

BPOO - Sujet TD 1

Dut/Info-S2/M2103

Table des matières

1. Comprendre l'implantation mémoire des données
 - 1.1. Principe de stockage des données en mémoire : les variables
 - 1.2. Application : l-value ? r-value ?
 - 1.3. Importance du type
 - 1.4. Emplacement mémoire
2. Implantation mémoire des tableaux
 - 2.1. Exemple de déclaration de tableau en java
 - 2.2. Un tableau Java : une référence
 - 2.3. Représentation mémoire des tableaux
 - 2.4. L'opérateur new pour créer des tableaux
 - 2.5. Type tableau : valeurs et opérations
 - 2.6. Typage et expressions
 - 2.7. Application : représentation mémoire des tableaux
3. Les objets Java
 - 3.1. Un objet Java : une référence
 - 3.2. Représentation mémoire des objets
 - 3.3. L'opérateur new pour les objets
 - 3.4. Type objet : valeurs et opérations
 - 3.5. Application : représentation mémoire des objets
 - 3.6. Egalité d'objet : "==" et "equals()"
4. Appel de sous-programme - Passage des paramètres
 - 4.1. Règle de passage des paramètre
 - 4.2. Exemple 1
 - 4.3. Exemple 2
 - 4.4. Exemple 3



Une grande partie des informations données ici s'applique au langage **Java** uniquement.

PreReq	S1, notion de type, variables simples, tableaux.
ObjTD	Comprendre l'implantation mémoire des données.
Durée	1 séance de 1,5h

1. Comprendre l'implantation mémoire des données

1.1. Principe de stockage des données en mémoire : les variables

Une variable permet d'associer

- un identificateur
- un type
- un emplacement mémoire qui contient une valeur



Quel que soit le langage, on parle pour une "variable" (sens dépendant du contexte) de :

- **r-value** : right value, expression qui a une valeur qui "n'a pas d'adresse", tout ce qui peut se mettre à droite d'une affectation (=). Ex : la valeur 10.
- **l-value** : left value, expression dont la valeur permet d'accéder à un emplacement mémoire, tout ce qui peut se mettre à gauche d'une affectation (=). **Contre exemple** : la valeur 10.

Une variable permet, en utilisant l'identificateur :

- d'accéder à la valeur → aspect r-value
- de modifier le contenu de l'emplacement mémoire avec une valeur du type attendu → aspect l-value

1.2. Application : l-value ? r-valeur ?

Dans le code suivant, indiquer ci-après quand est ce que l'on s'intéresse à une r-value ou à une l-value

```
public static void main(String[] argv) {
    int i, j, k ;
    double d, e, f;
    i = 1; // ❶
    j = i + 10 ; // ❷❸
    k = j ;
    e = 10.5;
    if (e > i) { // ❹❺
        f = j + k ;
        k = (int) (f - e) ;
    }
}
```

- ❶ i ?
- ❷ j ?
- ❸ i ?
- ❹ e ?
- ❺ i ?

1.3. Importance du type

Le type d'une variable définit :

- l'ensemble des valeurs possibles que peut contenir la variable
 - `int` : codage en complément à 2 sur 4 octets
 - `double` : codage en flottant 2 sur 8 octets
 - `boolean` : sur un octet, valeurs possibles `true` et `false`
- les opérations possibles sur les valeurs
 - `int` : `+, -, /, *, %, ++, --` entre `int`
 - `double` : `+, -, /, *` entre `double`
 - `boolean` : `&&, ||, !` entre `boolean`

Voir aussi le cours S1 structures de données.

1.4. Emplacement mémoire

Quel que soit le langage, au runtime, un programme dispose de mémoire allouée par le système d'exploitation :

- un segment de mémoire pour stocker le code (segment de code),
- un segment de mémoire pour stocker les données (segment de données) découpé en trois parties : i) un sous-segment pour la pile, ii) un sous-segment pour le tas, iii) un sous-segment permanent : données globales, données statiques des fonctions.

Un programme Java ne s'exécute pas directement sur le système d'exploitation, mais dans la Machine Virtuelle Java (JVM).

Un programme Java ne "dispose" que de deux segments de données :

- un sous-segment pour la **pile** – variables automatiques,
 - il est utilisé lors des appels de sous-programmes pour stocker :

- les paramètres de la signature (en-tête),
 - les variables locales où qu'elles soient définies dans le corps du sous-programme,
 - fait partie de ce qu'on appelle le contexte de la fonction, cf. cours système,
- sa gestion est **"automatique"** : allocation pour les variables lors du début de la fonction et libération à la fin de la fonction,
- **durée de vie** des espaces alloués : le temps d'exécution du sous-programme.
- un sous-segment pour le **tas** – mémoire gérée par le programmeur,
 - il est utilisé lors des appels à l'opérateur `new` (tableaux et objets),
 - sa gestion est **assurée par le programmeur** : allocation (`new`) et libération de la mémoire
 - responsabilité au programmeur de faire les allocations **lorsque de besoin**,
 - responsabilité au programmeur de libérer la mémoire lorsqu'elle n'est plus nécessaire (c'est automatique en Java vous verrez ... mais cf. Langage C),
 - **durée de vie** des espaces alloués : entre le `new` et la libération, indépendamment du sous-programme qui fait le `new`.

Représentation graphique : pour schématiser la représentation mémoire, il faut imaginer deux espaces séparés, l'un de l'autre : la pile et le tas.

