Chapitre 10

Exécution de programmes et mémoire (2)

Nous avons vu des principes d'organisation de la mémoire utilisée pour l'exécution d'un programme en deux espaces : la pile et le tas, les variables étant stockées dans la pile, les tableaux et les chaînes de caractères étant stockées dans le tas.

Nous avons vu que les variables de type String ou de type tableau contiennent non pas directement la chaîne ou le tableau, mais l'adresse d'un emplacement dans le tas où la chaîne ou le tableau est stockée.

Il nous reste à compléter le modèle en détaillant ce qui se passe lorsqu'une méthode est appelée : où sont stockés les paramètres et comment certaines valeurs sont utilisées par plusieurs méthodes.

10.1 Méthodes et mémoire

Un principe fondamental du langage Java est que chaque méthode a ses propres variables et aucune variable ne peut être partagée par deux méthodes différentes.

Pour chaque exécution d'une méthode, une portion de la pile lui est affectée pour y stocker ses paramètres et ses variables. Ses variables sont celles qui sont déclarées dans son corps.

Le calcul d'un appel de méthode se fait en plusieurs temps :

— Calcul de la valeur des paramètres

— Affectation d'une mémoire privée dans la pile. Dans cette mémoire privée les paramètres sont stockés avec leurs noms et leurs valeurs calculées à l'étape précédente.

— Exécution des instructions du corps de la méthode.

— Une fois l'exécution de ces instructions terminée, la mémoire privée de la méthode dans la pile est libéré. La valeur calculée est renvoyée à la méthode contenant l'appel (sauf méthode void, pas de valeur renvoyée dans ce cas).

Voyons un exemple d'exécution d'un programme comportant un appel à une méthode.

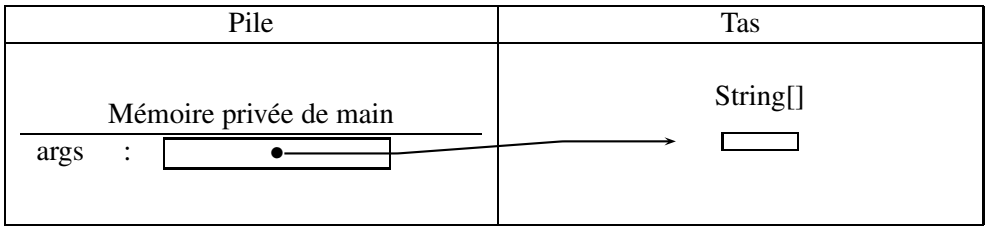
10.1.1 Appel de méthode : premier exemple

```
1 public class LeMin{
2    public static int min(int nb1, int nb2){
3       int res;
4       if (nb1<nb2){
5          res = nb1;
6       }else{
```

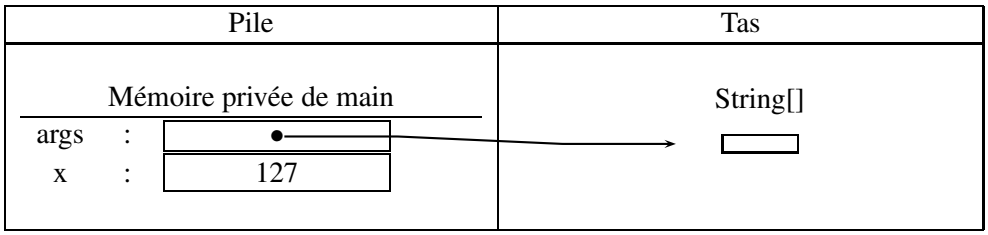
```
7            res = nb2;
8         }
9         return res;
10    }
11    public static void main(String[] args){
12        int x;
13        x = 127;
14        x = min(x,12);
15        x = x *x;
16        System.out.println(x);
17    }
18 }
```

Le programme commence par l'exécution de la méthode main. Pour réaliser cette exécution, une mémoire privée est allouée pour la méthode main. Cette mémoire contient en début d'exécution le paramètre de la méthode qui s'appelle `args` et qui contient l'adresse d'un tableau de String en mémoire. Nous supposons ici que c'est un tableau vide, sans case, qui pourrait être créé par l'instruction `new String[0]`.

Cet état initial peut se représenter comme suit.



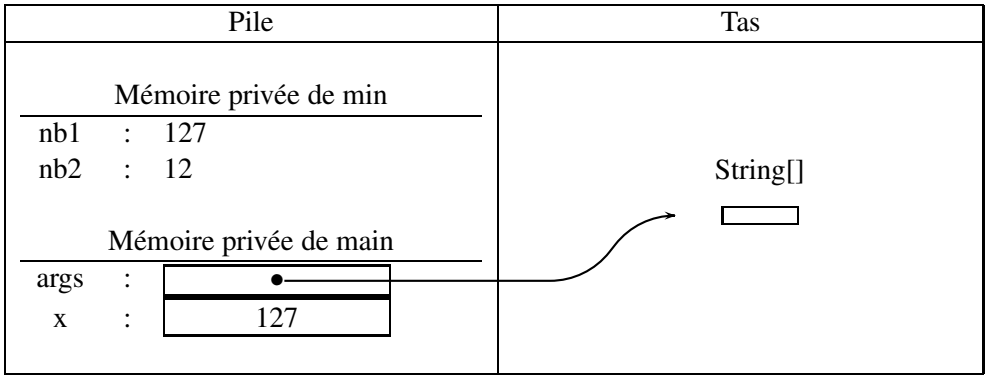
L'exécution commence par la ligne 12 : la déclaration de la variable x conduit à l'allocation d'un emplacement mémoire pour `x` à l'intérieur de la mémoire privée de `main`. La variable est une variable locale à la méthode `main`. Elle existe à partir du moment où la ligne 12 est exécutée et son existence se poursuivra jusqu'à la fin de l'exécution de la méthode main (fin de la ligne 16). La ligne 13 est exécutée qui va mettre dans l'espace alloué à `x` la valeur `127`. À la fin de l'exécution de la ligne 13 la mémoire est donc dans l'état suivant.



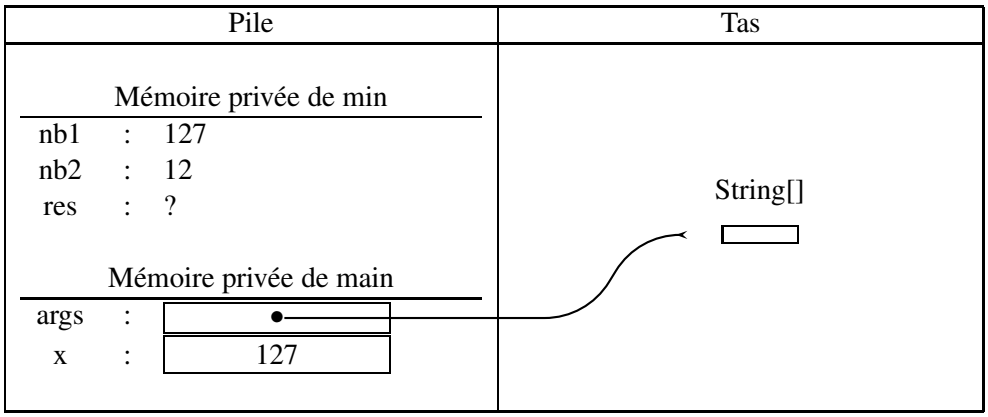
L'exécution se poursuit avec la ligne 14. Il s'agit d'une affectation. La partie droite doit être calculée, puis le résultat stocké dans la mémoire privée de la varaible `x`. La partie droite est un appel de méthode : le calcul se déroule en qautre étapes, comme expliqué ci-dessus.

— Étape 1 : calcul des valeurs des paramètres : premier paramètre `x`, valeur : 127. Deuxième paramètre 12, valeur : 12.

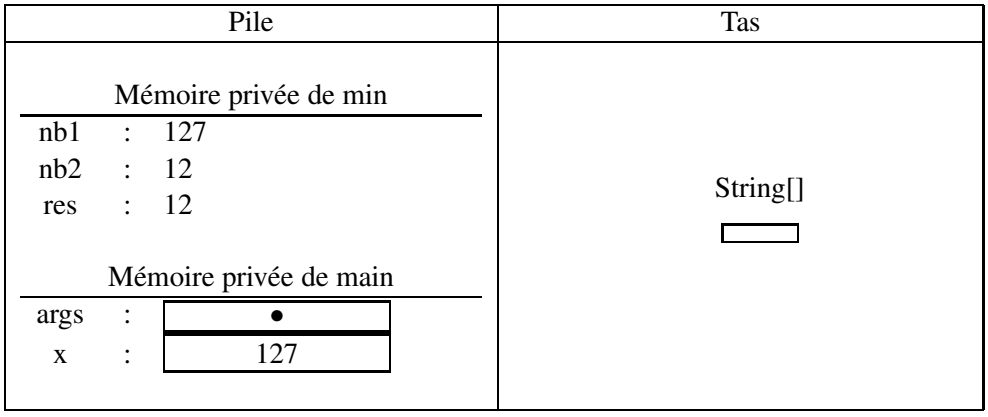
— Étape 2 : allocation d'une mémoire privée pour la méthode `min` avec les deux paramètres (`nb1` et `nb2`) et leurs valeurs calculée précedemment. Cela nous conduit à l'état suivant.



