JAVA ALGO

PROGRAMME

- Structure du langage java
- Utilisation des primitives
- Les enveloppeurs (wrappers)
- Les opérateurs
- Condition
- Les boucles
- Les méthodes (utilisation et surcharge)
- Les tableaux

STRUCTURE DU LANGAGE JAVA

Au niveau syntaxe, Java est un langage de programmation qui ressemble beaucoup au langage c++

(**java est sensible a la casse**)

Toute fois quelques simplifications ont été apportées à java pour des raisons de sécurité et d'optimisation.

Dans cette partie nous ferons une présentation succincte des types primitifs, les enveloppeurs,

déclarations des variables, le casting des primitives,

les opérateurs arithmétiques et logiques, les

structures de contrôle (if, swich, for et while)

Java?

- Programmation orientée Objet Java
- Objet et Classe
- Héritage et accessibilité
- Polymorphisme
- Collections
- Notions de Design Patterns
- Exceptions et Entrées Sorties
- Interfaces graphique AWT et SWING
- Accès aux bases de données
- Threads et Sockets

Langage de programmation orienté objet (Classe, Objet, Héritage, Encapsulation et Polymorphisme)

Avec java on peut créer des application multiplateformes. Les applications java sont portables. C'est-à-dire, on peut créer une application java dans une plateforme donnée et on peut l'exécuter sur n'importe quelle autre plateforme.

Le principe de java est : Write Once Run Every Where

Open source: On peut récupérer le code source de java. Ce qui permet aux développeurs, en cas de besoin, de développer ou modifier des fonctionnalités de java.

La méthode main

Méthode nécessaire pour executer le programme.

Attention la signature (ligne de déclaration de la méthode) doit être exactement : **public static void main(String[] args)**

sinon ne peut pas s'executer

A créer lors de la création de la classe (case à cocher sous éclipse)

JSDK : Java Standard Developpement Kit, pour développer les application

```
public class PremierProgramme {

public static void main(String[] args) {
System.out.println("First Test");
}
}
```

Java est utilisé pour créer :

- Des applications Desktop
- Des applets java (applications java destinées à s'exécuter dans une page web)
- Des applications pour les smart phones
- Des applications embarquées dans des cartes à puces
- Des application JEE (Java Entreprise Edition)
- Pour créer une application java, il faut installer un kit de développement

LE MODE CONSOLE: CLAVIER ET AFFICHAGE

```
import java.util.Scanner;
public class Test {
public static void main(String[] args) {
System.out.println("Bonjour... ?");
Scanner clavier =new Scanner(System.in);
String nom = clavier .nextLine();
System.out.println("Bonjour "+nom+" !");
```

Les primitives

- Java dispose des primitives suivantes :

Primitive	*Étendue*	*Taille*
char	0 à 65 535	16 bits
byte	-128 à +127	8 bits
short	-32 768 à +32 767	16 bits
int	-2 147 483 648 à + 2 147 483 647	32 bits
long		64 bits
Float	de ± 1.4E-45 à ± 3.40282347E38	32 bits
double		64 bits
boolean	true ou false	1 bit
void	-	0 bit

UTILISATION DES PRIMITIVES

Les primitives sont utilisées de façon très simple. Elles doivent être déclarées, avec une syntaxe similaire, par exemple :

```
int i;
char c;
boolean fini;
```

Les primitives peuvent être initialisées en même temps que la déclaration.

```
int i = 12;
char c = 'a';
boolean fini = true;
```

UTILISATION DES PRIMITIVES

Comment choisir le nom d'une variable:

Pour respecter la typologie de java, les nom des variables commencent toujours par un caractère en minuscule et pour indiquer un séparateur de mots,on utilise les majuscules.

Exemples:

int nbPersonnes;

String nomPersonne;

Les variables n ont pas de valeurs par défaut

LE CAST(CONVERSION) DES PRIMITIVES

Il consiste à effectuer une conversion d'un type vers un autre type.

Le casting peut être effectué dans deux conditions différentes

- Vers un type plus général. On parle alors de *sur-casting* ou de *sur-typage*.
- Vers un type plus particulier. On parle alors de *sous casting* ou de *sous-typage*.