Exemple :

```

---
public static void main(String[] argv) {
    int i, j, k ;
    double d, e, f;
    i = 1;
    j = i + 10 ;
    k = j ;
    e = 10.5;
    // ❶
    if (e > i) {
        f = j + k ;
        k = (int) (f - e) ;
    }
}
---
```

État de la pile lorsque le programme ci-dessus arrive en <1>

	Pile		Espace des objets (géré par la JVM - tas)
main()	i	1	
	j	11	
	k	11	
	d	?	
	e	10.5	
	f	?	
	argv	4D4D	

2. Implantation mémoire des tableaux

2.1. Exemple de déclaration de tableau en java

```
int    tab [], i; // ❶ // espaces ajoutés exprès
int [] tab2, t; // ❷
int [] tab3 = {1, 2, 3}; // ❸
```

- ❶ Déclare : `tab` : tableau ([]) d'entiers (int), et `i` entier (int)
- ❷ Déclare : `tab2` : tableau d'entiers (int []) et `t` tableau d'entiers (int []) - A PREFERER
- ❸ Déclare : `tab3` : tableau d'entiers (int []), ET EN PLUS crée un tableau de 3 éléments de valeurs 1, 2, 3

Différence entre les déclarations :

- déclarations 1 et 2, les tableaux d'entiers n'existent pas encore ...,
- déclaration 3 : crée un tableau de 3 entiers avec les valeurs 1, 2, 3

A noter : le type "tableau" n'existe pas, mais seul le type "tableau de ..." (quelque chose) existe. Une déclaration de tableau **est toujours liée au type des éléments** : tableau d'un type particulier (int, float, Personne, Pile, Etudiant, ...).

A noter : ces variables tableaux de entiers, en tant que variables déclarées dans un sous-programme, sont créées/stockées, dans la pile d'exécution, comme les variables simples des exemples ci-avant.

2.2. Un tableau Java : une **référence**

Que contient une variable de type tableau (de type de base) :

- pas les éléments du tableau, sinon la réservation mémoire de `tab` et `tab2` ci-avant serait impossible,
- mais une **référence** vers un tableau qui lui contient les éléments, le tableau est stocké dans le **tas**.



Définition : référence de tableau java

Une **référence de tableau** en Java permet de **désigner** un tableau :

- appelé aussi "handle" ou "poignée",
- c'est un numéro d'objet : Old (Object Identifier) alloué par le système (JVM),
- donne une identité (unique) au tableau et permet d'y accéder,
- plusieurs variables tableau peuvent référencer le même tableau (chacun a la "poignée")

A noter : Une référence de tableau est un numéro unique mais **n'est pas** une adresse mémoire. Sa valeur ne permet pas d'accéder à la RAM. Seule la JVM sait faire le lien entre ce numéro et un emplacement mémoire.

Pour obtenir la valeur d'une référence de tableau :

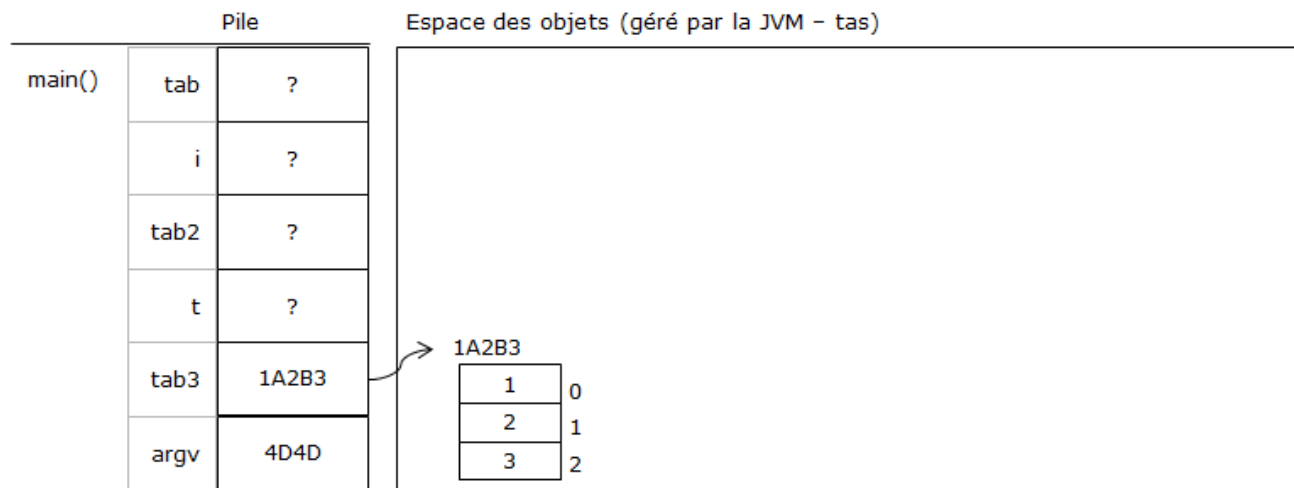
```
System.identityHashCode(tab);
```

2.3. Représentation mémoire des tableaux

Les **variables tableau** déclarées sont automatiques \Rightarrow elles sont stockées dans la pile.

```
int    tab [], i;  
int [] tab2, t;  
int [] tab3 = {1, 2, 3};
```

Représentation mémoire associée :



2.4. L'opérateur `new` pour créer des tableaux

Utiliser l'opérateur `new` pour créer un tableau :

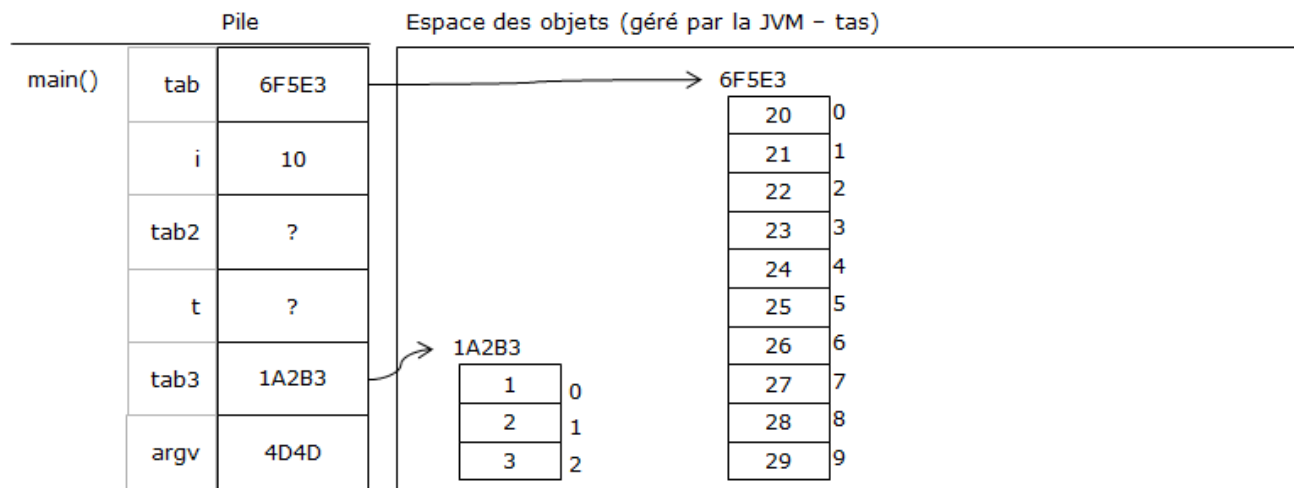
- Il faut donner à l'opérateur :
 - le type des éléments du tableau à créer,
 - le nombre d'éléments voulu entre [et] (nombre ≥ 0). Sic : un tableau de longueur 0 est possible.
- Permet de créer un tableau :
 - réserve la mémoire nécessaire dans le tas : environ le nombre d'octets du type indiqué \times nombre des éléments,
 - initialise chaque élément du tableau alloué à une valeur par défaut (0 pour les nombres, `false` pour les booléens, `null` pour les objets et les tableaux),
 - **renvoie la référence** du tableau créé dans le tas.

