Exécution de programmes et mémoire

Maria-Virginia Aponte, François Barthélémy

NFA031

1 Variables et mémoire

La mémoire contient les données du programme enregistrées dans des variables. Une variable a un nom et désigne un emplacement de la mémoire où la valeur courante de la variable est enregistrée. Ce qui fait varier la mémoire au cours de l'exécution d'un programme sont trois choses :

- la déclaration d'une variable, qui se traduit par l'allocation d'un espace mémoire et son association avec le nom de la variable.
- l'affectation, qui va stocker une valeur dans l'espace mémoire associé au nom de la variable.
- la fin de vie d'une variable, qui fait que le nom ne désigne plus rien et l'espace mémoire correspondant est libéré. Il pourra être réutilisé pour une autre variable.

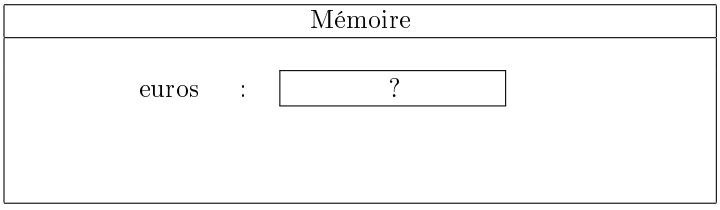
Nous reviendrons sur le troisième point dans une section dédiée à la fin de la vie des variables, et pour l'heure, nous allons nous concentrer sur les deux premiers points à l'aide d'un exemple.

1.1 Exemple

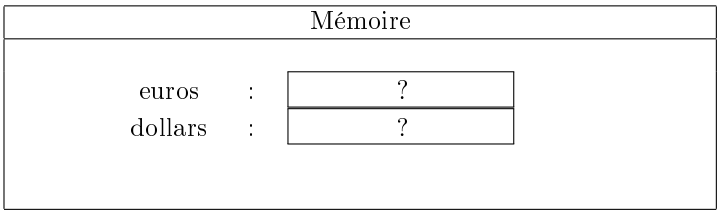
```
1  public class Conversion {
2   public static void main (String[] args) {
3     double euros;
4     double dollars;
5     System.out.println("Somme en euros? ");
6     euros = Terminal.lireDouble();
7     dollars = euros *1.118;
8     System.out.println("La somme en dollars: ");
9     System.out.println(dollars);
10   }
11 }
```

L'exécution du programme commence par la déclaration de la variable `euros`. Cela conduit à l'**allocation** d'un emplacement mémoire suffisamment grand pour y loger un nombre à virgule (8 octets en Java) qui, à partir de là, sera réservé à la variable `euros`. On dira que c'est la **mémoire privée** de la variable.

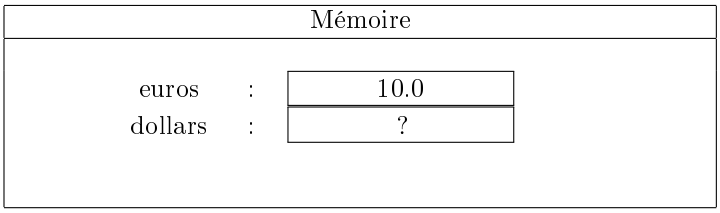
Dans cette mémoire, il n'y a rien d'utilisable : le compilateur refusera toute utilisation du contenu de cette variable tant qu'une valeur ne lui aura pas été affectée. Nous représenterons cela en notant un point d'interrogation dans l'espace correspondant à sa mémoire privée. Voici une représentation du contenu de la mémoire après exécution de cette déclaration :



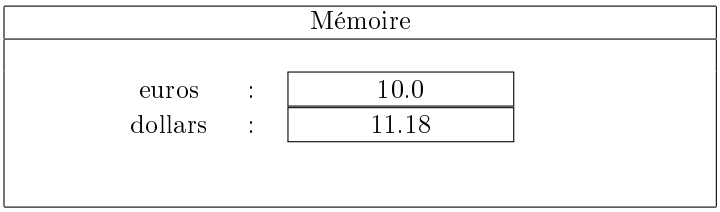
L'exécution de la ligne 4 est de même nature : il s'agit de l'allocation d'un espace mémoire de 8 octets à la variable `dollars`. L'état de la mémoire devient le suivant :



L'exécution de la ligne 5 ne change rien à la mémoire : c'est un affichage à l'écran. La ligne 6 affecte une valeur lue au clavier à la variable `euros`. Cela va se traduire par le stockage de cette valeur dans sa mémoire privée. Pour continuer à représenter une exécution du programme, nous supposons que la valeur 10.0 est lue au clavier. Après exécution de la ligne 6, la mémoire est dans l'état suivant.



La ligne 7 est une affectation à la variable `dollars`. Le calcul utilise la valeur de la variable `euros` qui va être lue dans la mémoire privée de la variable. Après exécution de cette ligne 7, la mémoire est dans l'état suivant.



L'état de la mémoire ne change plus par la suite, car il n'y a ni déclaration ni affectation dans les lignes 8 et 9 du programme.

