F	F
F	T	F
T	F	F
T	T	T

- `OR`

a	b	a OR b
F	F	F
F	T	T
T	F	T
T	T	T

En Java :

- ET : `&` ou `&&`
- OU : `|` ou `||`

Différence entre les simples opérateurs et les doubles opérateurs : les simples opérateurs (`&`, `|`) évaluent des deux côtés de l'opération.

Les doubles opérateurs évaluent du côté gauche d'abord et en fonction du resultat du côté droit.

Par exemple :

```
A && B
// Si A est Faux, alors peu importe la valeur de B, l'expression est fausse donc B n'est pas évalué
// Si A est Vrai, alors on n'est pas sûrs donc on évalue B
A || B
// Si A est Vrai, alors peu importe la valeur de B, l'expression est vraie donc B n'est pas évalué
// Si A est Faux, alors on évalue B
```

Opérateurs conditionnels

si `condition Booléenne` (sous-entendu : est **vrai**)
alors `{`
`bloc d'instruction`
`}`
sinon `{`
`bloc d'instruction`
`}`

On utilise les opérateur `if` et `else`.
Il est possible d'imbriquer ces opérateurs :

```
if (A > B){
        if (A > 10) System.out.println("premier choix");
        else if (B < 10) System.out.println("deuxième choix");
}
else {
        if (A==B) System.out.println("troisième choix")
        else System.out.println("quatrième choix")
}
```

Le `else if` présente un autre type de choix, mais il est non obligatoire.
C'est-à-dire que si on a que des `if` et `else if` il est possible de ne rentrer dans aucune des conditions. Alors que si on a un `else`, ce sera forcément la condition par défaut (dans l'exemple : la quatrième choix, donc il n'existe aucune valeur de A et B pour lesquelles le programme n'affiche rien)

Boucles

--> Servent à faire des itérations : pour écrire de manière générique des calculs répétitifs par exemple.

- Pour : `for`
- Tant que : `while`
- Faire tant que : `do while`

Il est possible d'imbriquer des boucles, comme pour les `if else`

Boucle for

Equivalent au `pour`
On a besoin : d'une **condition de départ (valeur initiale)**, d'un **test de continuation** (pour voir si on arrête la boucle ou pas), et du pas de la boucle (ou **incrément**)

Exemple de boucle `for` :

```
for(int i = 0 ; i<12 ; i = i + 1)
//  départ      test    pas de la boucle


// équivalent à
int i;
for (i =0; i<12 ; i++)
```

Dans le deuxième programme, la variable i n'est pas locale à la boucle, contrairement au premier

Boucle while

Équivalent au `tant que`
C'est-à-dire que tant que la condition en entrée est vraie, on continue la boucle

Exemple :

```
while (a<10){
        System.out.println(a)
        a++ // a augmente de 1 à chaque itération donc finit par être supérieur à 10 => on sort de la boucle
}
```

Danger

Attention au problème des boucles infinies !
Il faut bien prévoir qu'il doit être possible de sortir de la boucle sinon elle ne s'arrêtera pas. Par exemple, dans le code suivant :

```
int a = 0;
while (a<10){
        System.out.println(a)
```

On ne sort jamais de la boucle car a vaut toujours 0 !

Boucle do while

Similaire à la boucle while mais on présente les instructions avant :

```
do {
        System.out.println(a)
        a++
} while (a<10)
```

Donc, on passe toujours **au moins une fois dans la boucle** => ce n'est pas le cas pour une boucle while où si la condition d'entrée est fausse *(exemple : a=1 et condition d'entrée : a>10)* on ne rentre pas dans la boucle

Affecter une valeur à une variable

Affecter directement :

```
int a = 3;
a = 5+3;

int b = 12;
```

Incrémentation `a=a+2` remplace la "vieille" valeur d'une variable par une nouvelle valeur qui dépend de la vieille
`a = a+2` ⇔ `a+=2`
Fonctionne aussi pour les décrémentations : `a-=5`

Opérateur `++`

- `++a`
  1. on incrémente `a`
  2. on renvoie la valeur de `a`
- `a++`
  1. on renvoie la valeur de `a`
  2. on incrémente `a`

Donc :

```
a=7;
b = ++a;
// a = 8
// b = 8
// équivalent à :
// b=a;
// a=a+1;
```

```
b = a++;
// a = 8
// b = 7
// équivalent à :
// a=a+1;
// b=a;
```

Le type 'double'

Le type `double` représente un décimal (similaire au type `float` mais plus précis)
Pour initialiser une variable avec ce type :

```
double variable;
// ou en lui donnant une valeur
double variable = 3/5;
```

Eviter les erreurs en faisant des calculs sur des nombres décimaux/fractions

Si on veut effectuer le calcul $\frac{5}{9} \times x$, en écrivant juste :

```
double resultat = 5/9 * x
```

On obtient 0, car 5/9 est considéré comme un entier (qui vaut 0)
On peut donc faire :

```
double resultat = (double) 5 / (double) 9
// ou bien
double resultat = 5.0/9.0 //en ajoutant .0, on transforme automatiquement en double
```