Exemple : (on ajoute à la suite du code de déclaration précédent) :

```
tab = new int [10]; // ❶
for (int i = 0; i < tab.length; i++) {
    tab[i] = 20+i; // ❷
}
```

- ❶ Crée un tableau de 10 int (~40 octets) et renvoie la référence du tableau créé qui est stockée dans `tab`.
- ❷ On utilise le tableau ... au travers de sa référence contenue dans `tab`.

Représentation mémoire :



2.5. Type tableau : valeurs et opérations

Pour une variable `tab` déclarée comme tableau de type de base (`int`, `double`, ...):

- valeurs possibles : une référence vers un tableau du type déclaré,
- opérations possibles :
 - affecter une valeur de référence de tableau du type déclaré,
 - affecter une référence renvoyée par un `new` (ex : `tab = new int[10]`),
 - affecter une référence contenue dans une autre variable ou renvoyée par un appel de sous-programme (ex : `tab2 = tab`),
 - accéder à un élément par `[]` contenant une expression de type `int` donnant un indice valide (`0 <= indice < tab.length`),
 - accéder à la longueur du tableau donnée lors de sa création par l'attribut (en lecture seule)


```
length : tab.length.
```

Sur un élément de tableau, les valeurs et opérations possibles dépendent ... du type déclaré des éléments du tableau.

NB : d'autres opérations existent mais ne sont pas liées intrinsèquement au type mais au langage. Ex : le passage en paramètre d'un sous-programme.

2.6. Typage et expressions

Dans l'expression `tab[i]`, 2 types sont en jeu :

- `tab` : déclaré comme `int[]` ... donc tableau d'entiers ... donc REFERENCE de tableau d'entiers,
- `tab[i]` : de type `int` qui est interprété comme "le *i*ème élément du tableau dont la référence est dans `tab`".


```

tab  [i]
---|      tableau d'entiers (référence vers)
-----|  entier

```

L'expression `tab[i]` ne doit jamais être interprétée comme un tableau ... mais comme un **int**.

2.7. Application : représentation mémoire des tableaux

Donner le résultat affiché et la représentation mémoire (pile/tas) à la fin du programme suivant :

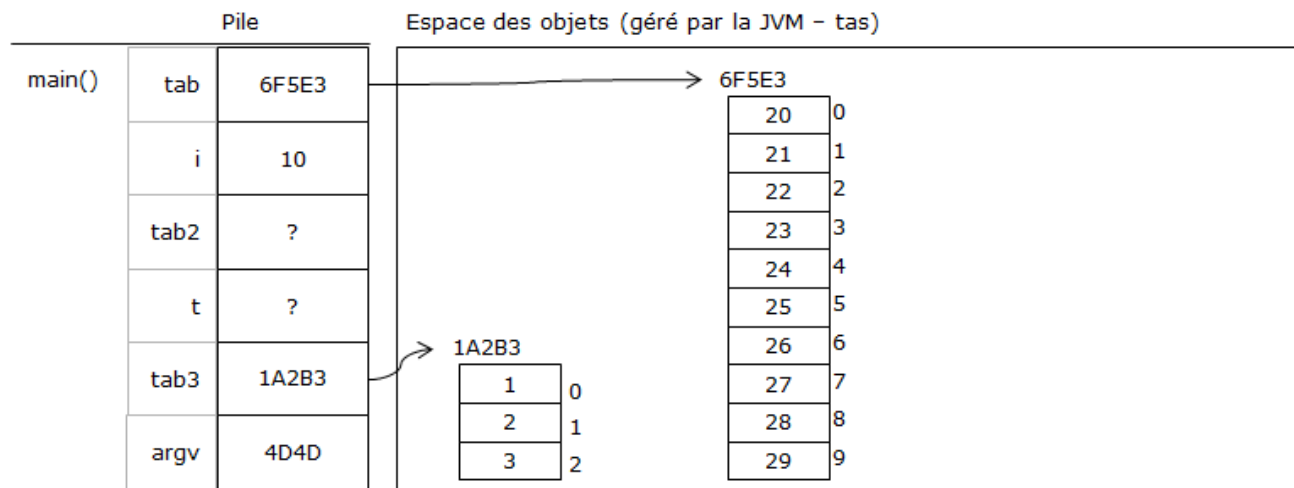
```

public static void main(String[] argv) {
    int tab [], i;
    int [] tab2, t;
    int [] tab3 = {1, 2, 3};
    tab = new int [10];
    for (i = 0; i < tab.length; i++) {
        tab[i] = 20+i;
    }
    tab2 = tab;
    t = tab3;
    for (i = 0; i < tab2.length; i++) {
        System.out.println (tab2[i]);
    }
    for (i = 0; i < t.length; i++) {
        System.out.println (t[i]);
    }
}

```

Réponse, affiche :

Représentation mémoire à mettre à jour :



3. Les objets Java

3.1. Un objet Java : une référence

Les objets sont issus d'une classe et sont créés par l'opérateur `new` avec appel d'un **constructeur**.