1.2 Notion de mémoire privée

Nous avons introduit la notion de mémoire privée d'une variable. C'est un espace mémoire qui est entièrement réservé à la variable et qui ne peut être modifié que par une affectation à cette même variable. Le contenu de cet espace privé ne peut être consulté qu'en utilisant le nom de cette variable dans un calcul (expression).

Nous retrouverons cette notion d'espace privé pour d'autres choses que des variables : des tableaux et des méthodes. Il y a aussi une notion de mémoire privée du programme : lorsqu'on demande l'exécution d'un programme Java, le système d'exploitation (Windows, Mac Os, Linux, Android, Ios,...) va allouer des ressources à l'exécution ce ce programme et notamment un espace mémoire privé où seul le programme aura le droit de lire et d'écrire des informations. C'est cette **mémoire privée du programme** que nous représentons dans nos dessins. A la fin de l'exécution du programme, l'espace mémoire privé du programme est rendu au système d'exploitation qui pourra l'attribuer à un autre programme pour son exécution.

1.3 Fin de vie d'une variable : les blocs

La vie d'une variable commence lors de l'exécution de sa déclaration. Sa vie se termine à la fin de la suite d'instructions où elle est déclarée. Cette suite d'instruction, qui est delimitée par deux accolades ouvrante et fermante, s'appelle un bloc.

Il arrive souvent que l'on déclare des variables en début de la méthode main. Ces variables existent jusqu'à la fin de l'exécution de la méthode main, qui est aussi la fin de l'exécution du programme.

Si une variable est déclarée dans le bloc du cas `else` d'un `if`, elle n'existe que pendant et jusqu'à la fin du bloc de ce cas `else`. Prenons un exemple.

```
1  public class Bloc{
2     public static void main(String[] args){
3        System.out.println("Bonjour");
4        if (Math.random()<0.5){
5           System.out.println("Je suis de bonne humeur");
6        }else{
7           String comment;
8           System.out.println("Comment allez-vous?");
9           comment=Terminal.lireString();
10          System.out.println("Vous allez " + comment);
11       }
12       System.out.println("Au revoir");
13    }
14 }
```

La variable `comment` commence à exister à la ligne où elle est déclarée et elle cesse d'exister à la fin de la suite d'instructions (bloc) du `else`, c'est à dire à la ligne 11. Lors d'une exécution, si la condition du `if` est vraie, c'est le cas `if` qui est exécutée, et la variable n'existe à aucun moment. Si la condition est fausse, elle commence à exister après l'exécution de la ligne 7 et elle cesse d'exister après l'exécution de la ligne 10. En particulier, lors de l'exécution de la ligne 12, la variable `comment` n'existe pas dans la mémoire.

Dans le cas d'une boucle `while`, si une variable est déclarée dans le corps de la boucle, une nouvelle variable est créée (nouvel emplacement mémoire) à chaque tour de boucle, au moment où cette déclaration est exécutée, et cesse d'exister à la fin de ce tour de boucle, qui est la fin de la suite d'instructions (bloc) qui constitue le corps de la boucle.

Dans tous les cas, la variable existe jusqu'à la fin du bloc où elle a été déclarée, cette fin étant matérialisée par une accolade fermante : celle qui ferme la méthode, celle qui ferme le bloc du cas `if` ou ou du cas `else`, celle qui ferme le bloc corps d'une boucle.

Il y a un cas un peu différent, celui d'une variable déclarée dans la partie initialisation d'une boucle `for`. Une variable de ce type n'existe que pendant l'exécution de la boucle, mais elle est la même pour tous les tours de boucle, ce qui est un comportement différent d'une variable déclarée dans le corps de la boucle. Il n'est pas possible d'utiliser cette variable après le `for` : elle n'existe plus ; si on a besoin de sa valeur après, il faut la déclarer avant la boucle. Voici un cas incorrect :

```
1  public class Bloc2{
2     public static void main(String[] args){
3        for (int i=0; i<3; i=i+1){
4           System.out.println("i vaut "+i);
5        }
6        System.out.println("i vaut "+i);
7     }
8  }
```

A la compilation : erreur !

```
> javac Bloc2.java
```

```
Bloc2.java:6: error: cannot find symbol
        System.out.println("i vaut "+i);
                                    ^
  symbol:   variable i
  location: class Bloc2
1 error
```

A la ligne 6, la variable n'existe plus. Elle a cessé d'exister à la fin de l'exécution du `for`. Voici comment corriger ce programme pour qu'il soit correct :

```
1 public class Bloc3{
2    public static void main(String[] args){
3       int i;
4       for (i=0; i<3; i=i+1){
5          System.out.println("i vaut "+i);
6       }
7       System.out.println("i vaut "+i);
8    }
9 }
```

La variable `i` déclarée dans le bloc qui est le corps de la méthode `main` existe jusqu'à la fin de ce bloc, et donc, jusqu'à la fin de l'exécution du programme.