Fonctions

= Valeur donnée à une suite d'instructions
utile quand on doit réaliser la même tâche plusieurs fois

1. Déclaration/définition de la fonction

```
public static TypeRenvoi nomDeLaFonction (TypeArgument NomArgument, TypeArg2 NomArg2, ...) {
        // corps de la fonction
}
```

Déclaration des fonctions

- TypeRenvoi : int, float, boolean, string, void (si la fonction ne renvoie rien) ...
- les arguments peuvent être de même type ou de type différent
- il peut ne pas y avoir d'arguments donc `nomDeLaFonction ()`
- la fonction `main` se présente sous cette syntaxe :

```
public static void main(String[] args) {
// le code
}
- il est toujours mieux d'indenter le code (avec une tabulation) une fois à l'intérieur des `{}`
```

Exemples de fonction

```
public static int reponseUltime(){
        return 42;
}

public static void etudiant(int numEtudiant, String prenom, String nom){
```

```
        //code
}
```

2. Appel de la fonction : permet de réaliser une tâche

```
nomDeLaFonction(TypeArg1 Arg1, TypeArg2 Arg2) // on doit avoir autant de valeurs données que d'arguments dans la définition de la fonction
```

`return`	`print`
renvoie une valeur	ne renvoie rien
n'affiche rien	affiche
est réutilisable	à usage unique

Intérêt de pouvoir réutiliser `return`

Exemple de fonction

```
public static float fonction(int x){
        return 2.3*x;
}

float valeur = fonction(7);
System.out.println("Le résultat est" + valeur);
System.out.println("Le double du résultat est"+ 2*valeur);
```

Grâce à cette fontion, on attribut le résultat de sortie à une variable qu'on peut ensuite réutiliser pour effectuer d'autres calculs.
Ce n'est pas possible avec un `print`

Fonction équivalente :

```
public static fonction(int x){
        System.out.println("Le résultat est" + 2.3*x);
}

fonction(7); // on ne peut pas réutiliser cette valeur !
```

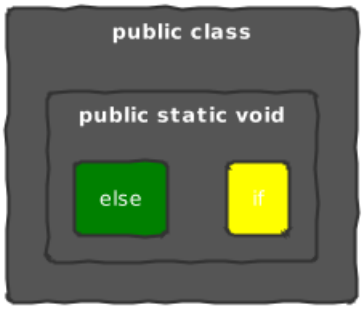
Notion de portée de variables

Notion de bloc d'instruction

- plus gros bloc d'instruction : `public class` ? `public static void main(String[] args)`
- Peuvent contenir des fonctions
- Peuvent contenir des opérateurs conditionnels
- etc.

Les blocs peuvent être :

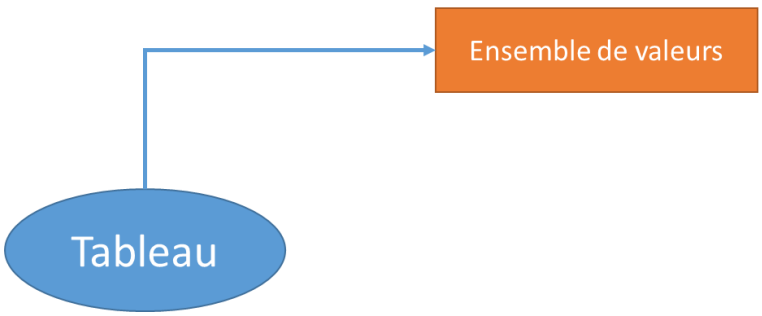
- disjoints bloc if et bloc else
- imbriqués
- **jamais** chevauchants



Principes de visibilité des variables

1. Les variables sont déclarées dans un bloc => on ne peut pas déclarer 2 variables avec le même nom dans 1 **même bloc** (mais dans des blocs disjoints ok)
2. Les variables déclarées dans un bloc sont visibles dans tous ses sous-blocs
   => les variables **globales** sont les variables déclarées dans le plus gros bloc (à l'inverse des variables locales)
   => une variable d'un sous-bloc peut masquer une variable d'un super bloc (ne pas faire ça)
3. les variables déclarées dans un bloc ne sont pas visibles à l'extérieur de ce bloc
   => les variables globales sont visibles partout

Tableau



- Toutes les valeurs ont le même type
- La **taille** prédéfinie est fixe (éventuellement vide)
- Ensemble ordonné rangé de façon **consécutive** : on les repère par leur position
- L'attribut **lenght** contient le nombre d'éléments dans le tableau
- La **position** des éléments est numérotée de **0** à **lenght -1**

3 étapes pour un tableau :

1. Déclaration

```
typeDesElements[] nomTableau
```

2. Allocation

```
nomTableau = new typeDesElements[5] // 5 est la taille du tableau
```

- réserve la place en mémoire
- met l'adresse de la zone comme valeur de `nomTableau` (donc `nomTableau` ne contient pas un tableau, mais l'adresse où il est situé)

3. Initialisation

Affecter une valeur à chaque élément

```
for(int i=0 ; i< nomTableau.lenght ; i++){
        nomTableau[i] = 0 // 0 pour toutes les valeurs du tableau
}
```