Une **variable déclarée de type objet** ne contient pas l'objet ... mais ... une **référence** vers l'objet réellement créé dans le tas.



Définition : référence d'objet java

Une **référence d'objet** en Java permet de **désigner** un objet :

- appelé aussi "handle" ou "poignée",
- c'est un numéro d'objet : Old (Object Identifier) alloué par le système (JVM),
- donne une identité (unique) à l'objet et permet d'y accéder,
- plusieurs variables objet peuvent référencer le même objet (chacun a la "poignée")

A noter : Une référence d'objet est un numéro unique mais **n'est pas** une adresse mémoire. Sa valeur ne permet pas d'accéder à la RAM pour accéder à l'objet. Seule la JVM sait faire le lien entre ce numéro et un emplacement mémoire.

Pour obtenir la valeur d'une référence d'objet :

```
System.identityHashCode(pers);
```



Attention

`null` : une valeur particulière de référence (objet ou tableau) qui veut dire "aucun objet référencé".

`null` **n'est pas** "rien" ou "indéfini". A distinguer d'une variable "may not have been initialized".

`null` : on peut tester si une variable contient `null` (`if (pers == null) ...`) mais on ne peut pas accéder à l'objet correspondant car cela génère une `NullPointerException`. Exemple :

```
pers = null;

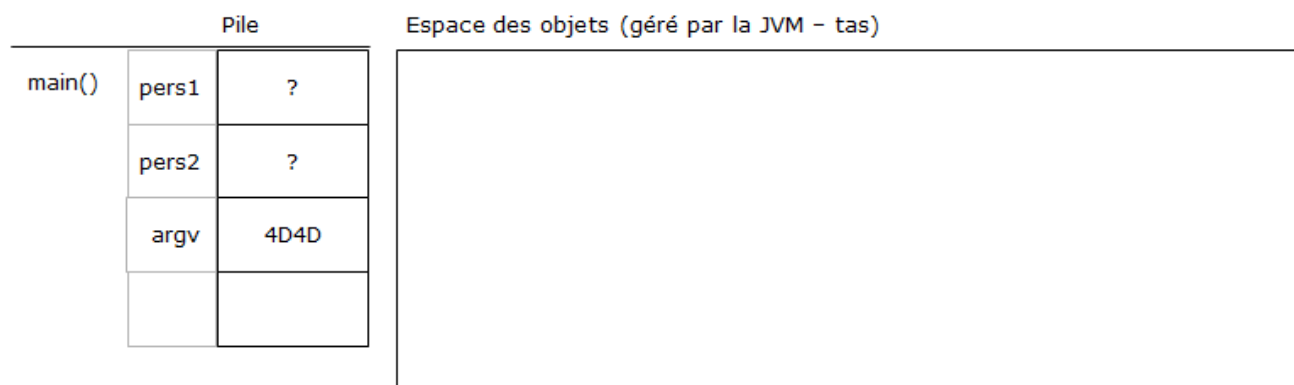
pers.nom = "" //NullPointerException.
```

3.2. Représentation mémoire des objets

Les **variables objet** déclarées sont automatiques \Rightarrow appartiennent à la pile.

```
class Personne {
    // ...
    public static void main(String[] argv) {
        Personne pers1, pers2; // ❶
        // ...
    }
}
```

Représentation mémoire **en position <1>** :



3.3. L'opérateur `new` pour les objets

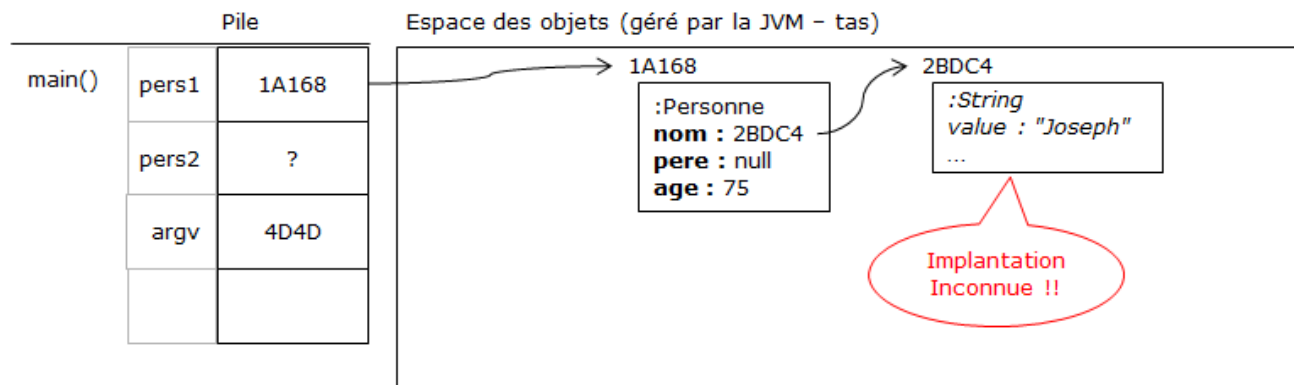
Utiliser l'opérateur `new` pour créer un objet :

- L'appel à l'opérateur `new` pour créer un objet se fait en donnant (Ex : `new Personne ("Joseph", null, 75)`) :
 - un nom de classe,
 - les paramètres éventuels du constructeur entre parenthèses, `()` sinon.
- Permet de créer un objet :
 - réserve la mémoire nécessaire : environ le nombre d'octets pour stocker les attributs définis par la classe
 - initialise chaque attribut (`0` pour les nombres, `false` pour les booléens, `null` pour les objets et les tableaux),
 - applique le constructeur spécifié par les paramètres,
 - **renvoie la référence** de l'objet créé dans le tas.