— Étape 3 : exécution du corps de la méthode. Cela commence par la déclaration de la variable `res`. Cette variable ets locale à la méthode `min`. La mémoire privée de cette variable est dsituée dans la mémoire privée de la méthode `min`.

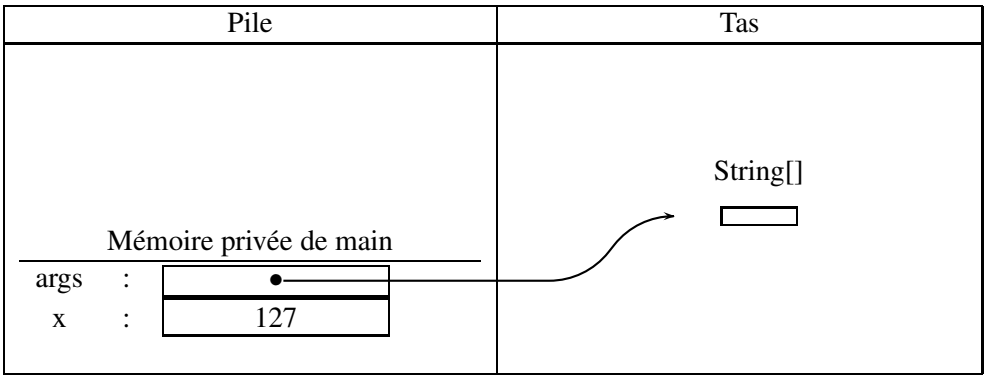


L'exécution se poursuit avec le calcul de la condition du `if` de la ligne 4. Cette condition compare les valeurs des paramètres `nb1` et `nb2`. Comme 127 n'est pas plus petit que 12, la condition est fausse et c'est le cas `else` qui s'exécute. La ligne 7 est exécutée : la valeur de `nb2`, 12, est affectée à la mémoire privée de `res`. L'état est alors :

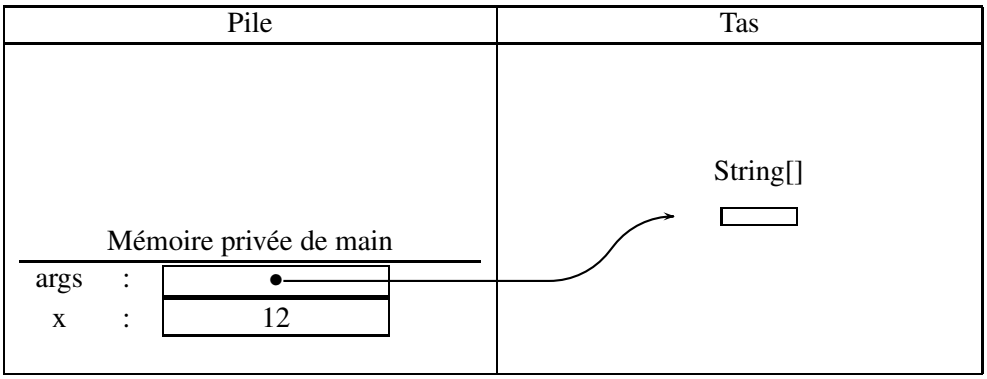


Le `if` est terminé, il reste à exécuter la ligne 9. Celle-ci renvoie la valeur de `res`, qui est 12. L'étape 3 du calcul de l'appel de la méthode est terminé.

— Étape 4 : la mémoire privée de `min` est effacée, rendue disponible pour la mémoire privée d'une autre méthode. La valeur 12 précédemment calculée est renvoyée.



On revient alors à la ligne 14 dont l'exécution a commencé depuis un moment et qu'il s'agit de terminer. L'appel de `min` a renvoyé la valeur 12. C'est cette valeur qui est stockée dans la mémoire privée de `x`.



La ligne 15 s'exécute : `x` prend la valeur 144. La ligne 16 affiche la valeur de `x`, 144.

10.1.2 Appel de méthode : deuxième exemple

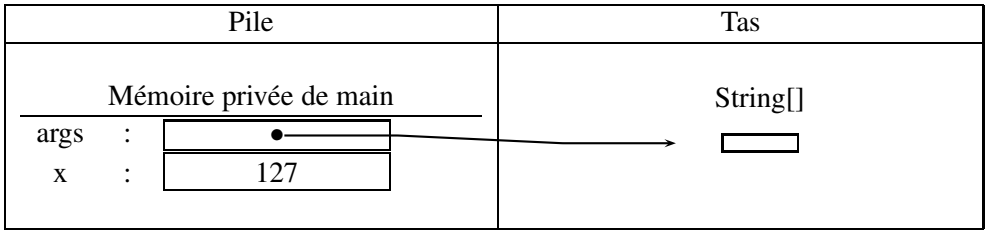
Nous allons voir à présent un second exemple où il y a deux variables portant le même nom dans deux méthodes différentes. Il s'agit en fait du même programme dans lequel la variable `res` de la méthode `min` a été renommé en `x`.

```
1 public class LeMinBis{
2    public static int min(int nb1, int nb2){
3       int x;
4       if (nb1<nb2){
5          x = nb1;
6       }else{
7          x = nb2;
```

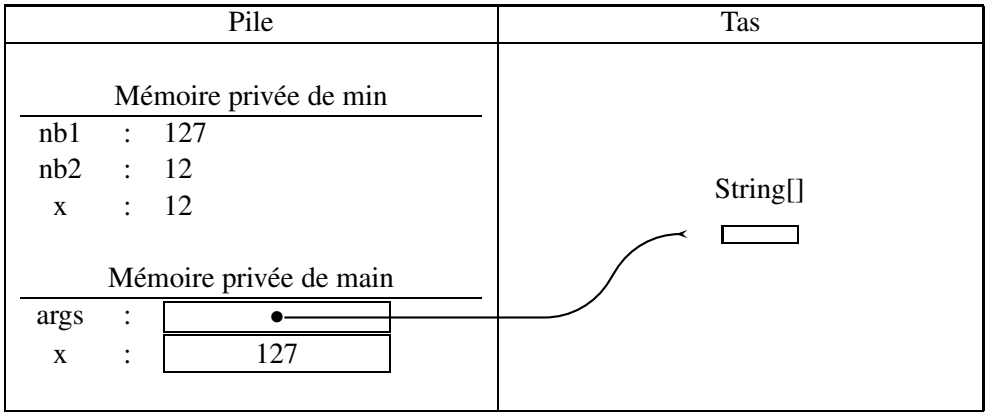
```
8         }
9         return x;
10    }
11    public static void main(String[] args){
12        int x;
13        x = 127;
14        x = min(x,12);
15        x = x *x;
16        System.out.println(x);
17    }
18 }
```

Le changement de nom de la variable ne change rien à l'exécution : les deux déclarations (lignes 3 et 12) conduisent à deux allocations de mémoire différentes dans les deux mémoires privées des deux méthodes `main` et `min`. Ce sont deux espaces différents qui évoluent indépendamment. Une affectation dans la méthode `main` (exemple : ligne 14) change le `x` de la mémoire privée de `main` et une affectation dans la méthode `min` (exemple : ligne 5) change le `x` de la mémoire privée de `min`.

Nous n'allons pas détailler tous les états mémoires successifs, mais nous allons en donner deux. Après exécution de la ligne 13, l'état est la suivant.



Au cours de l'appel à `min(x,12)` (ligne 14), après calcul des valeurs des paramètres, allocation d'une mémoire privée pour `min` avec les deux paramètres, déclaration de la variable locale `x`, évaluation de la condition du if (résultat : `false`) et exécution du cas `else` (ligne 7), l'état de la mémoire est le suivant.