LE CAST(CONVERSION) DES PRIMITIVES

Sur-casting :

Le sur-casting peut se faire implicitement ou explicitement.

Exemples :

```
int a=6; // le type int est codé sur 32 bits
long b; // le type long est codé sur 64 bits
```

Casting implicite :

```
b=a;
```

Casting explicite

```
int c=(int)b;
```

LE CAST(CONVERSION) DES PRIMITIVES

Sous-Casting : Le sous-casting ne peut

se faire qu'explicitement:

long a =10;

int b= (int)a;

On peut utiliser les classes wrappers pour effectuer des conversions

int a = Integer.parseInt(‘’2’’);

double b= Double .parseDouble(‘’2.5’’);

ACCOLADES

- Chaque méthode commence et fini par { … }
- De même pour les classes
- Source d’erreur au début : à vérifier

LES ENVELOPPEURS (WRAPPERS)

Les primitives sont enveloppées dans des objets appelés enveloppeurs (wrappers). Les enveloppeurs sont des classe

Classe	*Primitive*
Character	**char**
Byte	**byte**
Short	**short**
Integer	**int**
Long	**long**
Float	**float**
Double	**double**
Boolean	**boolean**

LES OPÉRATEURS

Les opérateurs arithmétiques à deux opérandes:

+ : addition

- : soustraction

***** : multiplication

/ : division

% : modulo (reste de la division euclidienne)

Les opérateurs arithmétiques à deux opérandes

(Les raccourcis)

x = x + 4; ou **x+=4;**

z = z * y; ou **Z*=y;**

v = v % w; ou **v%=w;**

LES OPÉRATEURS

Les opérateurs relationnels:

== : équivalent

< : plus petit que

> : plus grand que

<= : plus petit ou égal

>= : plus grand ou égal

!= : non équivalent

Les opérateurs d'incrémentations et de décrémentation:

++ : Pour incrémenter **(i++ ou ++i)**

-- : Pour décrémenter **(i–- ou --i)**

LES OPÉRATEURS

a++ et a-- : incrément et décrément (aussi ++a ou –a, petite différence)

a+= 2 equivalent à a=a+2 (sur entier comme sur chaine)

```
int a =5;
int b;
b =a++;
System.out.println(a);System.out.println(b);
b=++a;
System.out.println(a);System.out.println(b);
```

Donne 6, 5, 7, 7

BOUTS DE CODES CONTENANT DES TRUCS UTILES

System.***out.println(a);System.out.println(b);***

String x = "a"; String y ="A";

if (x.equals(y))

System.***out.println("une fois vrai");***

if (x.equalsIgnoreCase(y)) System.*out.println("deux fois vrai");*

CONDITION

Il existe trois manières de noter la condition:

le if…..

Le condition ternaire

Le switch case

Les opérateurs logiques

&& Et (deux opérandes)

|| Ou (deux opérandes)

! Non (un seul opérande)

L'INSTRUCTION CONDITIONNELLE IF

La syntaxe de l'instruction **if** peut être décrite de la façon suivante:

```
if (expression) instruction;
```

ou :

```
if (expression) { instruction1; instruction2;}
```

L'instruction conditionnelle else

```
if (expression) {
instruction1;
}
else {
instruction2;
}
```

L'INSTRUCTION CONDITIONNELLE IF

Les instructions conditionnelles imbriquées

Java permet d'écrire ce type de structure sous la forme :

```
if (expression1) {
bloc1;
}
else if (expression2) {
bloc2;
}
else if (expression3) {
bloc3;
}
else {
bloc5;
}
```

CONDITION TERNAIRE

L'opérateur à trois opérandes ?:

condition ? expression_si_vrai : expression_si_faux

exemple : x = (y < 5) ? 4 * y : 2 * y;

Equivalent à :

```
if (y < 5)
x = 4 * y;
else
x = 2 * y;
```

CONDITION SWITCH CASE

```
import java.util.Scanner;
public class Test {
public static void main(String[] args) {
System.out.print("Donner un nombre:");
Scanner clavier=new Scanner(System.in);
int nb=clavier.nextInt();
switch(nb){
case 1 : System.out.println("Lundi");break;
case 2 : System.out.println("Mardi");break;
case 3 : System.out.println("Mercredi");break;
default :System.out.println("Autrement");break;
}}}
```