2 Trace d'exécution d'un programme

Un programme, lors de son exécution, utilise une mémoire pour stocker des données et interagit avec son environnement via le clavier et l'écran. Comme il est long et fastidieux de dessiner la mémoire après l'exécution de chaque instruction, on peut utiliser une représentation synthétique sous forme de tableau qui relate instruction par instruction l'exécution d'un programme.

Dans ce tableau, chaque ligne donne un état de la mémoire, du clavier et de l'écran à un moment donné de l'exécution. Les colonnes représentent la mémoire, le clavier et l'écran. Plus précisément, la première colonne contiendra le numéro de ligne de l'instruction exécutée, ensuite il y aura une colonne pour chaque variable du programme, une colonne pour le clavier et une pour l'écran. On pourra encore ajouter des colonnes, par exemple pour noter la valeur d'une condition d'un `if` ou d'une boucle.

2.1 Premier exemple de trace d'exécution

```
1  public class PrixTTC {
2     public static void main (String[] args) {
3        double pHT,pTTC;
4        int t;
5        Terminal.ecrireString("Entrer le prix HT: ");
6        pHT = Terminal.lireDouble();
7        Terminal.ecrireString("Entrer taux (normal->0 reduit ->1) ");
8        t = Terminal.lireInt();
9        if (t==0){
10          pTTC=pHT + (pHT*0.2);
11       }
12       else {
13          pTTC=pHT + (pHT*0.05);
14       }
15       Terminal.ecrireStringln("La somme TTC: "+ pTTC );
16    }
17 }
```

Après l'instruction	Mémoire			Entrées clavier	Sorties écran
	pHt	t	pTTC		
	NEP	NEP	NEP		
3	?	NEP	?		
4	?	?	?		
5	?	?			`Entrer le prix HT:`
6	10.0	?		10.0	
7	10.0	?			`Entrer le taux(0,1):`
8	10.0	1		1	
9	10.0	1			
13	10.0	1	10.5		
15	10.0	1	10.5		`La somme TTC: 10.5`

Dans ce tableau, on note NEP (pour n'existe pas) dans les cases représentant des variables non encore déclarées à ce moment de l'exécution. C'est notamment le cas sur la première ligne qui transcrit l'état initial, avant exécution de la première instruction. Un point d'interrogation est utilisé pour une variable déclarée mais à laquelle aucune valeur n'a encore été affectée.

2.2 Trace d'exécution d'une boucle for

La ligne d'en-tête d'une boucle `for` contient trois parties, exécutées à des moments différents du programme : l'initialisation, le test et l'incrémentation. Lorsqu'on mentionne cette ligne dans le tableau relatant l'exécution, il faut préciser laquelle des trois parties est exécutée.

```
1  class Exemple5_0{
2     public static void main(String[] args){
3        int total = 0;
4        int x;
5        Terminal.ecrireString("Entrez le multiplicateur: ");
6        x = Terminal.lireInt();
7        for (int i=1; i<=4; i++){
8           total = total + (i*x);
9        }
10       Terminal.ecrireString("La somme des 4 premiers multiples est ");
11       Terminal.ecrireInt(total);
12       Terminal.sautDeLigne();
13    }
14 }
```

Et voici le tableau qui retrace l'exécution de ce programme :

nb	test	total	x	i	clavier	écran
2		NEP	NEP	NEP		
3		0	NEP	NEP		
4		0	?	NEP		
5		0	?	NEP		Entrez le multiplicateur :
6		0	3	NEP	3	
7.init		0	3	1		
7.test	true	0	3	1		
8		3	3	1		
7.incr		3	3	2		
7.test	true	3	3	2		
8		9	3	2		
7.incr		9	3	3		
7.test	true	9	3	3		
8		18	3	3		
7.incr		18	3	4		
7.test	true	18	3	4		
8		30	3	4		
7.incr		30	3	5		
7.test	false	30	3	5		
9		30	3	NEP		
10		30	3	NEP		La somme des 4 premiers multiples est :
11		30	3	NEP		30
12		30	3	NEP		

La première ligne du tableau montre l'état initial avant exécution. L'instruction `for` a trois morceaux intervenant à des moments différents, c'est pourquoi on a distingué les effets de l'initialisation, du test de la condition `i<=4` (quand la condition vaut `true`, la boucle se poursuit, quand elle vaut `false`, la boucle s'arrête), et de la modification de la variable `i`. Les autres types de boucle (`while` et `do ... while`) ne comportent qu'une partie test.

La ligne 9 est prise en compte en ceci que l'accolade ferme un bloc, ce qui met fin à l'existence de la variable `i` qui est locale à ce bloc. On aurait pu de même prendre en compte la ligne 13 qui met fin aux variables `total` et `x`.

Un tel tableau relate une exécution donnée. Si on change la valeur `x` entrée ligne 6, il faut changer tout le tableau. On voit sur cet exemple que le corps de la boucle, la ligne 8, est exécutée 4 fois au cours du programme.