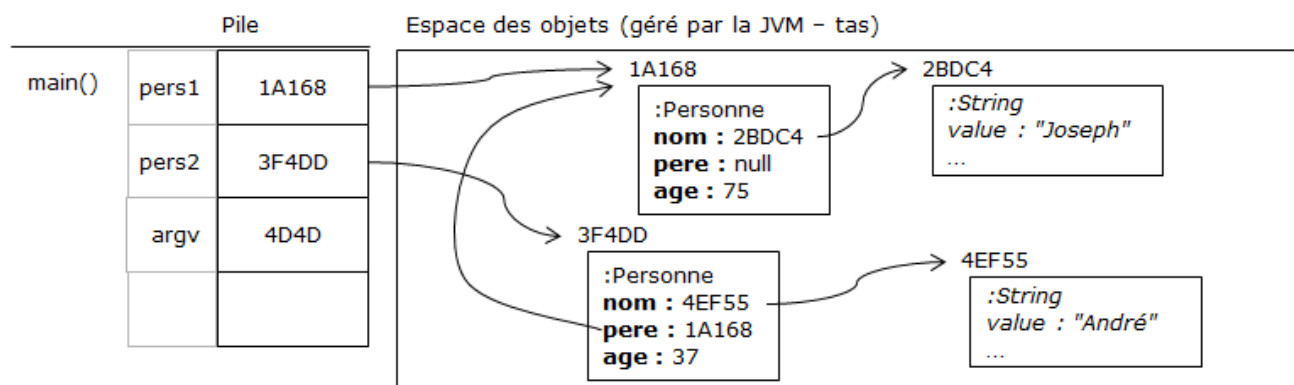
```
class Personne {
    String nom ;
    Personne pere ;
    int age ;
    public Personne (String pfNomPers, Personne pfPerePers, int pfAgePers) {
        this.nom = pfNomPers;
        pere = pfPerePers; // this optionnel MAIS TRES RECOMMANDE
        this.age = pfAgePers;
    }
    public static void main(String[] argv) {
        Personne pers1, pers2;
        pers1 = new Personne ("Joseph", null, 75); // ❶
        pers2 = new Personne ("André", pers1, 37); // ❷
    }
}
```

- ❶ Crée un objet `Personne` et renvoie la référence stockée dans `pers1`
- ❷ Crée un objet `Personne` et renvoie la référence stockée dans `pers2`

Représentation mémoire **en position <1>** :



Représentation mémoire **en position <2>** :



A noter :

- la valeur `null` de l'attribut `pere` de l'objet "pers1" (référéncé par pers1)
- la valeur de l'attribut `pere` de l'objet "pers2" (référéncé par pers2)
- les chaînes de caractères sont aussi des objets : elles ont une référence et contiennent des attributs. Mais on ne connaît pas ses attributs (private, encapsulation).

3.4. Type objet : valeurs et opérations

Pour une variable déclarée comme objet (ex : `Personne pers;`) :

- valeurs possibles : une référence vers un objet du type déclaré (`pers` \Rightarrow `Personne`),
- opérations possibles :
 - affecter une valeur de référence objet du type déclaré,
 - affecter une référence renvoyée par un `new` (ex : `pers = new Personne("Joseph", null, 75);`),
 - affecter une référence contenue dans une autre variable ou renvoyée par un appel de sous-programme (ex : `pers2 = pers;`),

- affecter la valeur `null` (ex : `pers2 = null;`) ; cela signifie que la variable ne référence aucun objet,
- utiliser l'objet : accès aux attributs lorsque possible ou appeler une méthode sur l'objet (ex : `pers.setPere(pers2);`).

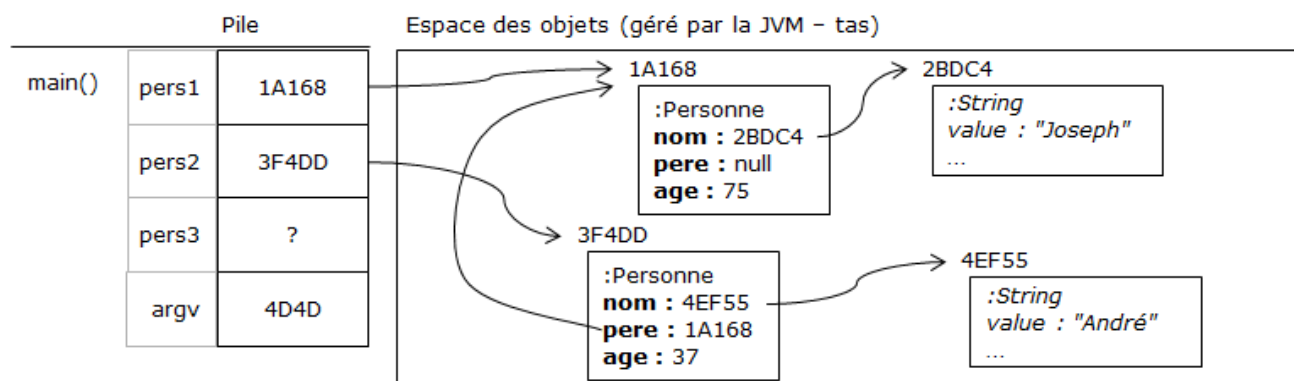
NB : d'autres opérations existent mais ne sont pas liées intrinsèquement au type mais au langage. Ex : le passage en paramètre d'un sous-programme. D'autres opérations seront détaillées plus tard.

3.5. Application : représentation mémoire des objets

Soit le programme suivant :

```
class Personne {
    String nom ;
    Personne pere ;
    int age ;
    public Personne (String pfNomPers, Personne pfPerePers, int pfAgePers) {
        this.nom = pfNomPers;
        pere = pfPerePers; // this optionnel MAIS TRES RECOMMANDE
        this.age = pfAgePers;
    }
    public static void main(String[] argv) {
        Personne pers1, pers2;
        Personne pers3;
        pers1 = new Personne ("Joseph", null, 75);
        pers2 = new Personne ("André", pers1, 37);
        pers3 = pers1; // ❶
        pers2 = pers1; // ❷
    }
}
```

Quel est le résultat en terme de contenu des variables et en terme de création/ou pas d'objets en mémoire des instructions <1> et <2> ci-dessus ? Dessinez-le ci-dessous.