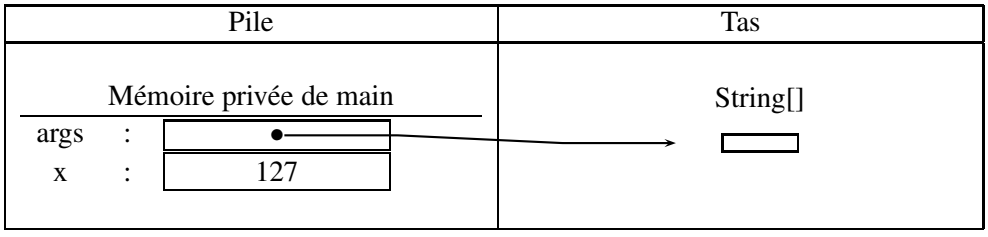
On voit ici qu'il y a deux variables `x` avec deux valeurs différentes (127 et 12).

10.1.3 Trois méthodes

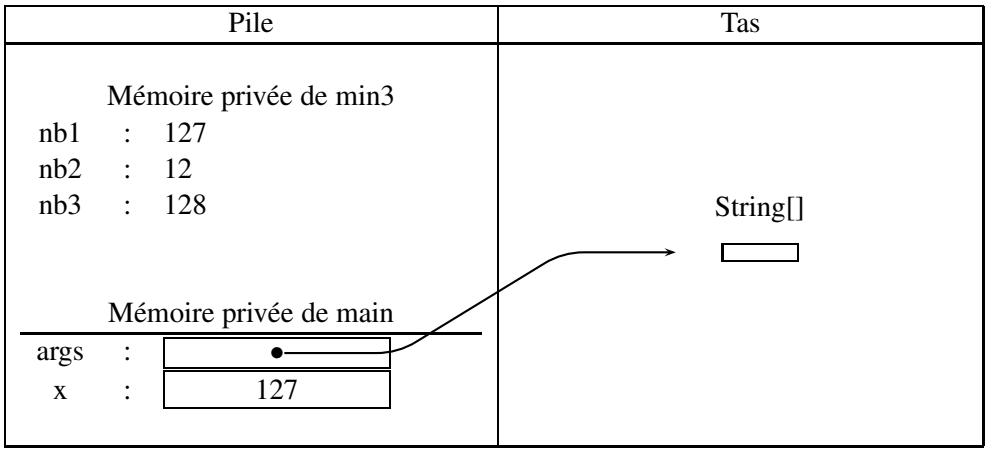
Nous allons voir un nouvel exemple où une méthode est appelée dans le corps d'une autre méthode, elle-même appelée par le `main`.

```
1  public class MinDe3{
2     public static int min2(int nb1, int nb2){
3        int res;
4        if (nb1<nb2){
5           res = nb1;
6        }else{
7           res = nb2;
8        }
9        return res;
10    }
11    public static int min3(int nb1, int nb2, int nb3){
12       int res;
13       res = min2(nb2,nb3);
14       if (nb1<res){
15          res = nb1;
16       }
17       return res;
18    }
19    public static void main(String[] args){
20       int x;
21       x = 127;
22       x = min3(x,12,x+1);
23       x = x *x;
24       System.out.println(x);
25    }
26 }
```

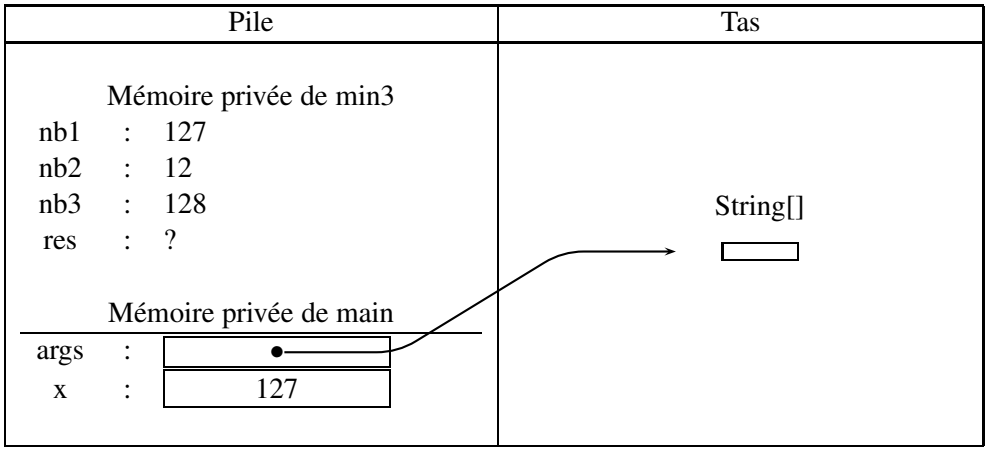
Le début de l'exécution du programme est identique à celui des deux exemples déjà vus. Après exécution de la ligne 21, l'état de la mémoire est :



Ligne 22, il faut exécuter l'appel `min3(x,12,x+1)`. L'étape 1 consiste à calculer les valeurs des trois paramètres. Ce sont respectivement 127, 12 et 128. Une mémoire privée pour `min3` est alors allouée avec les mémoires privées des trois paramètres contenant leurs valeurs respectives.

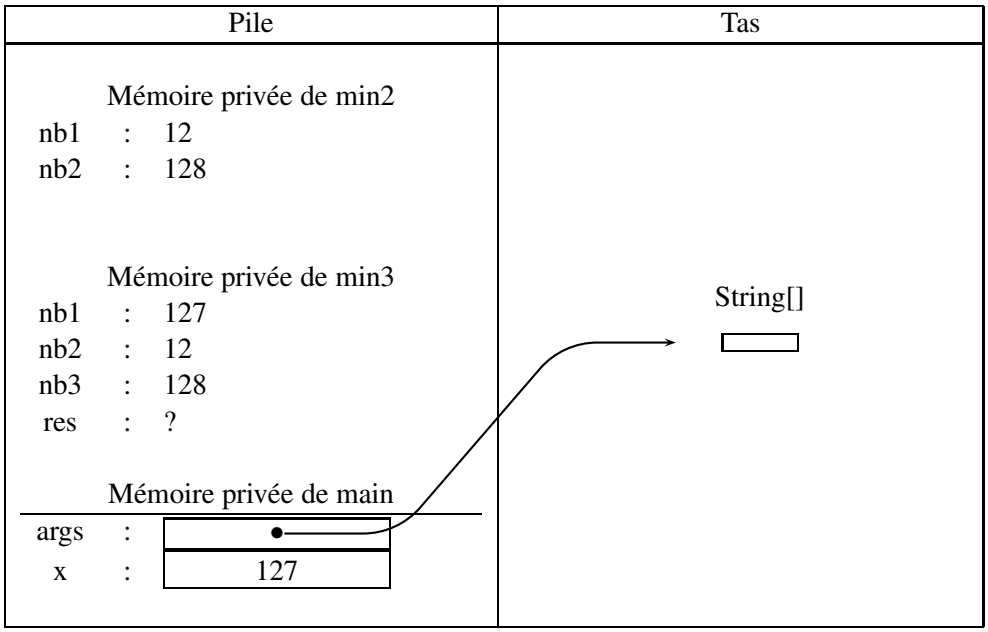


L'étape 3 consiste à exécuter les instructions de `min3` en commençant par la ligne 12. Une mémoire privée est allouée pour la varaible local `res`.



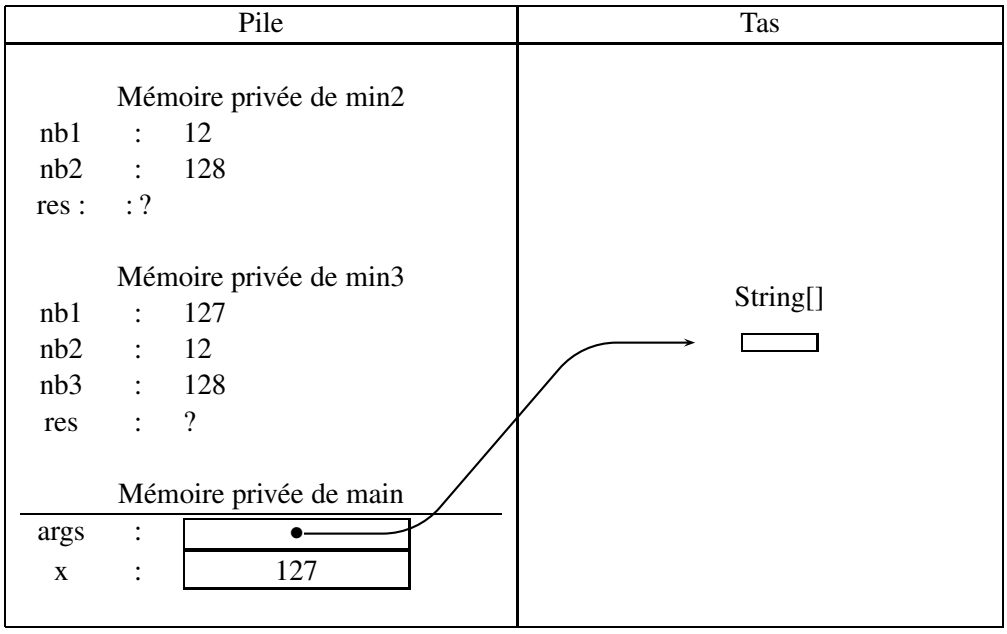
A la ligne suivante (ligne 13), il y aune affectation. Il faut commencer par calculer l'appel `min2(nb2,nb3)`. L'étape 1 du calcul coniste à chercher les valeurs des deux paramètres. Le premier paramètre vaut ce qu'il y a dans le paramètre `nb2`, c'est-à-dire 12. Le second vaut ce qu'il y a dans `nb3`, c'est-à-dire 128. L'étape 2 alloue une mémoire privée pour la méthode `min2` contenant les deux mémoires privées des paramètres de `min2`, qui sont `nb1` et `nb2`.