3 Les tableaux et la mémoire

Avec les tableaux arrive une nouvelle instruction qui a un effet sur la mémoire : l'instruction `new`. Elle a pour effet d'allouer une mémoire privée pour le tableau. Cette mémoire n'est pas identifiée par un nom mais par une adresselocalisée dans une partie de la mémoire différente de celle qui contient les variables.

La mémoire privée du programme est ainsi divisée en deux parties :

— la **Pile** (anglais : Stack) qui contient les mémoires privées des variables déclarées dans les méthodes.

— le **Tas** (anglais : Heap) qui contient les mémoires privées des tableaux (et comme on le verra plus tard, des objets).

Les emplacements dans la pile sont repérés par des noms, et ceux dans le tas par des adresses. Que sont les adresses ? Des identifiants uniques qui permettent de repérer un emplacement précis de la mémoire. On verra bientôt qu'en Java, on peut (plus ou moins) afficher les adresses et faire des tests d'égalité des adresses.

Tout ce que nous avons dit à propos des variables dans la section précédente est vrai pour toutes les variables, y compris celles des types tableaux. Mais l'instuction `new` a un comportement

tout à fait nouveau que nous allons présenter au moyen d'un exemple simple.

3.1 Exemple de création d'un tableau

```
public class Memoire{
   public static void main(String[] args){
      boolean[] tab;
      tab = new boolean[3];
      tab[0] = true;
      System.out.println(tab);
   }
}
```

La ligne 3 a pour effet de réserver une mémoire privée pour la variable `tab` dans la pile. L'état de la mémoire après exécution de cette ligne est le suivant :

Pile	Tas
tab : ?	

L'instruction `new` a deux effets :

— réserver un espace mémoire suffisament grand pour 3 booléens (au moins 3 bits) dans le tas pour en faire la mémoire privée du tableau.

— renvoyer l'adresse de cet espace mémoire.

L'affectation de la ligne 4 met la valeur renvoyée par `new`, c'est-à-dire l'adresse de la mémoire privée du tableau, dans la mémoire privée de la variable `tab`. Les variables de type tableau contiennent l'adresse de la mémoire privée du tableau. La taille de l'adresse ne dépend pas de la taille du tableau et c'est pourquoi l'on n'a pas besoin de spécifier la taille du tableau lors de la déclaration de la variable.

Après exécution de la ligne 4, la mémoire est dans l'état suivant.

Pile	Tas
tab : adresse-1	boolean[] adresse-1 0 false 1 false 2 false

La ligne suivante est une affectation à une case du tableau. Cette affectation change le contenu d'une case du tableau dans la mémoire privée du tableau, dans le tas. L'état mémoire devient :

Pile	Tas
tab : adresse-1	boolean[] adresse-1 0 true 1 false 2 false

La dernière ligne du programme consiste à afficher le contenu de la mémoire privée de tab, autrement dit, ce que nous avons appelé `adresse-1`.

Une exécution réelle affiche la chaîne suivante.

```
> java Memoire
[Z@15db9742
```

Cet affichage ne nous dit pas grand chose : c'est juste un nom donné par la machine virtuelle Java à l'emplacement de la mémoire privée du tableau.

3.2 Affectation entre tableaux

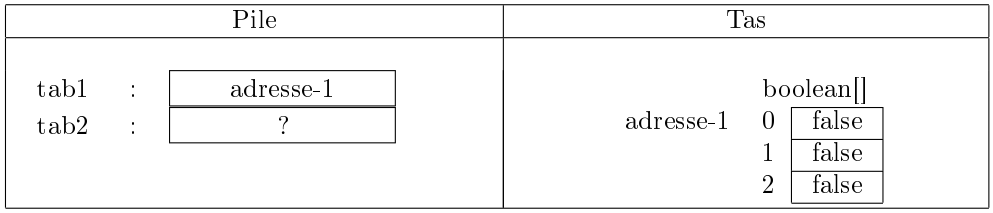
Lorsque dans un programme, on réalise une affectation entre variables de type tableau, cela recopie l'adresse de la mémoire privée d'une variable dans la mémoire privée de l'autre. C'est la même chose lorsque l'on fait une affectation entre deux variables de type `int` : l'entier stocké dans la mémoire privée d'une variable est recopié dans la mémoire privée de l'autre variable.

Cette opération de copie d'adresse ne change rien dans le tas et ne crée pas de nouveau tableau. Seule l'instruction `new` crée des tableaux.