LA BOUCLE FOR

La boucle **for** est une structure employée pour exécuter un bloc d'instructions un nombre de fois en principe connu à l'avance. Elle utilise la syntaxe suivante :

```
for (initialisation;test;incrémentation) {
instructions;
}
```

Exemple :

```
for (int i = 0; i < 10;i++) {
System.out.println("I="+i);
}
```

Branchement au moyen des instructions break et continue

break:

```
int x = 10;
for (int i = 0; i < 10; i++) {
x--;
if (x == 5) break;
}
System.out.println(x);
```

continue:

```
for (int i = 0; i < 10; i++) {
if (i == 5) continue;
System.out.println(i);
}
```

L ’instruction While

```
while (condition){
BlocInstructions;
}
```

Exemple :

```
int s=0;int i=0;
while (i<10){
s+=i;
i++;
}
System.out.println("Somme="+s);
```

L ’instruction do ..while

```
do{
BlocInstructions;
}
while (condition);
Exemple :
int s=0;int i=0;
do{
s+=i;
i++;
}while (i<10);
System.out.println("Somme="+s);
```

LES MÉTHODES

Une méthode a une signature et un contenue

Ex: signature → void afficheBonjour()

→ {System.out.println(« bonjour»);}

une méthode peut avoir des paramètres en entrée ou pas

void afficheBonjour() ou void afficheBonjour(String str)

- une méthode peut avoir une valeur de retour ou pas

```
static void add(int a, int b){
int somme =a+;System,out,println(« somme»);}
static int addition(int a, int b){
int somme =a+;return somme;}
```

Il est intéressant de découper son code en méthodes cela permet d avoir un code clair et réutilisable

EXERCICES

Ecrire deux méthodes :

static void m1(){sop(« je suis m1 »);} //sop=System.out.println
static void m2(String str){sop(« je suis m2 »+str);}
static String m3 (String x){sop(« je suis m3 »); return « a bientôt »+x;}

Dans le main :
Appeler m1 : m1();
Appeler m2 : m2(«test »);
Appeler m3 : String s = m3(«test »); // on veut pouvoir utiliser le retour
Ou : m3(« toto »)// on veut juste ce qu'affiche la méthode, sans exploiter le retour

TP : MINI CALCULETTE

Exemple: A intégrer au menu de notre première classe

Signature : static void calcul()

Ce que ça fait :

Une boucle infinie qui propose de faire une opération ou de quitter.

Demande de rentrer +, Add, - , Moins, *, Mult,/, Div ou Quitter.

Une fois l'opération choisie, demande deux opérande, renvoi le résultat et repropose le menu.

Contrainte : utiliser une méthode de signature static int add(int x, int y) (idem pour les autres opérations)

La méthode Swap

Codons une méthode qui permet d intervertir des valeurs :

```
…main(…)
{
    int a = 10;
    int b =15;
    swap (a,b);
}
```

après l exécution de la méthode swap contient a et b n ont pas changé

Mais ça ne marche pas !!!

Solutions ? (oui, mais en objet, à voir plus tard)

POURQUOI ?

- Passage par référence : n'existe pas en java
- Passage par valeur / par copie : c'est ce qui se passe (n'utilise pas les valeurs des variables contenues dans la stack mais en fait des copies, modifie ses copies, mais à la fin de la méthode les variables initiales n'ont pas été modifiées

SURCHARGE DE MÉTHODE

- Une méthode peut avoir plusieurs surcharges
- Il s agit de la même méthode avec des versions différentes
- La surcharge est définit par les paramètres en entrée et pas la valeur de retour

SURCHARGE DE MÉTHODE

- Idée : on veut plusieurs méthode de même nom avec des signatures différentes

On peut mettre des nombres et/ou des types de paramètres différents, ce qui permet d'accéder à des méthodes similaires dans des cas différents.