Il convient ici de bien distinguer les paramètres de `min3` et ceux de `min2`, bien que certains d'entre eux aient le même nom. A la ligne 13, comme on est dans le corps de `min3`, `nb2` désigne le paramètre de ce nom dans `min3` et va chercher sa valeur dans la mémoire privée de `min3`. Quant à `nb3`, c'est un nom qui n'apparaît que dans `min3`, il n'y a pas de doute possible.



Dans cette situation, il y a deux `nb1` avec deux valeurs différentes et de même deux `nb2` avec deux valeurs différentes. Dans cette situation, il y a 3 mémoires privées de méthodes correspondant aux trois méthodes différentes du programme. On va commencer l'étape 3 du calcul de l'appel `min2(nb2,nb3)`. On est également en cours de calcul de l'étape 3 de l'appel `min3(x,12,x+1)`.

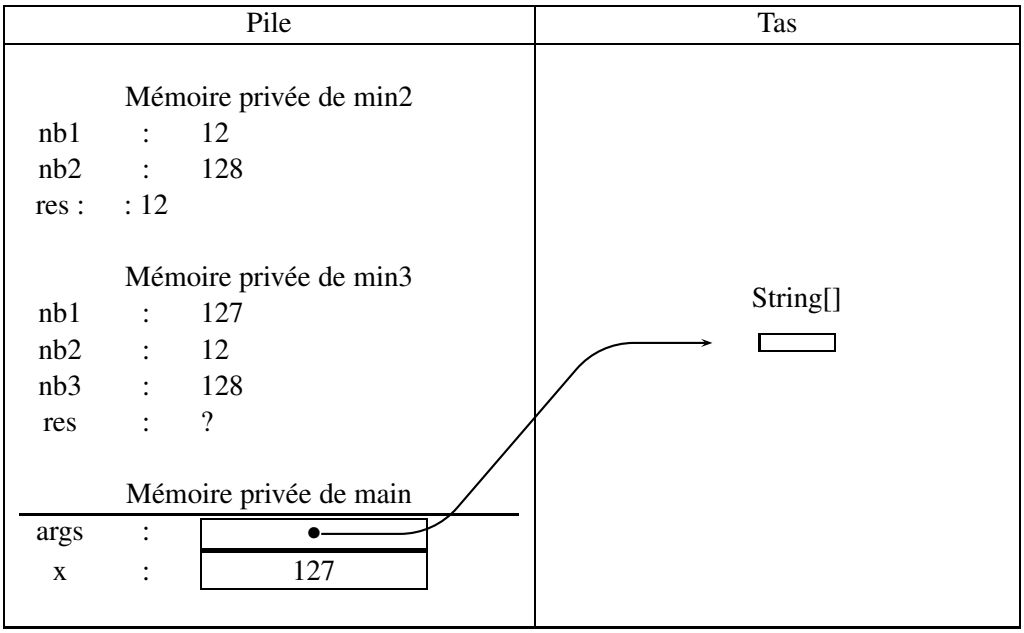
La ligne 3 s'exécute et alloue une mémoire privée pour la variable locale de `min2`.



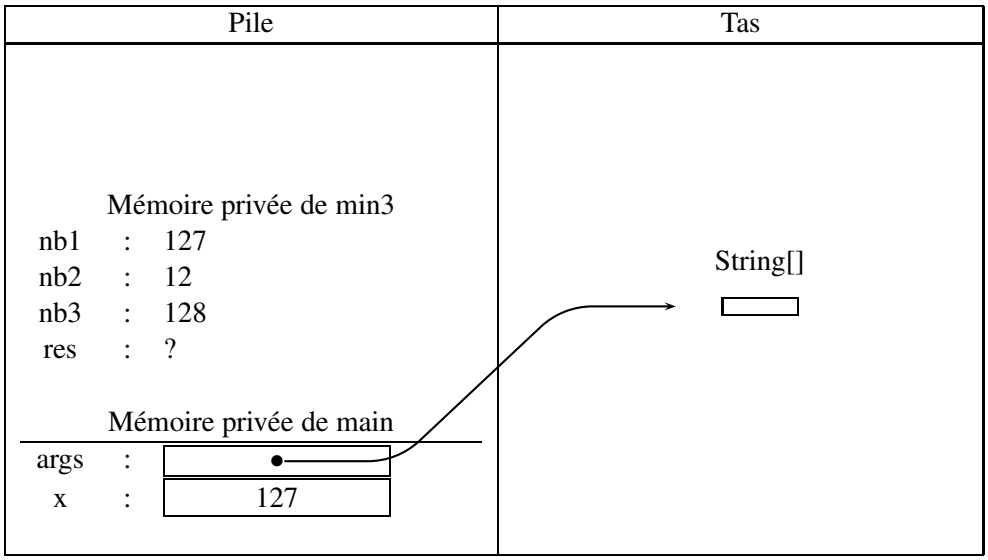
Il y a à présent deux variables locales qui s'appellent `res`. Lorsqu'on exécute une instruction, il n'y a pas de question à se poser : ce n'est que celle qui est le plus haut dans la pile qui peut être utilisée. A un moment donné, même s'il y a plusieurs mémoires privées de méthodes dans la pile, seule celle qui est au plus haut peut être utilisée. Les autres mémoires sont en attente et redeviendront actives ultérieurement en redevenant la mémoire du sommet de la pile.

Au moment où il y a trois mémoires privées de méthodes dans la pile, il y a également trois lignes de code en cours d'exécution, une dans chacune des trois méthodes. Au point relaté ci-dessus, la ligne 4 (mehode `min2` va être exécutée, la ligne 13 (méthode `min3`) est commencée et elle est en attente du résultat renvoyé par `min2` pour s'achever et la ligne 22 (méthode `main`) est commencée et en attente du résultat de `min3` pour s'achever. A ce point, les instructions de `min3` vont toutes s'exécuter avant que la ligne 13 puisse s'achever. Et au moment où la ligne 13 sera terminée, les lignes 14 à 17 seront exécutées avant que la ligne 22 puisse se terminer.

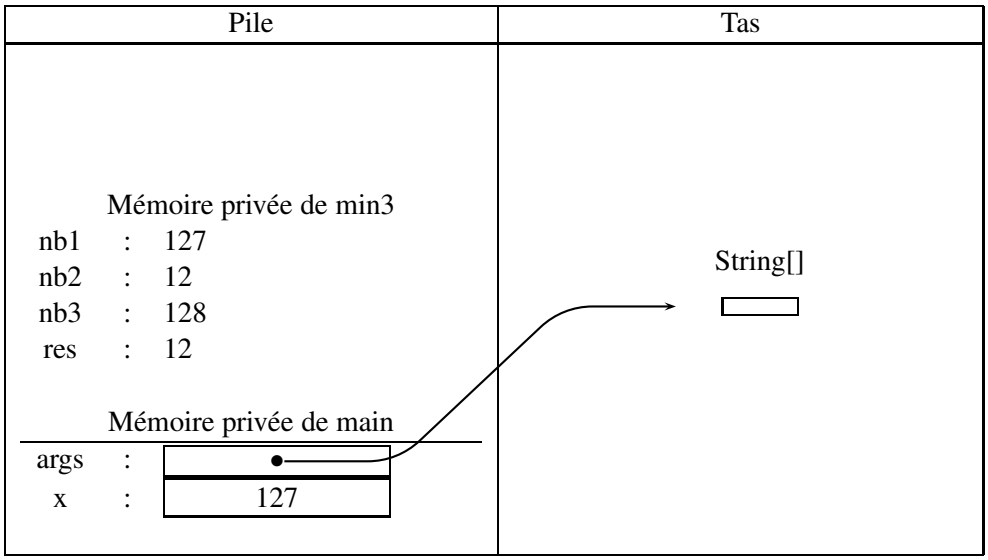
Continuons l'exécution avec la ligne 4 : il y a comparaison de `nb1` et `nb2`. Dans la mémoire de sommet de pile, on rouve respectivement 12 et 128. 12 est plus petit que 128, donc c'est la condition est vraie et c'est la première branche du if (ligne 5) qui est exécutée. C'est la valeur de `nb1`, 12, qui est mise dans `res`.