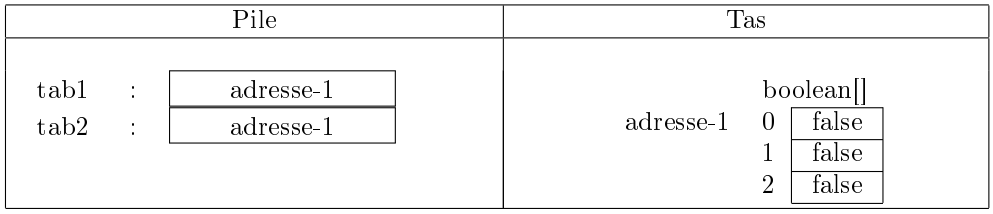
Voyons un exemple.

```
1  public class Memoire2{
2     public static void main(String[] args){
3        boolean[] tab1, tab2;
4        tab1 = new boolean[3];
5        tab2 = tab1;
6        System.out.println(tab1);
7        System.out.println(tab2);
8        tab1[0] = true;
9        System.out.println(tab2[0]);
10    }
11 }
```

Après exécution des lignes 3 et 4, la mémoire est dans l'état suivant :



La ligne 5 consiste à lire le contenu de la mémoire privée de la variable `tab1` et à l'écrire dans la mémoire privée de la variable `tab2`. Cela conduit à l'état suivant.



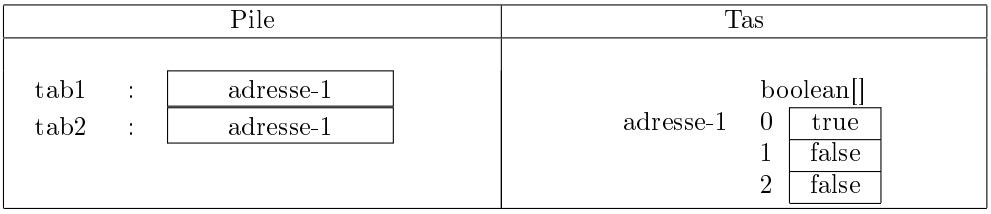
Il y a maintenant, deux variables différentes qui contiennent l'adresse du seul tableau qui existe dans le tas. Du coup, il y a deux façons différentes d'accéder à une case de ce tableau, par exemple celle d'indice 0. Ces deux façons s'écrivent respectivement `tab1[0]` et `tab2[0]`. Ce sont deux manières différentes de désigner le même emplacement dans le tas.

Les lignes 6 et 7 affichent les adresses contenues dans `tab1` et `tab2`, ce qui permet de constater que ce sont une seule et même adresse.

```
> java Memoire2
[Z@15db9742
[Z@15db9742
true
```

La ligne 8 modifie `tab1[0]`, en y mettant `true` à la place de `false`. Ce faisant, le programme modifie aussi `tab2[0]`, puisque dans l'état courant du programme, `tab1[0]` et `tab2[0]` sont une seule et même chose.

En fin de programme, l'état de la mémoire est le suivant.



A la ligne 9, c'est donc `true` qui est affiché.

3.3 Comparaisons d'adresses

Les opérateurs `==` et `!=` permettent de comparer le contenu de deux variables de type tableau. Ce sont donc les adresses que ces variables tableaux contiennent qui sont comparées, et non pas le contenu des cases des tableaux.

Prenons un exemple avec deux tableaux aux contenus identiques.

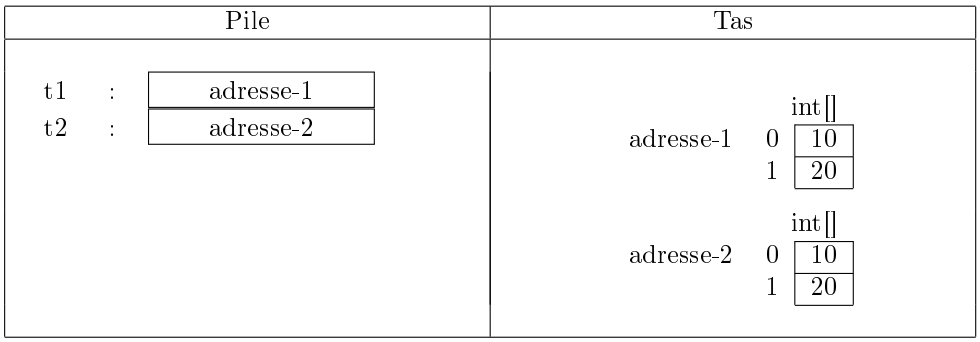
```
public class CompTab{
   public static void main(String[] args){
      int[] t1 = {10, 20};
      int[] t2 = {10, 20};
      if (t1 == t2){
         System.out.println("t1 et t2 sont égaux");
      }else{
         System.out.println("t1 et t2 ne sont pas égaux");
      }
      if (t1 != t2){
         System.out.println("t1 et t2 sont différents");
      }else{
         System.out.println("t1 et t2 ne sont pas différents");
      }
   }
}
```

Notons que la ligne `int[] t1 = {10, 20};` crée un nouveau tableau, comme s'il y avait un `new int[2]` suivi de deux affectations pour mettre 10 et 20 dans les deux cases du tableau. Dans cet exemple, il y a donc deux variables et deux tableaux à deux adresses différentes. Voyons ce que donne l'exécution de ce programme.

```
> java CompTab
t1 et t2 ne sont pas égaux
t1 et t2 sont différents
```

Les deux opérateurs `==` et `!=` ne comparent pas les deux tableaux contenus dans le tas, mais juste les deux adresses stockées dans la pile, dans les deux espaces privés des variables `t1` et `t2`.