Différentes manières de réaliser ces 3 étapes

```
int[] nomTab = int[5];
for(int i=0 ; i< nomTableau.lenght ; i++){
        nomTableau[i] = 0 // 0 pour toutes les valeurs du tableau
}

// ou avec toutes les étapes d'un coup

int[] nomTab={4, -8, 12, 42, 67} // on a directement toutes les valeurs du tableau
```

Utiliser un tableau

- `nomTab.lenght` : renvoie la longueur du tableau
- `nomTab[2]` : renvoie la **troisième** (car on commence à 0) valeur du tableau (dans l'exemple précédent : 12)

Copier un tableau

```
int[] newTab={6,-4,7};
// équivalent à
int[] newTab = new int[3]; // allocation
newTab[0]=6;    // indentation
newTab[1]=-4;   // indentation
newTab[2]=7;    // indentation

int[] autreTab = newTab;
```

`autreTab` et `newTab` pointent sur la même zone en mémoire, donc 2 variables pour la même zone.

```
for(int i =0 ; i<newTab.length ; i++){
        System.out.print(newTab[i]+" "); // on affiche succesivement les valeurs du tableau avec un epsace entre chaque
}

/* OUTPUT :
6 -4 7
*/

System.out.println(); //nouvelle ligne

for(int i =0 ; i<autreTab.length ; i++){
        System.out.print(autreTab[i]+" ");
}

/* OUTPUT :
6 -4 7
*/
```

On a l'impression qu'on manipule 2 tableaux différents, mais il s'agit en fait de la **même zone en mémoire**.

Modifions les valeurs dans la zone en mémoire pour le vérifier :

```
autreTab[1] = 42;
for(int i =0 ; i<newTab.length ; i++){
        System.out.print(newTab[i]+" ");
}

/* OUTPUT :
6 42 7
*/
```

Copier un tableau

Affecter une variable tableau à une nouvelle variable comme effectué précedemment ne suffit pas pour copier/dupliquer un tableau. En effet, toute modification sur 1 des 2 tableaux affectera aussi l'autre.

Donc, on fera plutôt :

```
int[] autreTab = new int[newTab.length];
for(i=0,i<newTab.length;i++){
        autreTab[i]=newTab[i]
}
```

Types de recopies

Par référence :

```
int[] autreTab = newTab;
```

Ne copie que l'adresse du tableau

Par valeur :

```
int[] autreTab = new int[newTab.length];
for(i=0,i<newTab.length;i++){
        autreTab[i]=newTab[i]
}
```

Copie les valeurs contenues dans le tableau

Comparaisons de tableaux

```
int[] t1={6,-4,7};
int[] t2=t1;
int[] t3={6,-4,7};
int[] t4={6,42,7};
```

Pour voir si 2 élements `tx` et `ty` sont les mêmes (même adresse) : opérateur `tx == ty`
Pour voir si 2 objets `tx` et `ty` ont les mêmes valeurs : `tx.equals(ty)`

Tableau 1	Tableau 2	`tx == ty`	`tx.equals(ty)`
t1	t2	T	T
t1	t3	F	T
t2	t3	F	T
t1	t4	F	F

```
int[] t1={6,42,7};
int[] t2=t1;
int[] t3={6,-4,7};
int[] t4={6,42,7};
```

Tableau 1	Tableau 2	`tx == ty`	`tx.equals(ty)`
t1	t2	T	T
t1	t3	F	F
t2	t3	F	F
t1	t4	F	T

Parcours intégral du tableau

```
for (int i = 0 ; i < tab.length ; i++){
        System.out.println(tab[i]);
}
```

Calcul de la somme des valeurs d'un tableau

```
int somme = 0;
for(int i = 0; i< tab.length ; i++){
        somme+=tab[i]
}
```

Trouver des valeurs dans un tableau

Trouver la **plus grande valeur**

```
int plusGrandeValeur =  tab[0]; //on ne prend pas 0 car si on a un tableau avec uniquement des valeurs négatives : pb
for(int i=0 ; i<tab.length ;i++){
```

```
        if(tab[i]>plusGrandeValeur) plusGrandeValeur = tab[i]
}
```

Trouver la première valeur **positive** (quand on sait qu'il y a 0 dans le tableau)

```
int i = 0;
while(tab[i]<0){
        i++;
}
```

Trouver la première valeur **positive** (quand on sait **pas** s'il y a 0 dans le tableau)

```
int i=0;
while(i<tab.length && tab[i]<0){
        i++;
}
```

Interagir avec l'utilisateur

On peut interagir avec l'utilisateur par la console avec :

```
import java.util.Scanner; // pour utiliser le scanner

public class ma_classe {
        public static void main(String[] args) {
                Scanner sc = new Scanner(System.in); // on créé un scanner
                System.out.println("Entrer un nombre entier >= 1"); // on demande à l'utilisateur d'entrer un nombre
                int nombre = sc.nextInt(); // on prend un nombre entier en entrée par l'utilisateur

                /*

                LE CODE SE TROUVE ICI


                */


                sc.close(); // on ferme le scanner

        }
```