On peut avoir autant de surcharge que l'on veut.

```
static add (int a, int b){return a+b);
static add (int a, int b, int c){return a+b+c);
```

- On ne peut pas avoir deux méthodes différentes de même signature !
- La surcharge demande d'avoir des paramètres différents

EXERCICE GUIDÉ

- Coder les méthodes pour avoir ce main :

```
main() {   //raccourci pour le tableau, il faut la vraie signature en vrai
        affiche ();
        affiche (« Hello »); //ligne marquée
        affiche (1,2);
        affiche (1, « toto »);
        affiche (« toto », 1);
        String x =affiche (« Bonjour »);
//il va y avoir un pb, voir p.suivante
```

CORRECTION


```
static void affiche (){System.out.println("Rien à afficher");}
static void affiche (int a, int b){System.out.println(a+" "+b);}
static void affiche (String str){System.out.println(str);}
static void affiche (int a, String b){System.out.println(a+b);}
static void affiche (String a, int b){System.out.println(a+b);}
static String affiche (String str){System.out.println(str);}


// soucis : la JVM a deux choix pour la ligne marquée : celle définie en 3eme ou celle définie en dernier. (on peut parfaitement
utilisé une méthode renvoyant quelque chose en ignorant son retour.
Ca ne peut pas compiler tel quel !
```

LES TABLEAUX

- Un tableau est une liste d éléments consécutifs de même type
- Il a une taille fixe contrairement a une collection qui est un tableau a taille dynamique
- 2 caractéristiques: sa taille et son type

TABLEAUX DE PRIMITIVES

Tableaux de primitives:

Déclaration :

Exemple : Tableau de nombres entiers

```
int[] liste;
```

liste est un handle destiné à pointer vers un tableau d'entier

Création du tableau

```
liste = new int[11];
```

Manipulation des éléments du tableau:

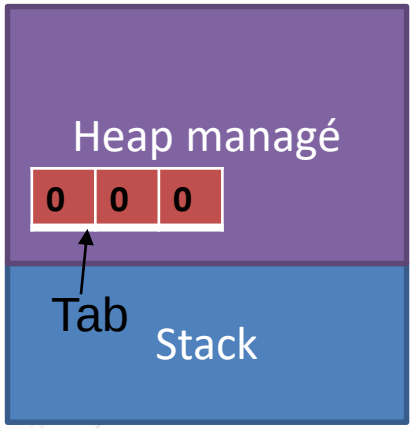
```
liste[0]=5; liste[1]=12; liste[3]=23;
for(int i=0;i<liste.length;i++){
System.out.println(liste[i]);
}
```

PAS À PAS, INITIATION AUX TABLEAUX

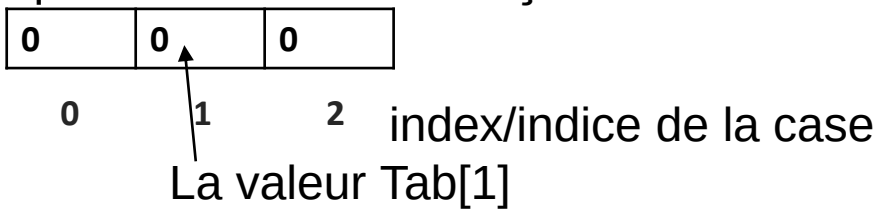
int[] Tab = new int[3]; // crée un tableau d'entiers de 3 éléments, initialement à 0 (valeur par défaut)

- Coté mémoire :

La variable Tab est dans la stack, mais le tableau (ce qu'il y a dedans) est dans la heap managé



Ce que l'on a ressemble à ça :



TABLEAU

- Idée : tableau simple, plutôt que de créer plusieurs variables (par exemple 3 variables), créer une sorte de liste de variables
- Deux caractéristiques : une taille (ex : 3 éléments)- forcément défini et un type d'éléments.
- Deux façon de créer : on va les voir

Taille fixée et non redimensionnable !

```
static void testtab(){
   int[] Tab = new int[3];
   for (int i=0; i<3; i++){
      System.out.println(Tab[i]);
   }
}
```

Affiche :

0
0
0

Ou :

```
for (int i=0; i<Tab.length; i++) //ce qui nous évite d'avoir à écrire 3 en dur
System.out.println(Tab[i]);
```

Boucle for each

Ou (boucle « for each »):