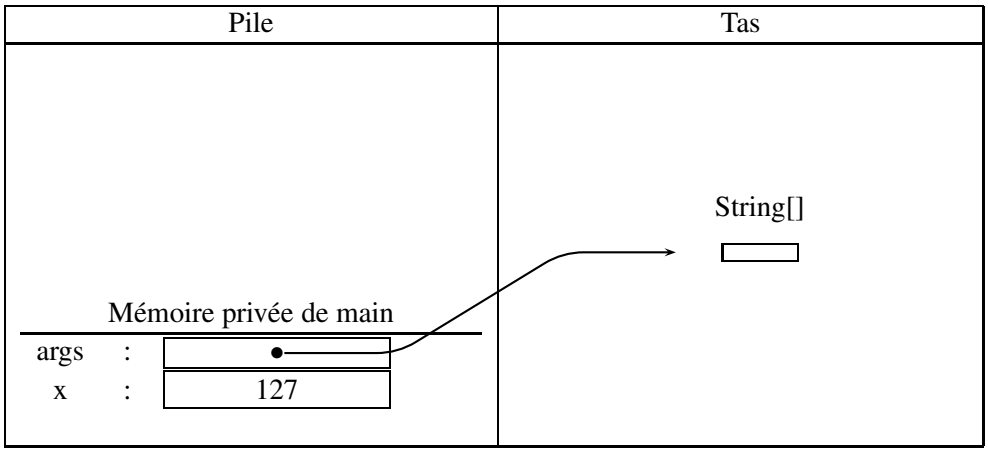
Le if (lignes 4 à 8) est terminé. La ligne 9 doit être exécutée : elle retourne la valeur de `res`, c'est-à-dire 12. La mémoire privée de `min2` est libérée et la mémoire privée de `min3` redevient le sommet de la pile.



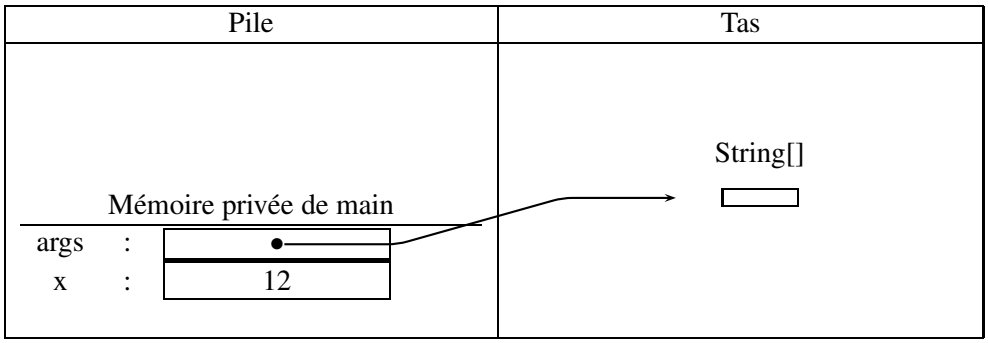
La ligne 13 peut enfin se terminer. La valeur 12 retournée par `min2` est mise dans `res`.



Ligne 14, il y a comparaison de `nb1` (valeur 127) et de `res` (valeur 12). 127 n'est pas plus petit que 12, la condition est fausse. Comme il n'y a pas de cas `else` dans ce `if`, l'exécution du `if` est terminée et l'on passe à la ligne suivante. Celle-ci (ligne 17) renvoie la valeur de `res` : 12. La mémoire privée de `min3` est libérée et la mémoire privée de `main` redevient le sommet de la pile.



La ligne 22 peut s'achever : la valeur 12 est mise dans `x`.



La ligne 23 commence par le calcul de `x*x` qui vaut 144. Cette valeur est mise dans `x`. Puis ligne 24 elle est affichée à l'écran. Ainsi se termine l'exécution du programme.

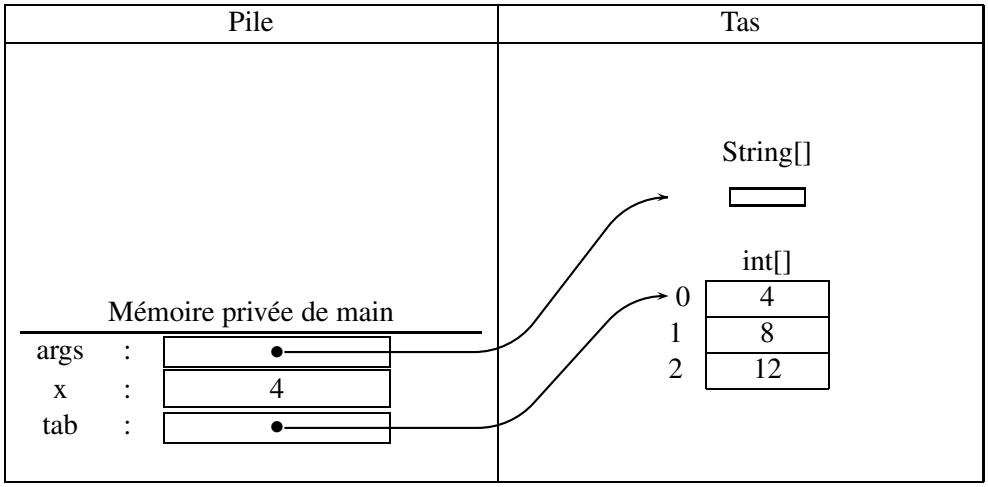
10.2 Passage de paramètre : valeur primitive ou adresse

Une méthode ne peut pas changer le contenu de la mémoire d'une autre méthode. Mais elle peut modifier dans le tas un espace mémoire accessible depuis une autre méthode. Nous pouvons voir la différence entre les valeurs primitives stockées dans la pile et les tableaux stockés dans le tas dans l'exemple suivant.

```
1  public class ChangeXEtTab{
2     public static void main(String[] args){
3        int x = 4;
4        int[] tab = {4, 8, 12};
5        changeXEtTab(x,tab);
6        System.out.println(x);
7        System.out.println(tab[0]);
8     }
9     public static void changeXEtTab(int valint, int[] valtab){
10       valint = valint *10;
11       valtab[0] = valtab[0] *10;
```

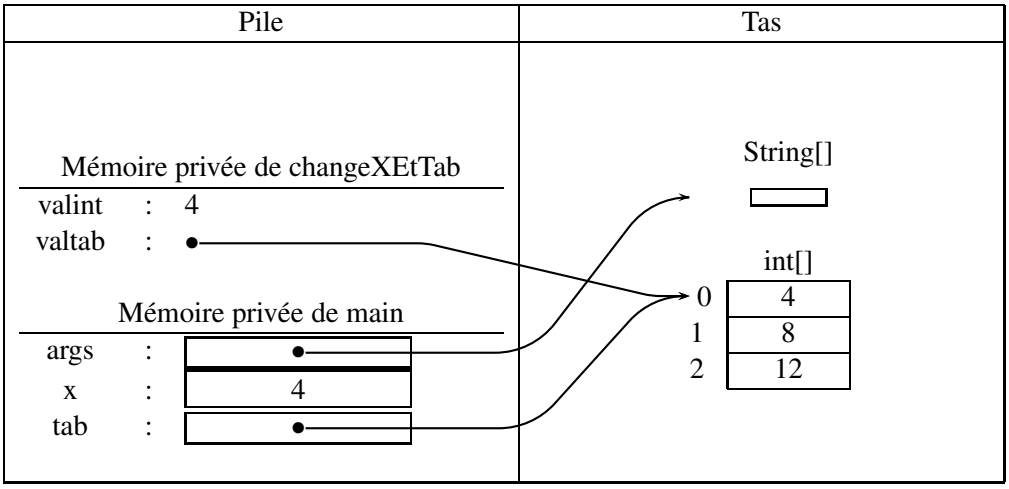
```
12      }
13  }
```

L'exécution du programme commence ligne 3. Après axécution des lignes 3 et 4, l'état de la mémoire est le suivant.