Pour bien illustrer le cas, voyons l'état de la mémoire après déclaration des deux variables.



Voyons maintenant un cas où deux tableaux sont considérés comme égaux par ces opérateurs. Il suffit de reprendre et d'augmenter le programme `Memoire2` de la section précédente.

```
public class Memoire2Bis{
   public static void main(String[] args){
      boolean[] tab1, tab2;
      tab1 = new boolean[3];
      tab2 = tab1;
      System.out.println(tab1);
      System.out.println(tab2);
      tab1[0] = true;
      System.out.println(tab2[0]);
      if (tab1 == tab2){
         System.out.println("tab1 et tab2 sont égaux");
      }else{
         System.out.println("tab1 et tab2 ne sont pas égaux");
      }
   }
}
```

L'exécution produit l'affichage suivant :

```
> java Memoire2Bis
[Z@15db9742
[Z@15db9742
true
tab1 et tab2 sont égaux
```

Une autre façon d'expliquer le comportement de l'opérateur `==` est de dire qu'il teste si `tab1` et `tab2` sont deux moyens différents d'accéder au même tableau dans le tas.

3.4 Évolutions de l'état de la mémoire

Il y a trois opérations qui modifient l'état de la mémoire :
— L'allocation de mémoire privée à un élément du programme.
— L'écriture dans une mémoire privée d'un élément du programme.
— La désallocation d'une mémoire privée.

Pour ce qui est de la pile, voyons quelles instructions apportent les modifications.
— L'allocation de mémoire privée est l'effet de l'exécution d'une déclaration de variable.
— L'écriture dans une mémoire privée d'une variable est l'effet d'une affectation dans cette variable. Le nom de la variable est à gauche de l'affectation, à droite, il y a le calcul qui détermine ce qui est écrit dans cette mémoire privée.
— La désallocation d'une mémoire privée d'une variable a lieu lorsque la variable cesse d'exister, à la fin de l'exécution du bloc d'instructions où elle est déclarée.

Voyons maintenant ce qui modifie l'état du tas.

— L'allocation de mémoire privée est l'effet de l'exécution d'une instruction `new`. Il s'agit de la mémoire privée d'un tableau. L'initialisation d'une variable avec un tableau écrit en accolade a le même effet. C'est un `new` déguisé.

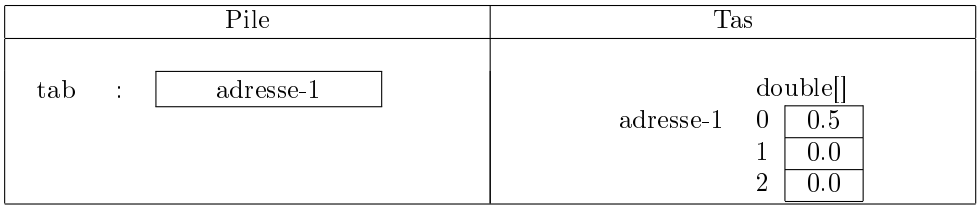
— L'écriture dans une mémoire privée d'un tableau est l'effet de l'affectation d'une valeur à une case de ce tableau. C'est donc la même opération que pour la pile, mais la différence est qu'à gauche du signe =, il n'y a pas que le nom d'une variable mais aussi des crochets et un numéro de case (directement ou via un calcul). Pour savoir si une affectation écrit dans la pile ou dans le tas, il suffit de regarder s'il y a des crochets ou pas àgauche du =.

— La désallocation d'une mémoire privée d'un tableau a lieu lorsque ce tableau n'est plus accessible depuis le programme (il n'y a aucun moyen d'accès, plus de variable qui contienne l'adresse de ce tableau) et que par ailleurs, le programme a besoin de mémoire pour créer de nouveaux tableaux. En pratique, on ne sait jamais bien quand a lieu la désallocation et on peut tout simplement ne pas s'en préoccupper et supposer que cette opération n'a jamais lieu.

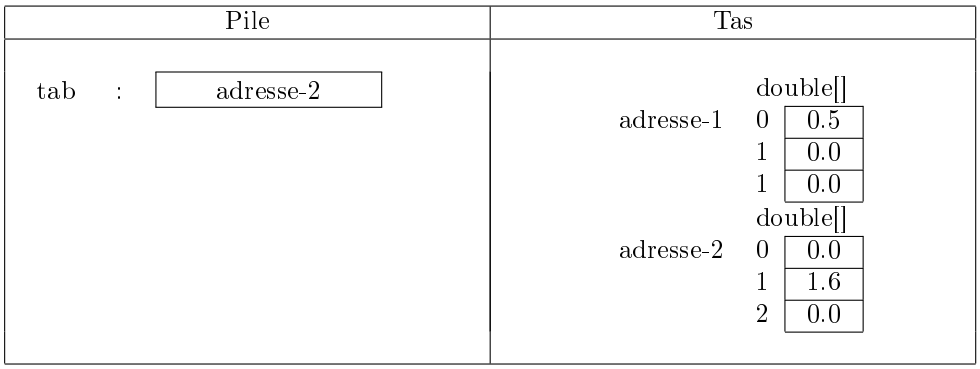
Voyons un exemple de programme contenant un tableau qui n'est plus accessible.

```
1 public class Memoire3{
2    public static void main(String[] args){
3       double[] tab;
4       tab=new double[3];
5       tab[0]=0.5;
6       tab=new double[3];
7       tab[1]=1.6;
8    }
9 }
```

Après exécution de la ligne 5, l'état de la mémoire est le suivant.



