Séquence 7

Références et exécution de programme

1 Introduction

Nous avons vu comment la mémoire permettant d'exécuter un programme est structurée en trois parties : mémoire des classes, pile et tas. Nous avons vu comment une référence permet d'associer un espace du tas contenant la mémoire privée d'un objet au nom d'un paramètre ou d'une variable qui est dans la pile, dans la mémoire privée d'une méthode ou d'un constructeur.

Cette référence peut être figurée par un identifiant (par exemple id478 dans Pythontutor) que l'on retrouve à la fois dans la pile et dans le tas ou par une flèche qui part de la pile et arrive dans le tas.

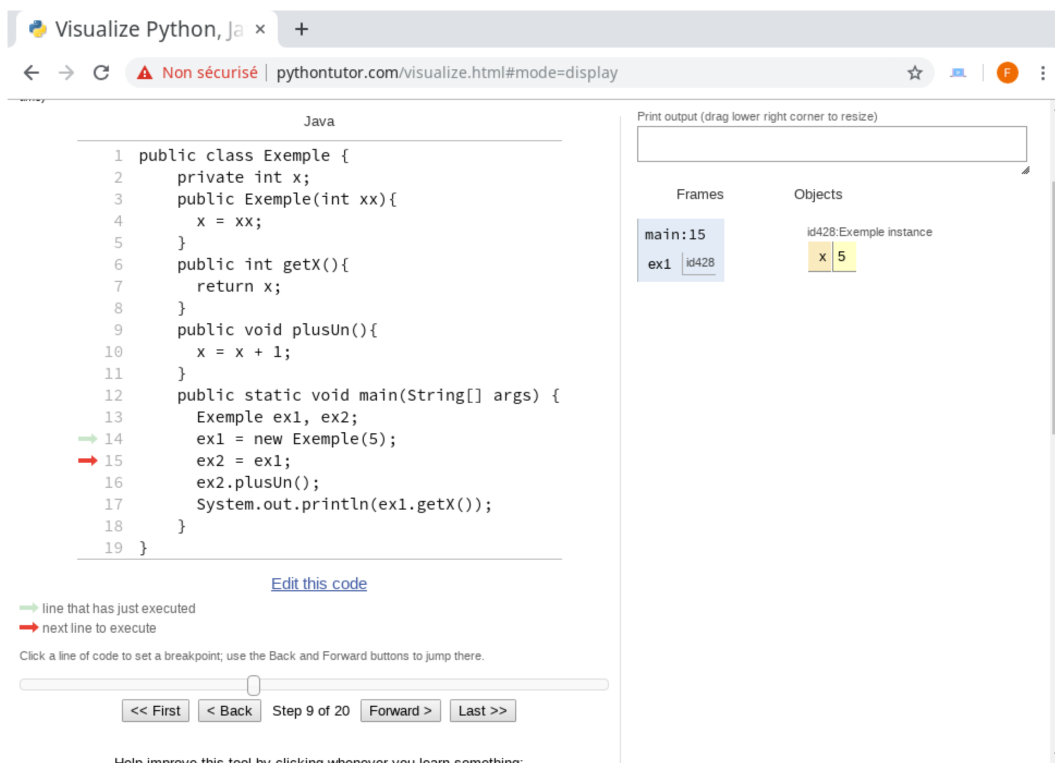
Nous avons vu que Pythontutor permet d'afficher un schéma de la mémoire au fil de l'exécution d'un programme simple. Nous allons voir ici, toujours en utilisant Pythontutor, ce qui se passe lors de l'exécution d'affectations et de passage de paramètres.

2 Affectation et aliasing

L'affectation d'un objet à une variable ne crée pas un nouvel objet : ce qui crée un objet, c'est l'instruction `new` directe ou indirecte. Voyons l'exécution du petit programme suivant :

```
public class Exemple {
    private int x;
    public Exemple(int xx){
      x = xx;
    }
    public int getX(){
      return x;
    }
    public void plusUn(){
      x = x + 1;
    }
    public static void main(String[] args) {
      Exemple ex1, ex2;
      ex1 = new Exemple(5);
      ex2 = ex1;
      ex2.plusUn();
      System.out.println(ex1.getX());
    }
}
```

Visualisons l'exécution avec Pythontutor pour voir l'effet de l'affectation de la ligne 15. Après exécution de la ligne 14 et avant exécution de la ligne 15, nous avons la situation suivante.