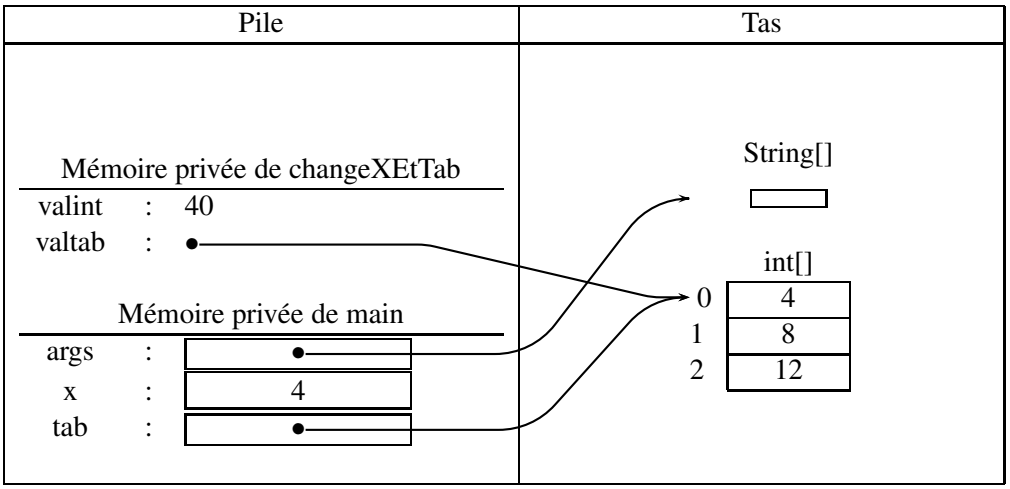
A la ligne suivante, il y a un appel de méthode. l'étape 1 consiste à calculer les valeurs des paramètres. La valeur de `x` est 4 et celle de `tab` est l'adresse du tableau d'int dans la tas, matérialisée par une flèche vers la première case de ce tableau. La valeur qui sera donnée au paramètre sera cette adresse.

L'étape 2 de l'exécution de l'appel de méthode est la création d'une mémoire privée comportant les paramètres `valint` et `valtab`.

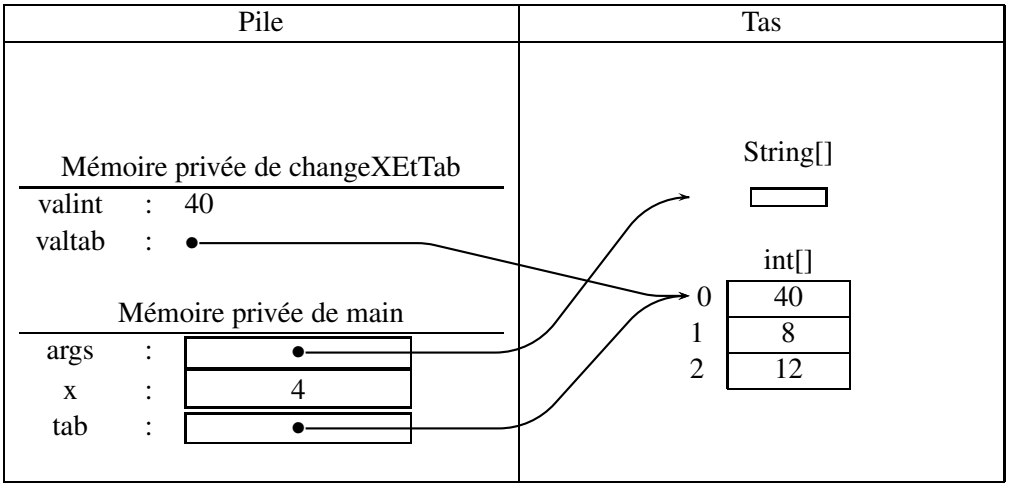


On voit dans cette représentation de la mémoire que passer un tableau en paramètre ne crée pas un nouveau tableau. Ce n'est pas le tableau lui-même qui est recopié dans la mémoire privée de la méthode mais seulement son adresse, l'emplacement de la mémoire privée du tableau dans le tas.

À partir de là, l'étape 3 consiste à exécuter les instructions de la méthode. Ligne 10 : le contenu de `valint` est multiplié par 10, ce qui donne 40. Ce résultat est stocké dans `valint`.



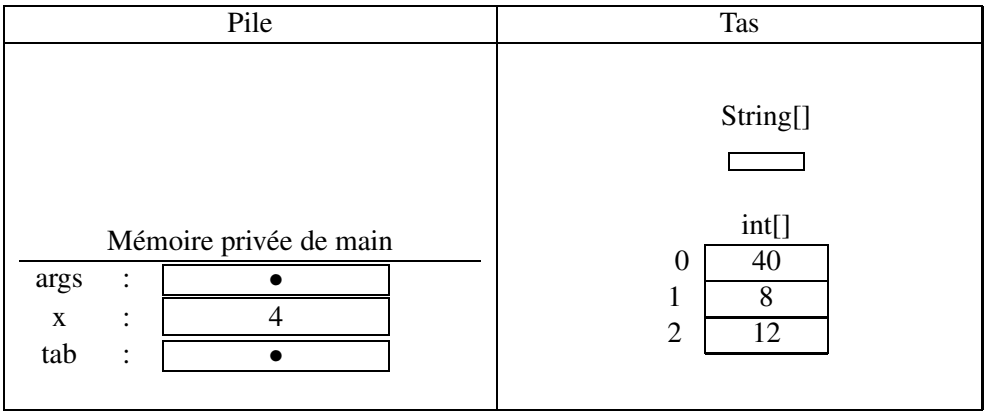
Ligne 11 : le contenu de `valtab[0]` ets multiplié par 10 ce qui donne 40. Le résultat est stocké dans `valtab[0]`.



En changeant `valtab[0]` dans `changeXEtTab` on change aussi `tab[0]` dans `main` car `valtab[0]` et `tab[0]` désigne le même emplacement dans la mémoire, dans le tas.

Dans le cas de `x`, l'affectation change `x` dans l'esapce privé de `changeXEtTab`, dans la pile, ce qui ne peut avoir aucun effet sur les instructions exécutées dans la méthode `main`.

La méthode étant à présent terminée, sa mémoire privée est libérée.



L'exéuction de la ligne 5 est terminée, il faut à présent exécuter la ligne 6. Elle affiche le contenu de la variable x, c'est-à-dire 4. Puis ligne 7, c'est `tab[0]`, c'est-à-dire 40 qu'il faut afficher.

On voit donc que l'exécution de la méthode n'a rien changé à la variable de type `int` mais qu'elle a modifié le tableau passé en paramètre.

10.2.1 Terminologie

On dit qu'un paramètre est passé par copie si la valeur qu'il contient est écrite dans un nouvel emplacement en mémoire. On dit que ce paramètre est passé par adresse ou par référence si la valeur n'est pas copiée mais son adresse en mémoire est transmise au sous-programmme (méthode, fonction ou préocédure appelée). En java, les types primitifs sont transmis par copie et les tableaux et objets (comme les String par exemple) sont transmis par référence.

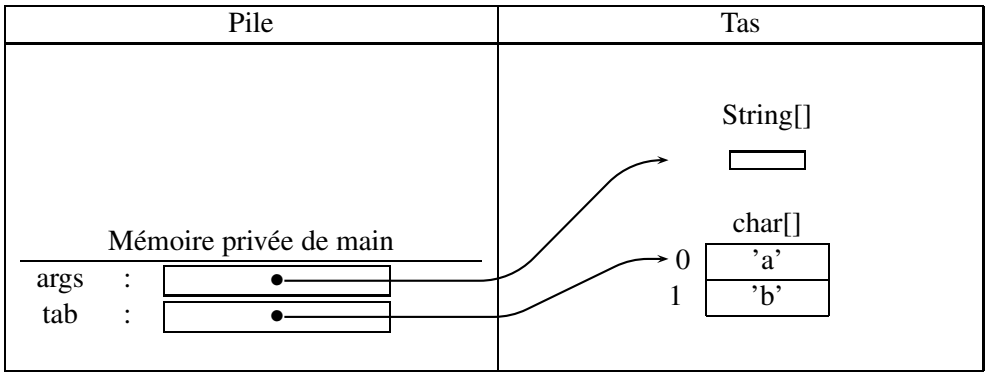
Dans d'autres langages, le programmeur a le choix du mode de passage des paramètres. C'est par exmple le cas en C et en C++.