Ligne 6, il y a création d'un nouveau tableau par l'instruction `new` à une nouvelle adresse, `adresse-2`. Cette adresse est affectée à la variable `tab`. Après ces opérations, l'état de la mémoire est le suivant.



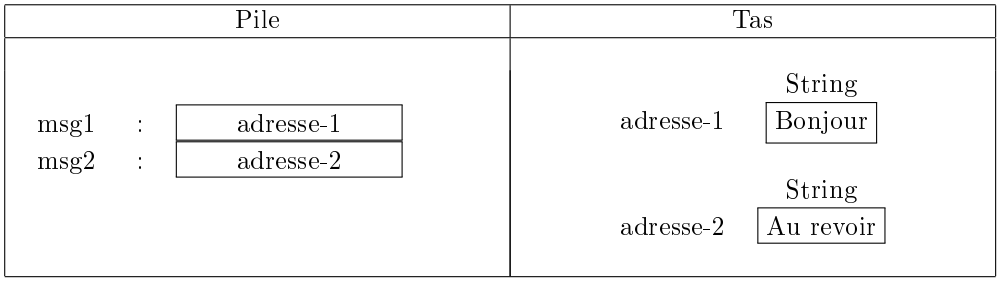
Dans cet état, le programme ne peut plus accéder au tableau dont la mémoire privée est à `adresse-1`, car il n'y a plus de variable qui contienne cette adresse. Il n'existe pas en Java d'instruction qui permettrait de retrouver ce tableau à partir de cet état. Dès lors, que ce tableau soit présent en mémoire ou pas, cela ne change pas le résultat de l'exécution et c'est pourquoi le système peut récupérer cet emplacement s'il est à court de mémoire.

4 Les String et la mémoire

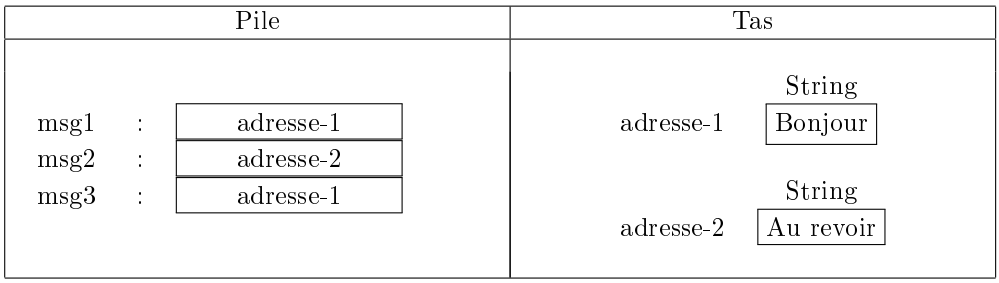
Les `String`, comme les tableaux, sont stockés dans le tas et les variable de type `String` contiennent l'adresse (dans le tas) de la mémoire privée d'une chaîne de caractères.

```
1  public class Memoire4{
2     public static void main(String[] args){
3        String msg1 = "Bonjour";
4        String msg2 = "Au revoir";
5        String msg3 = msg1;
6     }
7  }
```

Après exécution de la ligne 4, l'état de la mémoire est le suivant.



La ligne 5 déclare une nouvelle variable `msg3` et lui affecte le contenu de `msg1`, c'est-à-dire `adresse-1`. L'état de la mémoire en fin de programme est donc le suivant.



Dans cet état `msg1` et `msg3` sont deux moyens d'accéder à la même chaîne de caractères, à la même mémoire privée.