3.6. Egalité d'objet : "==" et "equals()"

Rapportez ce que vous venez de voir avec le TP réalisé en S1 (module M1102-IAP) dans lequel vous faisiez des tests sur l'utilisation de l'opérateur == et de la méthode equals().

Voici un extrait du code écrit en S1 :

```
public class EqualOrNotEqual {
    public static void main(String[] arguments) {
        String message1 = new String ("IUT 2013-2014");
        String message2 = new String ("IUT" + " 2013-2014");
        String equalOrNot ;

        if (message1 == message2) { equalOrNot = "=="; } else { equalOrNot =
"!="; }

        System.out.println("message1 "+equalOrNot+" message2");

        if (message1.equals(message2)) { equalOrNot = "equals"; } else {
equalOrNot = "not equals"; }
        System.out.println("message1 "+equalOrNot+" message2");
    }
}
```

Expliquer le résultat d'exécution ci-dessous :

Résultat exécution :

```
message1 != message2      // CAR :
message1 equals message2  // CAR :
```

4. Appel de sous-programme - Passage des paramètres

4.1. Règle de passage des paramètres

Lors d'un appel de sous-programme, les opérations suivantes sont réalisées par le système (~JVM) :

- à partir de la signature et des déclarations, les paramètres et les variables locales du sous-programmes sont implantés dans la pile,
- la valeur de chaque paramètre effectif est copiée dans le paramètre formel correspondant,
- le corps du sous-programme est exécuté,

- à la fin du sous-programme, la valeur éventuellement retournée par le sous-programme remplace l'appel et l'évaluation de l'appelant continue.

4.2. Exemple 1

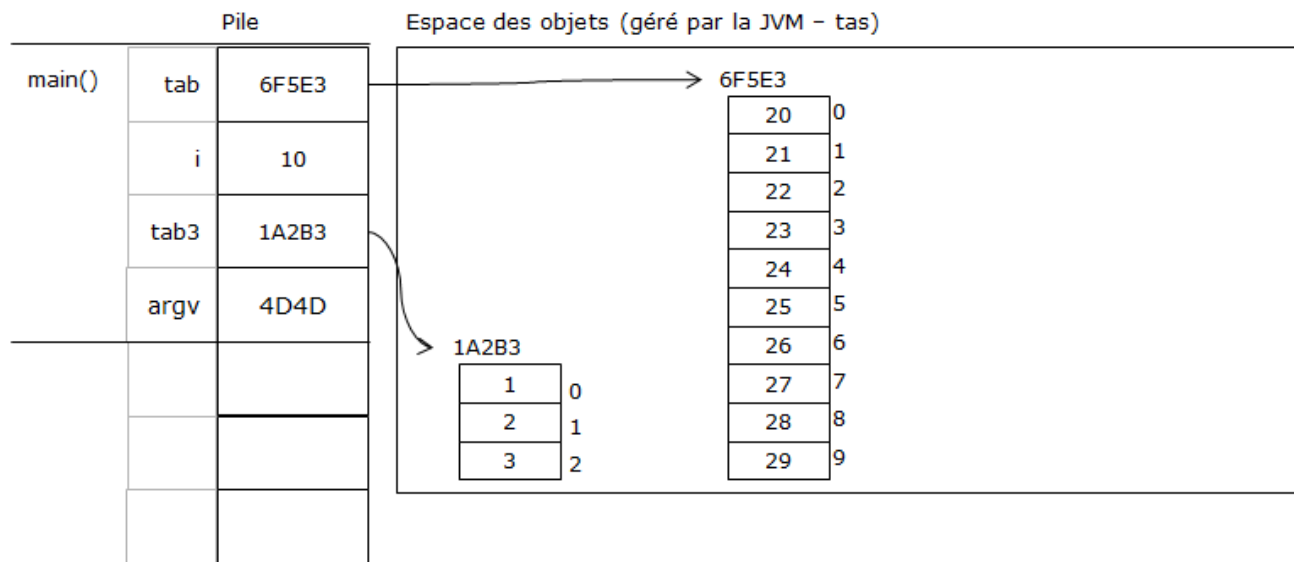
```
public MaClasse {  
    public static void afficheTab (int[] t, int i) {  
        int j;  
        System.out.println (i);  
        for (j=0; j<t.length; j++) {  
            System.out.println(t[j]);  
        }  
    }  
    public static void main(String[] argv) {  
        int [] tab ;  
        int i;  
        int [] tab3 = {1, 2, 3};  
        tab = new int [10];  
        for (i = 0; i < tab.length; i++) {  
            tab[i] = 20+i;  
        }  
        MaClasse.afficheTab (tab, 150); // ❶  
        MaClasse.afficheTab (tab3, tab[9]*100); // ❷  
    }  
}
```

Qu'affiche ce programme ?

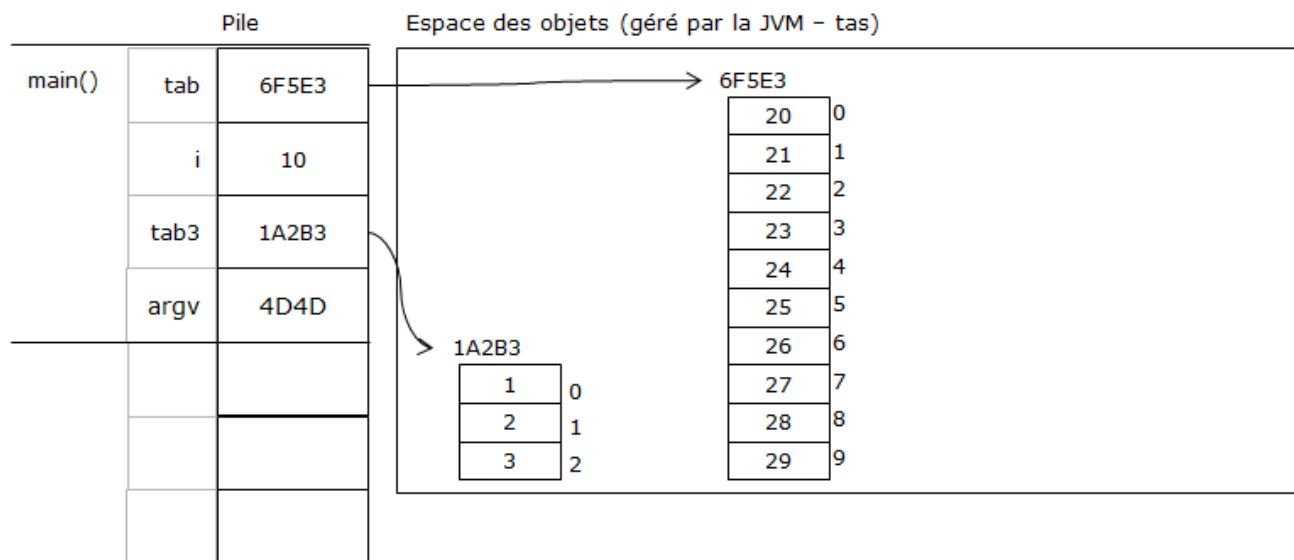
Réponse, affiche : ...