10.2.2 Agrandissement d'un tableau

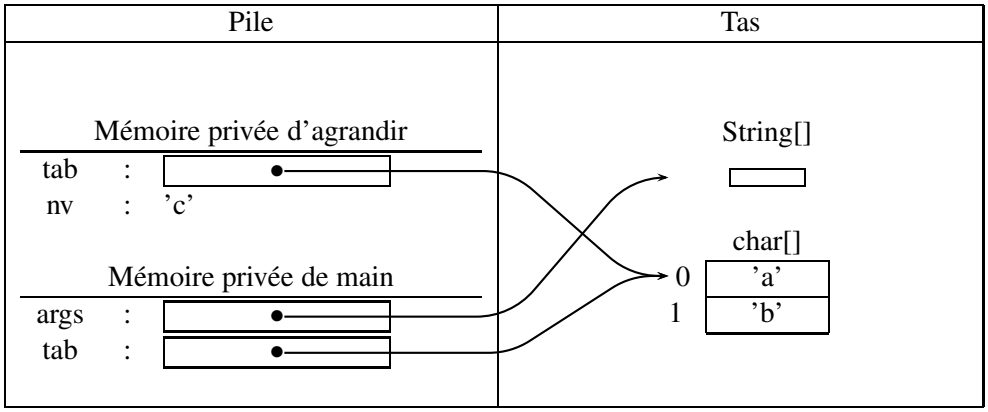
On pourrait avoir l'impression que toute opération faite sur un tableau passé en paramètre dans une méthode change nécessairement le tableau dans la méthode appelante. Or ce n'est pas le cas comme va le montrer l'exemple suivant. On veut écrire une méthode qui va ajouter une valeur dans un tableau et pour ce faire va ajouter une case de plus. Comme il n'est pas possible d'agrandir un tableau existant, la méthode va en fait créer un nouveau tableau plus grand, recopier les valeurs du premier tableau et ajouter la nouvelle valeur dans la case en plus.

```
1  public class Agrandir{
2     public static void agrandir(char[] tab, char nv){
3        char[] res = new char[tab.length+1];
4        for (int i=0; i<tab.length; i++){
5           res[i]=tab[i];
6        }
7        res[tab.length]=nv;
8        tab = res;
9     }
10    public static void main(String[] args){
11       char[] tab = {'a', 'b'};
12       agrandir(tab,'c');
13       for (int i=0; i<tab.length; i++){
14          System.out.print(tab[i]);
15       }
16       System.out.println();
17    }
18 }
```

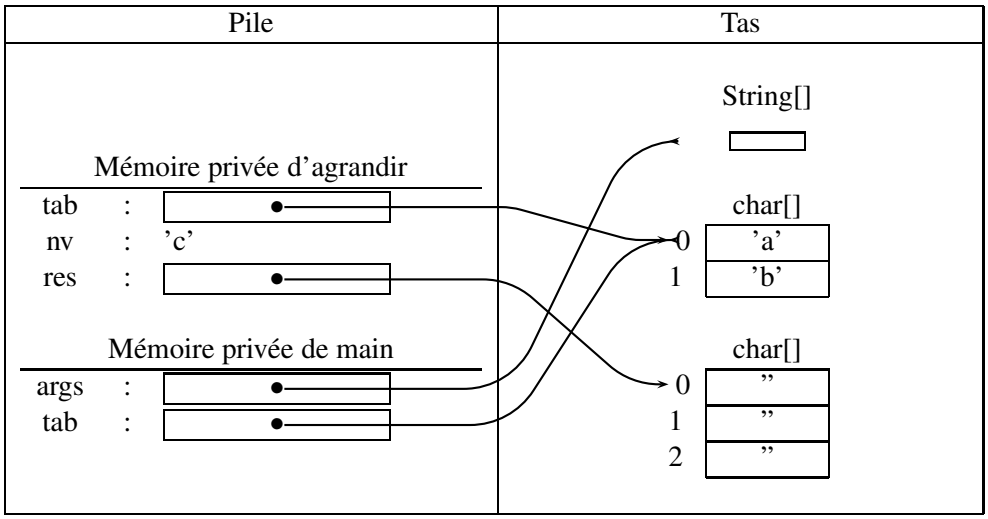
L'exécution commence par la ligne 11, la création d'un tableau à deux cases dans le tas et son affectation à la variable tab déclarée sur cette ligne.



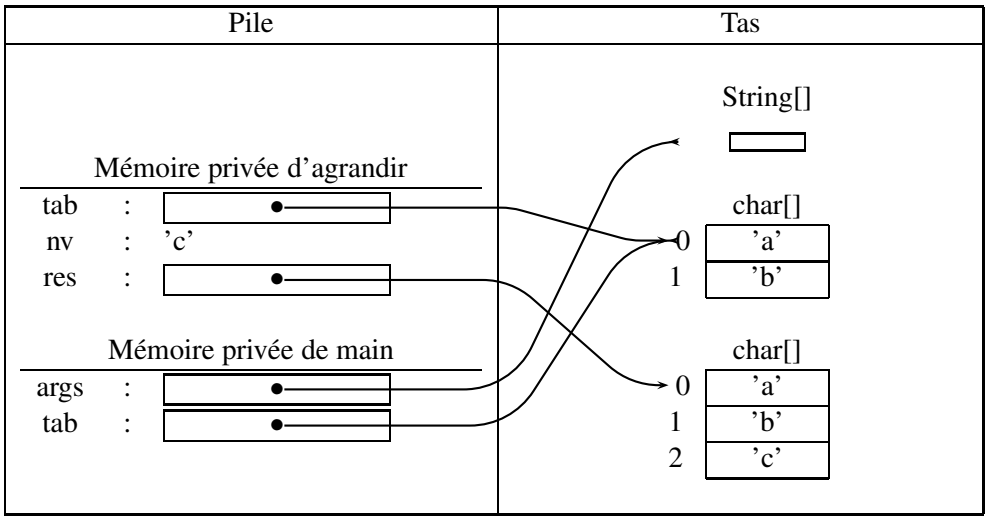
À la ligne 12, appel de méthode. Étape 1 : calcul des valeurs de `tab` et `'c'`. Ce sont respectivement l'adresse du tableau dans le tas (la flèche vers le tableau) et le caractère `'c'`. Étape 2 : création d'une mémoire privée pour `agrandir` avec les paramètres et leurs valeurs respectives.



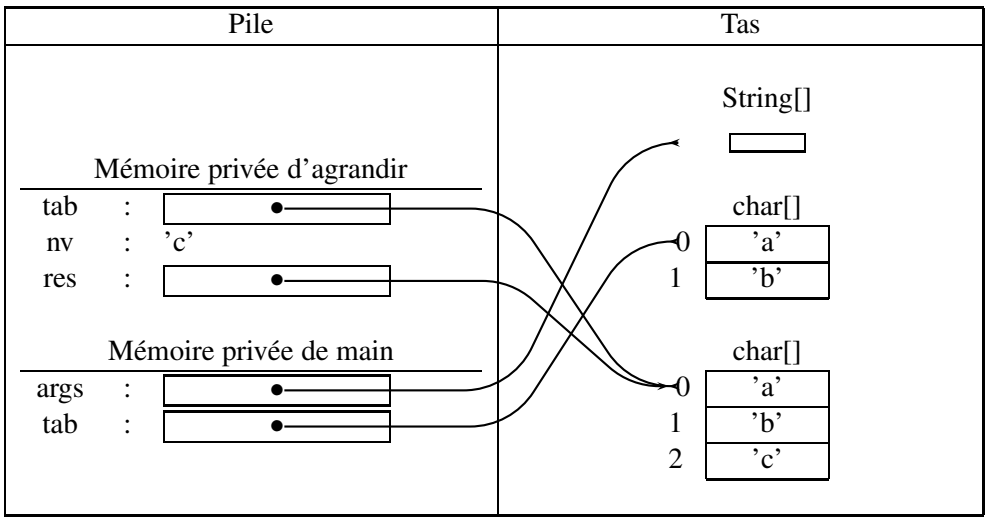
A partir de là, l'étape 3 exécute les instructions du corps d'`agrandir` en commençant par la ligne 3, déclaration de `res`, création d'un tableau à 3 cases et affectation de l'adresse de ce tableau à `res`.