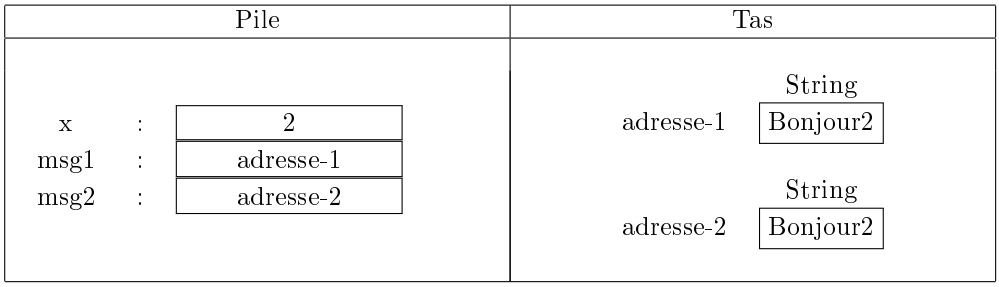
Comme pour les tableaux, `==` et `!=` testent si deux expressions Java désignent la même mémoire privée et non pas si les deux expressions désignent la même suite de caractères. En effet, il est possible d'avoir deux mémoires privées contenant la même suite de caractères. Dans ce cas, `==` et `!=` considèrent ces deux chaînes comme différentes. La programme suivant illustre cette situation.

```
1  public class Memoire5{
2     public static void main(String[] args){
3        int x = 2;
4        String msg1 = "Bonjour2";
5        String msg2 = "Bonjour"+x ;
6        System.out.println("<"+msg1+">");
7        System.out.println("<"+msg2+">");
8        if (msg1==msg2){
9           System.out.println("chaîne égales selon ==");
10       }else{
11          System.out.println("chaîne différentes selon ==");
12       }
13    }
14 }
```

L'exécution du programme donne le résultat suivant :

```
> java Memoire5
<Bonjour2>
<Bonjour2>
chaîne différentes selon ==
```

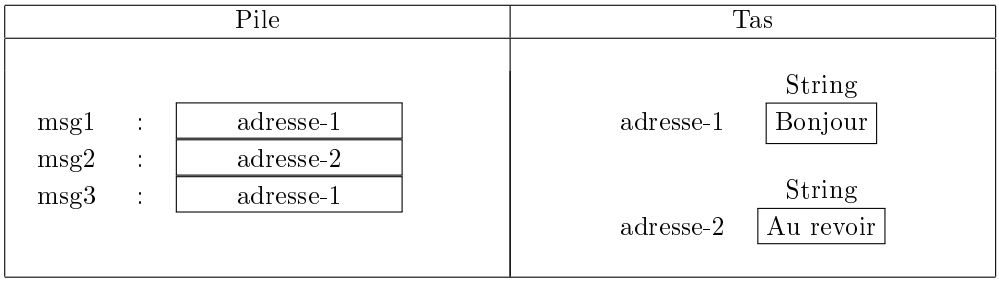
L'état de la mémoire à partir de la fin de l'exécution de la ligne 5 est le suivant.



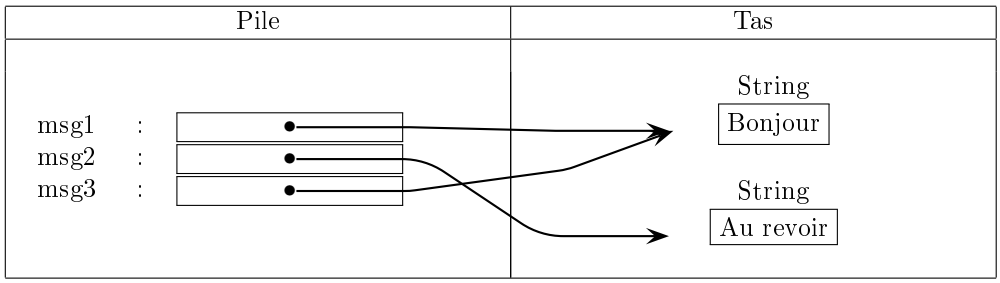
Ceci explique pourquoi il ne faut pas utiliser `==` et `!=` pour comparer deux chaînes de caractères si l'on veut comparer leurs contenus et non pas l'adresse de leurs mémoires privées. Nous reviendrons sur ce point dans le chapitre dédié aux `String`.

5 Représentation des adresses par des flèches

Lorsqu'on représente un état de la mémoire, l'utilisation des adresses n'est pas toujours très parlante. On préfère souvent représenter les références par des flèches. Ce qui compte, dans une flèche, c'est son point d'arrivée. Prenons un exemple d'état mémoire vu précédemment.



En remplaçant les adresses par des flèches, cela nous donne le schéma suivant.



6 Outil de dessin de la mémoire

Le site `https://pythontutor.com/` vous permet de dessiner la mémoire à chaque étape de l'exécution d'un petit programme. Cet outil a des limitations, notamment, le programme ne

doit pas contenir de saisies au clavier et est limité en nombre de lignes. Il peut être très utile pour comprendre les principes exposés dans ce chapitre.

Dans Pythontutor, la pile s'appelle `frames` et le tas `objects`. Vous avez le choix entre l'utilisation de labels textuels ou de flèches pour représenter des adresses du tas.

Voici le schéma produit par Pythontutor pour l'exemple donné à la section précédente.

7 Résumé du chapitre

— La mémoire est divisée en deux parties : la pile et le tas
— La pile contient les mémoires privées des variables
— Le tas contient les mémoires privées des tableaux et des objets, dont les chaînes de caractères.
— Les mémoires privées des variables de type tableau ou de type String contiennent une adresse désignant un espace dans le tas où est enregistrée le contenu du tableau ou de la chaîne.
— Pour les tableaux et les chaînes, `==` et `!=` comparent les adresses et non le contenu des tableaux et des chaînes.