On voit qu'il y a un seul objet dans le tas, de type Exemple, avec une variable `x` qui vaut 5. Cet objet a pour référence `id428`. Après exécution de la ligne 15 (affectation de l'objet contenu dans `ex1` à la variable `ex2`), la mémoire est la suivante.

La variable `ex2` dans la pile contient la référence contenue dans `ex1`, à savoir `id428`. Il y a deux variables contenant un objet mais un seul objet dans le tas. Il y a un seul objet parce qu'il y a un seul `new` dans le programme. Il y a deux variables de type `Exemple` et deux affectations de valeurs. Les deux variables contiennent le même objet. Ce sont deux noms différents qui désignent la même chose. On dit que ce sont deux *alias*.

Dans la suite, lorsqu'on modifie l'objet `ex2` et qu'on affiche la variable de `ex1`, on voit qu'elle a été impactée par la modification de `ex2`. En effet, on voit bien que dans la mémoire, il n'y a qu'une seule variable `x`.

On le constate aussi avec l'affichage à l'écran qui affiche 6 en fin de programme.

3 Comparaison avec les types primitifs

Ce que l'on vient de voir ne peut jamais se produire avec les valeurs des types primitifs (int, double, char, boolean). En effet, on ne peut pas avoir deux variables qui désignent le même nombre de telle sorte que si l'on modifie l'une, cela modifie l'autre aussi.

Gardons le même code, avec cette fois le type `int` pour les deux variables. Exécutons le code avec Pythontutor.

Juste avant l'affectation de la ligne 15, la situation est la suivante.

may crash at any time)

Java

```
1 public class Exemple {
2     public static void main(String[] args) {
3       int ex1, ex2;
4       ex1 = 5;
5       ex2 = ex1;
6       ex2 = ex2 + 1;
7       System.out.println(ex1);
8     }
9 }
```

Edit this code

line that has just executed
next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 3 of 6 Forward > Last >>

Print output (drag lower right corner to resize)

Frames Objects

main:5
ex1 5

Après l'affectation de la ligne 15, on a la situation suivante.

ervice
ıay crash at any
me)

Java

```
1 public class Exemple {
2     public static void main(String[] args) {
3       int ex1, ex2;
4       ex1 = 5;
5       ex2 = ex1;
6       ex2 = ex2 + 1;
7       System.out.println(ex1);
8     }
9 }
```

Edit this code

line that has just executed
next line to execute

:lick a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 4 of 6 Forward > Last >>

Print output (drag lower right corner to resize)

Frames Objects

main:6
ex1 5
ex2 5

On voit ici que c'est l'entier lui-même qui a été recopié d'une case mémoire dans l'autre, dans la pile. Il y a deux cases mémoires distinctes avec la valeur 5. Changer la valeur contenue dans une case ne change pas la valeur contenue dans l'autre.

D'où la situation après exécution de la ligne 16 où les deux variables contiennent des entiers différents.

time)

Java

```
1 public class Exemple {
2     public static void main(String[] args) {
3       int ex1, ex2;
4       ex1 = 5;
5       ex2 = ex1;
6       ex2 = ex2 + 1;
7       System.out.println(ex1);
8     }
9 }
```

Edit this code

line that has just executed
next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 6 of 6 Forward > Last >>

Print output (drag lower right corner to resize)

5

Frames Objects

main:8
ex1 5
ex2 6

Dans le cas des objets étudié à la section précédente, il y a une seule case mémoire avec la valeur

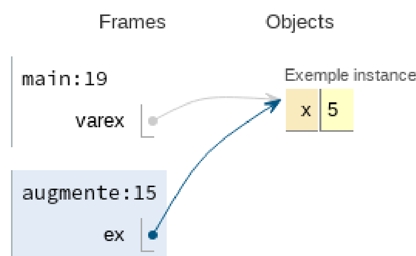
5 et elle est dans le tas. On voit donc qu'il y a de vraies différences entre les types primitifs et les types qu'on appelle les *types références* qui sont les types des tableaux et les types des objets.