Donnez ci-dessous la représentation mémoire correspondante au début du sous-programme

afficheTab() **en position <1>** (MaClasse.afficheTab(tab, 150)) :



Donnez ci-dessous la représentation mémoire correspondante au début du sous-programme `afficheTab()` **en position <2>** (`MaClasse.afficheTab(tab3, tab[9]*100)`) :



4.3. Exemple 2

```

public class MaClasse {
    public static int somTab (int[] t) {
        int j, som;
        som = 0;
        for (j=0; j<t.length; j++) {
            som = som+t[j];
        }
        return som;
    }

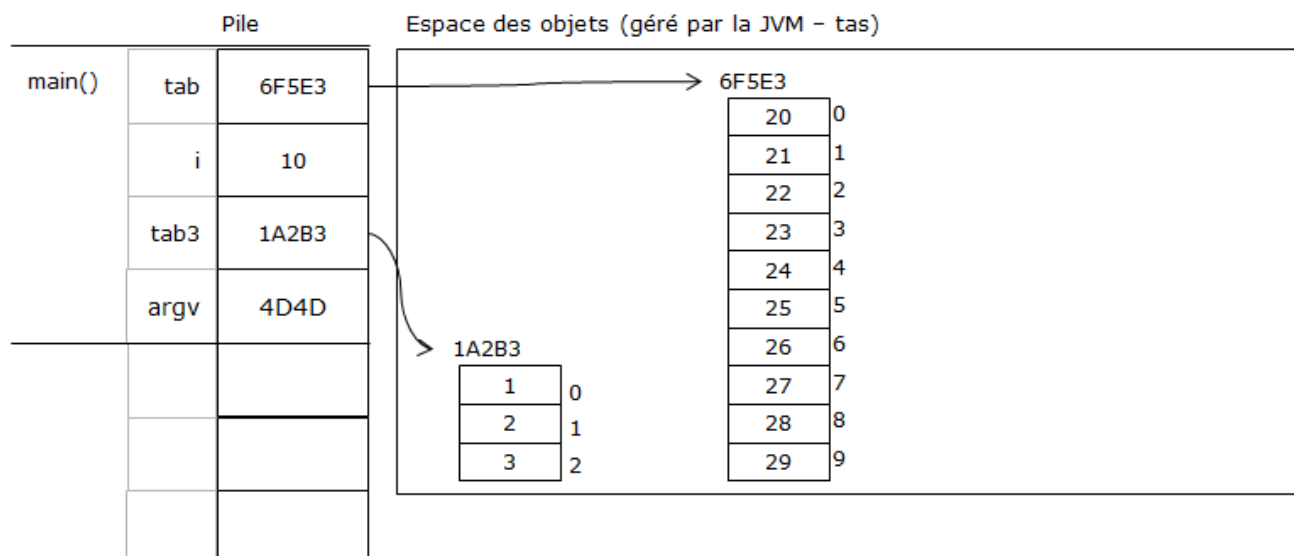
    public static void main(String[] argv) {
        int [] tab ;
        int i;
        int [] tab3 = {1, 2, 3};
        tab = new int [10];
        for (i = 0; i < tab.length; i++) {
            tab[i] = 20+i;
        }
        System.out.println(MaClasse.somTab(tab)); // ❶
        System.out.println(MaClasse.somTab(tab3)); // ❷
    }
}

```

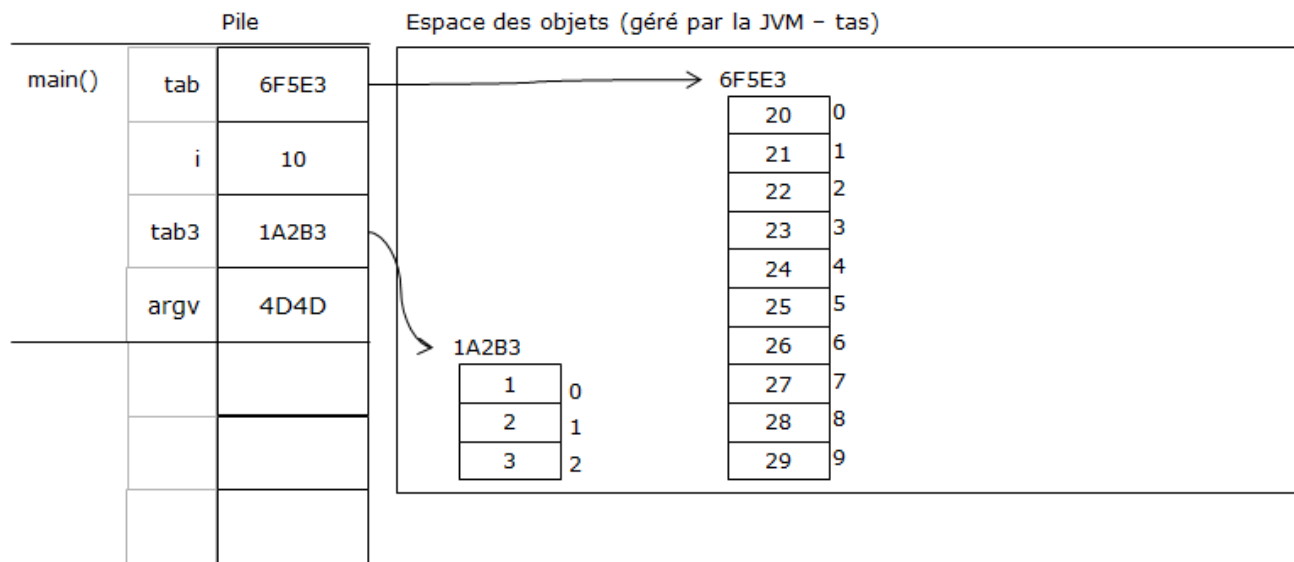
Qu'affiche ce programme ?