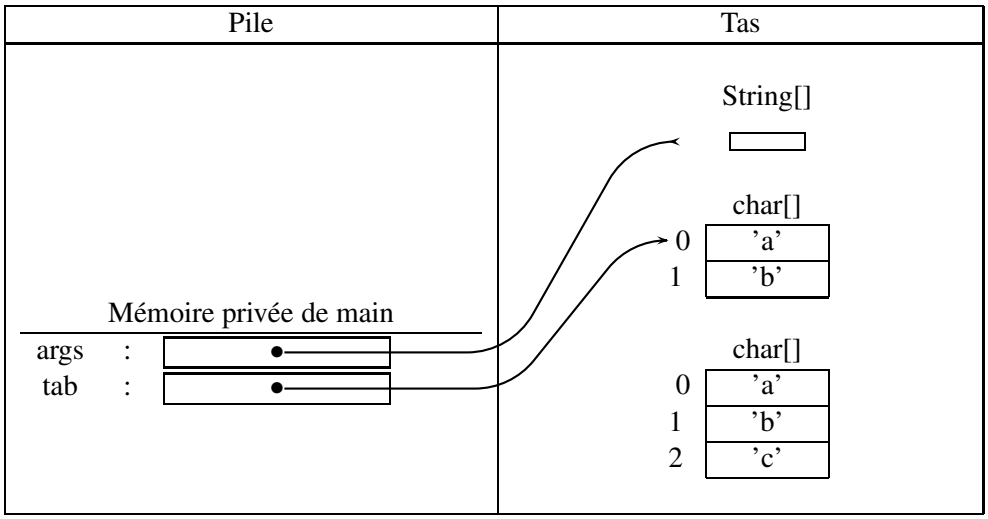
Nous ne détaillerons pas l'exécution de la boucle (lignes 4 à 6), qui va recopier le contenu des deux cases de `tab` dans `res`, ni celle de la ligne 7 qui va recopier le contenu de `nv` dans la dernière case de `res`. A l'issue de ces exécutions, la mémoire est dans l'état suivant.



A partir de là, la ligne 8 est exécutée. Elle consiste à recopier l'adresse de `res` dans `tab`. L'état de la mémoire est le suivant.



La méthode est terminée. Elle ne renvoit pas de résultat car c'est une méthode `void`. Étape 4 de l'exécution : la mémoire privée de `agrandir` est libérée.



On voit que l'appel de la méthode n'a absolument pas changé le tableau `tab` de la méthode `main`, que l'agrandissement n'a pas eu lieu et que le tableau à trois cases n'est pas accessible depuis main faute d'un nom pour désigner son adresse.

Ce comportement est confirmé par l'exécution du programme.

```
> java Agrandir
ab
```

10.2.3 Formulation du phénomène observé

Le fait qu'une affectation ait une portée locale à une méthode ou affecte plusieurs méthodes ne dépend pas fondamentalement du type de la variable mais de l'emplacement mémoire qui est concerné par l'affectation. Si l'affectation change le contenu d'un espace mémoire dans la pile, c'est nécessairement local à une méthode et ne change rien aux variables des autres méthodes. Si l'affectation change le contenu d'un espace mémoire dans le tas (typiquement une case de tableau), cela peut changer la valeurs de variables situées dans d'autres méthodes. Pour savoir si une affectation désigne un espace dans la pile ou dans le tas, il suffit de regarder la partie gauche de l'affectation. Si c'est un nom de variable seul, c'est dans la pile. Si c'est un nom de variable suivi d'une valeur entre crochets, c'est dans le tas. `tab= ...` change la pile, `tab[0]= ...` change le tas.

10.2.4 Comment agrandir un tableau dans une méthode

Comme nous l'avons vu, il n'est pas possible d'agrandir un tableau par modification d'un tableau passé en paramètre. Un nouveau tableau créé dans la procédure ne peut pas être connu dans la méthode appelante (dans notre exemple, le `main`) si ce n'est en renvoyant ce tableau comme résultat de la méthode. Ce résultat peut être affecté à la variable adéquate dans la méthode main.

```
public class Agrandir2{
   public static char[] agrandir(char[] tab, char nv){
      char[] res = new char[tab.length+1];
      for (int i=0; i<tab.length; i++){
         res[i]=tab[i];
      }
```

```
      res[tab.length]=nv;
      return res;
   }
   public static void main(String[] args){
      char[] tab = {'a', 'b'};
      tab = agrandir(tab,'c');
      for (int i=0; i<tab.length; i++){
         System.out.print(tab[i]);
      }
      System.out.println();
   }
}
```

L'exécution du programme :

```
> java Agrandir2
abc
```

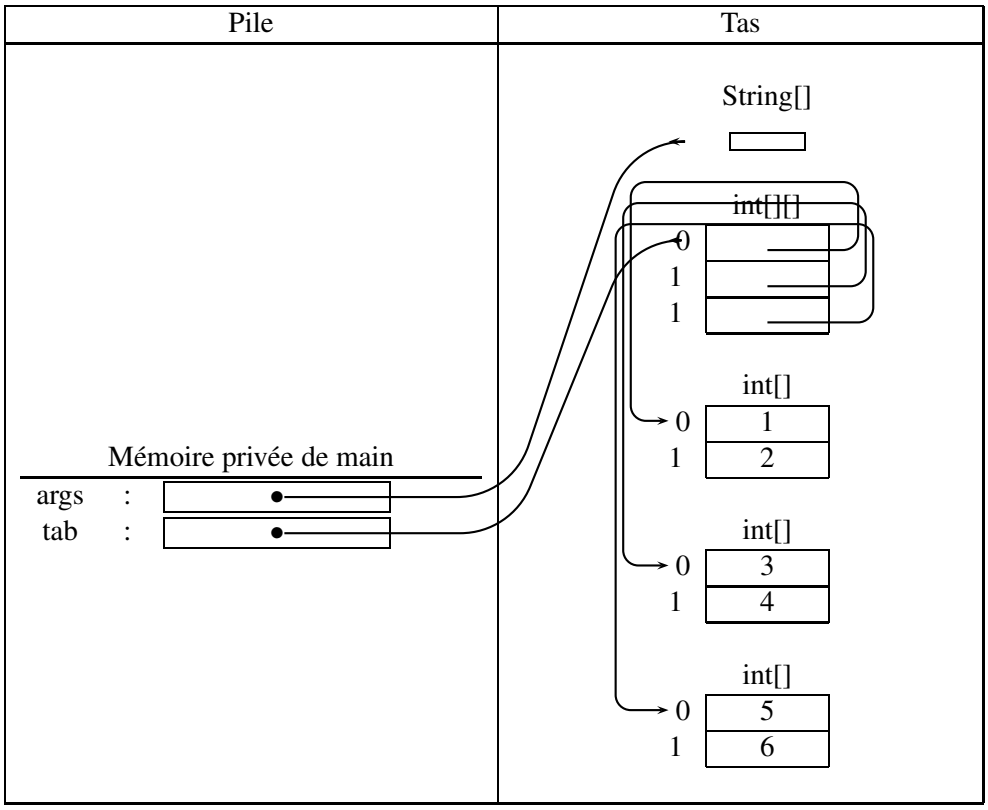
10.3 Tableaux à deux dimensions

Il n'existe pas à proprement parler de tableaux à deux dimensions en Java : ce qu'on appelle tableaux à deux dimensions sont en fait des tableaux à une dimension qui contiennent dans chaque cas un tableau à une dimension.

Si l'on déclare un tableau de la façon suivante :

```
int[][] tab = {{1,2},{3,4},{5,6}};
```

On définit un tableau de trois cases et dans chacune des trois cases il y a un tableau à deux cases. `tab[0]` désigne un tableau à deux cases. On peut accéder à la première case de ce tableau en ajoutant `[0]` après, ce qui nous donne `tab[0][0]`.



Il est possible d'affecter le contenu d'une case du tableau `tab` à une variable de type `int[]` et réciproquement. `tab[0]` désigne un tableau à une dimension.

10.4 Conclusion du chapitre

Les références ou adresses dans le tas jouent un rôle important en Java et font que les tableaux et les String se comportent différemment des valeurs des types primitifs (int, double, boolean, char).

— les tableaux et les String doivent être créés (opération new) avant d'exister. Les int et les boolean n'ont pas à être créés.

— les mémoires privées des tableaux et String sont dans le tas. Les int et boolean n'ont pas de mémoire privée.

— une variable de type int contient directement la valeur, une variable de type tableau contient l'adresse de la mémoire privée du tableau.

— un tableau passé en paramètre peut être modifié par la méthode appelée. Ce n'est pas le cas pour des valeurs primitives passées en paramètres.