4 Passage de paramètres par référence

De même qu'affecter un objet à une variable ne crée pas d'objet, donner un objet comme valeur à un paramètre ne crée pas un nouvel objet. En fait, ce n'est pas l'objet lui-même qui est affecté au paramètre mais sa référence. Nous allons le voir en exécutant le programme suivant dans Pythontutor.

```
1  class Exemple{
2      private int x;
3      public Exemple(int xx){
4          x = xx;
5      }
6      public void plusUn(){
7          x = x+1;
8      }
9      public String toString(){
10         return ""+x;
11     }
12 }
13 public class Ref{
14     public static void augmente(Exemple ex){
15         ex.plusUn();
16     }
17     public static void main(String[] args){
18         Exemple varex = new Exemple(5);
19         augmente(varex);
20         System.out.println(varex);
21     }
22 }
```

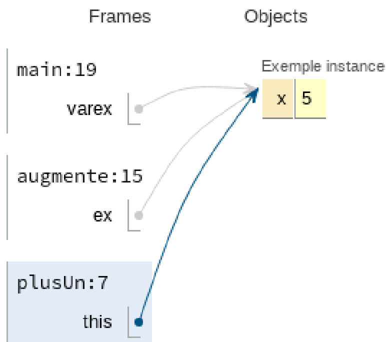
Alors que la ligne 19 est en cours d'exécution et que la prochaine instruction à exécuter est celle de la ligne 15, la mémoire est la suivante.



On voit qu'il n'y a qu'un objet dans le tas et que les deux noms `varex` dans la méthode `main` et `ex` dans la méthode `augmente` sont deux façons de désigner le même objet. Ce sont deux alias.

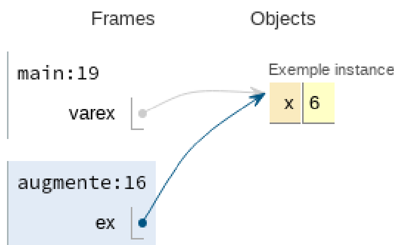
La ligne 15 consiste à modifier l'objet unique qui est en mémoire. On utilise le nom `ex` pour désigner l'objet, et la modification sera visible lorsqu'on désignera l'objet par son autre nom. Ce qu'est l'objet et ce que contiennent ses variable ne dépendent pas du nom utilisé, mais seulement de son emplacement dans la mémoire.

Voyons l'état de la mémoire lorsque l'on commence à exécuter la ligne 15 et que de ce fait, la prochaine instruction à exécuter sera celle de la ligne 7.



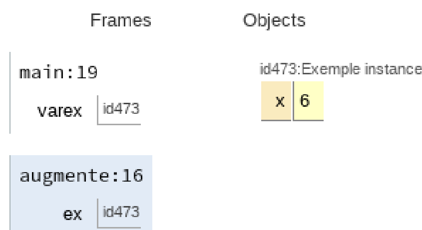
On voit qu'il y a trois méthodes en cours d'exécution : `main`, `augmente` et `plusUn`. Le même objet est référencé sous trois noms différents : `varex` et `ex` que l'on a déjà vus et `this` dans la méthode `plusUn`. Ce `this` n'est ni une variable ni un paramètre, mais un mot-clé qui dans une méthode, désigne l'objet sur lequel la méthode a été appelée.

Après exécution de la ligne 7, la variable `x` a été augmentée de une unité. Elle contient à présent la valeur 6. L'état de la mémoire est le suivant.



Lorsqu'en fin de programme, on affiche la contenu de la variable `x` de l'objet référé par `varex`, c'est 6 qui s'affiche.

Passer un objet en paramètre à une méthode ne crée pas de nouvel objet. C'est en fait la référence de l'objet qui est copiée dans une autre case mémoire de la pile. Cela apparaît peut-être plus clairement si l'on utise l'autre façon d'afficher la mémoire en notant les références au moyens d'identificateurs. Avec cette autre notation, la mémoire est figurée comme suit.



On voit que la référence `id473` qui désigne l'objet apparaît dans les deux espaces mémoires de la pile : `ex` dans la mémoire allouée à la méthode `augmente` et `varex` dans la mémoire allouée à la méthode `main`.