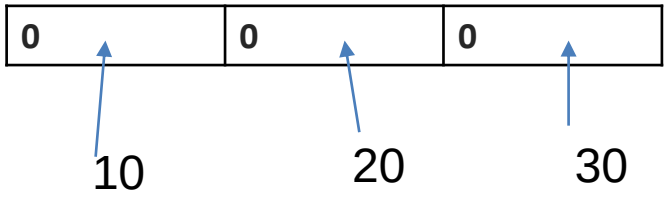
for(int e : Tab) {

System.***out.println(e);}***

- Ne permet pas de modifier ! Pas fait pour.
- Ne fait pas intervenir d'indices
- Utilisation typique : parcourir/afficher

SUITE EXEMPLE

Renseigner des valeurs




```
Tab[0]=10; // injecte 10 à la
case d'indice 0
Tab[1]=20;
Tab[2]=30;
```

Exo : écrire une méthode **static void testtableau2(){…}**
Qui demande à l'utilisateur une taille puis des valeurs pour chacune des cases
Ex : Quelle taille ? > 4 //> désigne la réponse de l'utilisateur
Quelle valeur pour la case d'indice 0 ? >3
Quelle valeur pour la case d'indice 1 ? >0
Quelle valeur pour la case d'indice 2 ? >13
Quelle valeur pour la case d'indice 3 ? >-3
Votre tableau : 3 0 13 -3

- Autre façon de coder un tableau (taille fixée automatiquement)
- ```
  static void test (){
  ```
- ```
  int [] Tab ={1,13,565,1,25,-5,0,0,2}; //affiche le tableau
  ```
- ```
  }
  ```
  ```

TABLEAUX SUITE

- Erreur d'éxécution java.lang.ArrayIndexOutOfBoundsException

```
int[] Tab = new int[3];
Tab[3]=10;
```

On a essayé d'accéder à une case qui n'existe pas.

- Création en deux temps

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

La deuxième ligne est nécessaire pour avoir un tableau effectivement existant : c'est à ce moment là que le tableau est créé dans le heap manager, et que le lien entre le nom de la variable (dans stack) et le contenu (le tableau créé dans H.M.) est fait.

SWAP TABLEAU : UNE CORRECTION

```
static void testswap(){
int a=10, b=15;
int[] Tab = new int[2];
Tab[0]=a; Tab[0]=b;
System.out.println(a+" "+b);
swapTableau(Tab);
a = Tab[0]; b= Tab[1];
System.out.println(a+" "+b);}
```

```
static void swapTableau(int[] T)
{
int temp=T[0];
T[0]=T[1];
T[1]=temp;
}
```

Pourquoi ?
Tab est une référence (dans stack) vers le tableau (idée : l'adresse d'un bâtiment, idée : Tab « pointe » vers le vrai tableau).
L'adresse est passée par copie, si on modifie la variable Tab ça ne change rien.
Par contre, le tableau lui-même est dans le Heap Managé et n'est pas copié lors de l'appel de la fonction.
Lorsque le tableau est modifié, son contenu dans le H.M. change mais pas sa référence.
On a donc bien un contenu modifié en H.M. qui reste référencé vers Tab, qui lui n'a pas changé.

Retourner plus d'une valeur

- Une méthode ne peut pas renvoyer plusieurs valeur

Par ex. On voudrait une méthode qui prend deux entiers et renvoie leur somme et leur différence. On ne peut pas renvoyer deux entiers.

Une solution pour palier à ce problème : les tableaux

Eviter la surcharge de méthodes
Tableaux à la volée

affiche(1);
affiche(1,2);
affiche(1,2,3);
affiche(10, 20, 30, 40);

Problématique : comment éviter de surcharger la méthode n fois?

```
static void affiche(int... x)
{
  for (int s : x)
     System.out.println(s);
  }
```

//transforme à la volée un ou plusieurs entiers en un tableau d'entier. Permet d'avoir n'importe quel nombre de paramètres de même type.

RÉCURSIVITÉ

- Exemple de la factorielle :
  3!= 3* 2! puis 2!=2* 1! puis 1!=1 (on peut aussi le voir en version itérative)

```
static int factorielle (int n){
if (n>1) return n*factorielle(n-1);
else return 1;

}
```

Exemple suivant : fibonacci