Réponse, affiche :

Dans le schéma suivant, donner la représentation mémoire correspondante au début du sous-programme `somTab()` **en position <1>** (`MaClasse.somTab(tab)`) :



Dans le schéma suivant, donner la représentation mémoire correspondante au début du sous-programme `somTab()` **en position <2>** (`MaClasse.somTab(tab3)`) :



4.4. Exemple 3

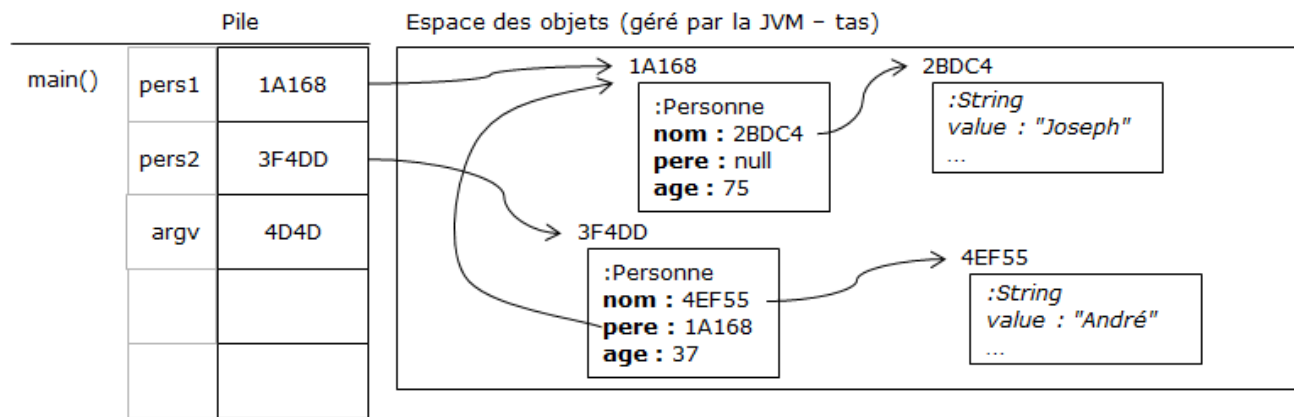
Soir le programme suivant :

```
public class Personne {
    private String nom ;
    private Personne pere ;
    private int age ;
    public Personne (String pfNomPers, Personne pfPerePers, int pfAgePers) {
        this.nom = pfNomPers;
        pere = pfPerePers; // this optionnel MAIS TRES RECOMMANDE
        this.age = pfAgePers;
    }
    public String getNom () {
        return this.nom;
    }
    public void setPere (Personne p) {
        this.pere = p;
    }
    public static void main(String[] argv) {
        Personne pers1, pers2;

        pers1 = new Personne ("Joseph", null, 75);
        pers2 = new Personne ("André", pers1, 37);
        pers2.setPere (pers1); // ❶ // Même si c'est inutile ici car va
        // refaire le même lien.
        System.out.println(pers1.getNom()); // ❷
        System.out.println(pers2.getNom()); // ❸
    }
}
```

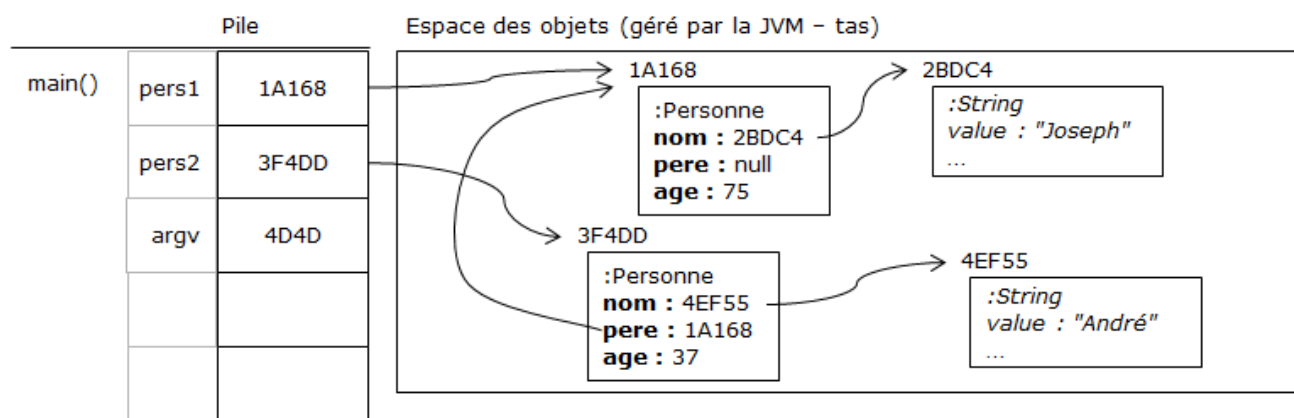
Donner la représentation mémoire correspondante au début du sous-programme `setPere()` en

position <1> (`pers2.setPere (pers1);`) :



Donner la représentation mémoire correspondante au début du sous-programme `getNom()` **en**

position <2> (`pers1.getNom();`) :



Donner la représentation mémoire correspondante au début du sous-programme `getNom()` **en**

position <3> (`pers2.getNom();`) :