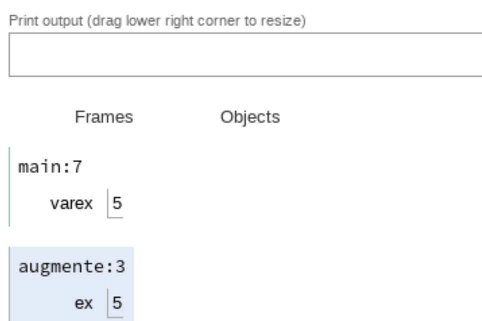
5 Passage de paramètres de types primitifs

Si un paramètre est de type primitif, le comportement du programme est différent. Prenons l'exemple d'un entier `int`. Si un entier est passé en paramètre, il n'y a pas deux noms différents pour désigner le même emplacement en mémoire. La valeur est dupliquée, recopiée dans un autre espace mémoire. Donc si l'on ajoute un au nombre présent dans une méthode, cela ne modifie pas celui utilisé par l'autre méthode.

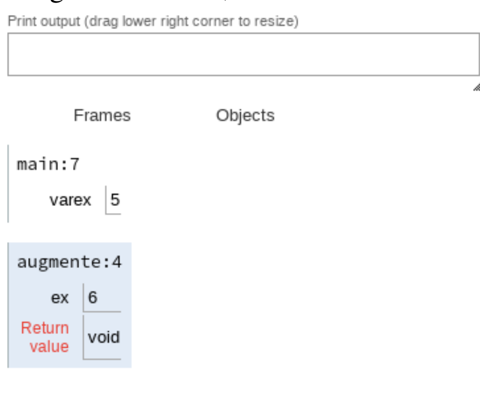
Adaptons un peu le code du programme `Ref` pour montrer cela.

```
1  public class Ref2{
2      public static void augmente(int ex){
3          ex = ex + 1;
4      }
5      public static void main(String[] args){
6          int varex = 5;
7          augmente(varex);
8          System.out.println(varex);
9      }
10 }
```

L'état de la mémoire au moment de commencer à exécuter la ligne 3 est le suivant.

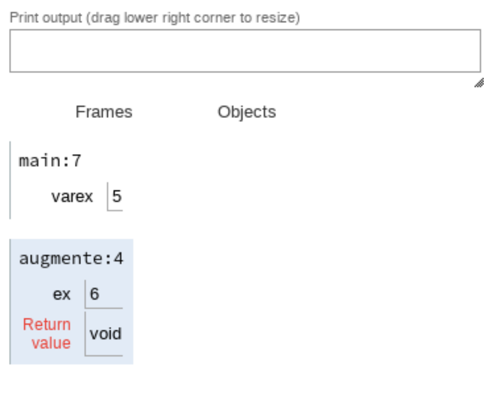


On voit que la valeur 5 apparaît dans deux cases mémoires distinctes. Lors du passage de paramètre, il y a eu recopie de l'entier lui-même dans l'espace mémoire de la méthode `augmente`. Une fois la ligne 3 exécutée, l'état est le suivant.



L'exécution de la méthode a modifié la case mémoire du paramètre `ex`, mais cela n'a pas d'influence sur la case mémoire de la variable `varex`.

En fin d'exécution, l'état mémoire est le suivant.



La mémoire allouée à l'exécution de `augmente` a été effacé, la variable `varex` vaut toujours 5. L'appel de la méthode `augmente` n'a servi à rien. La valeur 5 est affichée à l'écran.

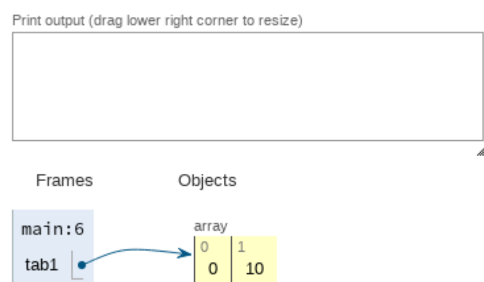
6 Et les tableaux ?

Un tableau comme un objet est créé par un `new`, il est en mémoire dans le tas et une variable de type tableau contient une référence au tableau. Nous pouvons voir cela avec le petit exemple suivant.

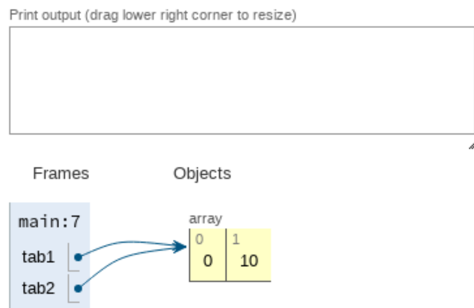
Les tableaux se comportent comme des objets et pas comme des types primitifs. Ils sont soumis au phénomène d'aliasing.

```
1  public class Ref3{
2      public static void main(String[] args){
3          int[] tab1, tab2;
4          tab1 = new int[2];
5          tab1[1] = 10;
6          tab2 = tab1;
7          tab2[0] = 5;
8          System.out.println("tab1: {"+tab1[0]+", "+tab1[1]+"}");
9          System.out.println(tab1);
10         System.out.println(tab2);
11     }
12 }
```

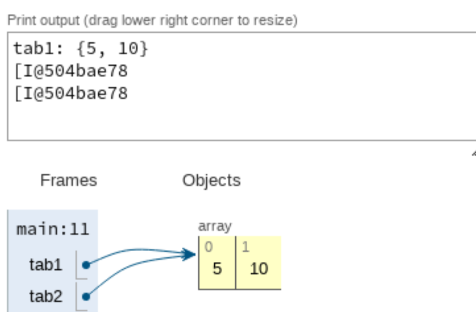
Après exécution de la ligne 5, le tableau `tab1` désigne un tableau de deux cases situé dans le tas. Dans la deuxième case du tableau, il y a l'entier 10.



Ensuite, l'affectation `tab2 = tab1;` ne crée pas un deuxième tableau : elle affecte la référence du tableau désigné par `tab1` à la variable `tab2`. Il y a dès lors deux noms différents pour désigner le même tableau.



Lorsque l'affectation de la ligne 7 change le contenu de la première case du tableau `tab2`, cela change aussi la première case du tableau `tab1` puisqu'il s'agit du même tableau. C'est ce que montre l'affichage de la ligne 8.



Notez que l'instruction `System.out.println` appliquée à un tableau affiche la référence de ce tableau. Les derniers affichages montrent bien que les deux variables contiennent la même référence.

7 Attention aux new cachés

Nous avons vu que ce sont les `new` qui créent les objets, mais il ne faut pas prendre cela trop littéralement. Certains objets sont créés par un programme sans que l'on ne voie directement une instruction `new` dans le code du programme. Cela peut venir de deux choses :

— Il existe une syntaxe spécialisée qui crée un objet ou un tableau. Dans ce cas, on a l'équivalent d'un `new` sans apparition de cet opérateur. Cela existe pour les chaînes de caractères et les tableaux.

— L'objet est créé par un `new`, mais celui-ci n'est pas dans le code du programme, il est dans une méthode invoquée par le programme.

Lorsque dans un programme, on écrit une chaîne de caractère entre guillemets, cela crée à l'exécution un nouvel objet instance de la classe `String` dans le tas.

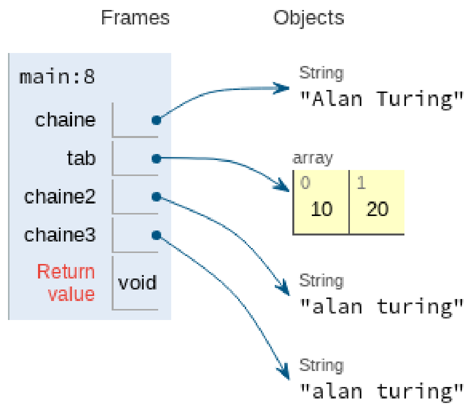
De même, un tableau est créé sans apparition de l'instruction `new` dans le code lorsqu'on le définit à la déclaration avec des accolades.

Un exemple de méthode qui crée un nouvel objet est la méthode `toLowerCase`. Cette méthode est invoquée sur une chaîne de caractère. elle renvoie un nouvel objet qui est une autre chaîne de caractère qui comporte les même caractères mais en minuscule. Chaque fois que la méthode est appelée, elle crée un nouvel objet.

Le programme suivant reprend tous les cas évoqués ici. Il crée trois objets et un tableau.

```
public class NewCache{
```

```
    public static void main(String[] args){
        String chaine = "Alan␣Turing";
        int[] tab = {10, 20};
        String chaine2, chaine3;
        chaine2 = chaine.toLowerCase();
        chaine3 = chaine.toLowerCase();
    }
}
```



Remarquez que chaque appel de `toLowerCase` crée un objet, bien que les deux chaînes créées ici soient identiques.