Chapitre 3

Les références et la mémoire

3.1 Introduction

En Java, pour déclarer une variable, il faut donner son nom, précédé du *type* qu'on souhaite lui attribuer.
Ces types peuvent être des types primitifs (`int n;`), ou bien correspondre à des classes (`Date d1;`) ou encore à des tableaux (`int[] tab;`). Les variables `n` et `d1` en Java, ne sont pas de la même sorte. La première contient une valeur élémentaire, tandis que la seconde contient une *référence* à un objet de type `Date`. De même, `tab` contient une référence à un tableau. Voilà ce que nous allons étudier en détail dans ce chapitre, en commençant par les variables de type primitif et en expliquant ensuite ce qu'est une variable référence.

3.2 Noms, valeurs et contexte d'exécution

Une méthode comporte des paramètres et des variables qui sont déclarées avec leurs types. Chaque paramètre et chaque nom est associé à un espace mémoire suffisant pour stocker une valeur du type. La liste des associations entre noms et valeurs qui existent dans une fonction donnée, nous l'appellerons *contexte d'exécution* de la méthode. Ce contexte d'exécution, comme son nom l'indique, sert à l'exécution pour stocker les valeurs des variables et paramètres. Une instruction comme `Terminal.ecrireIntln(x+1)` ne peut pas s'exécuter sans connaître la valeur de `x`. Dans un contexte où `x` vaut 10, elle affichera 11. Dans un contexte où `x` vaut autre chose, elle affichera une autre valeur.

On peut se réprésenter un contexte d'exécution comme un tableau avec dans la première colonne le nom et la deuxième colonne, la valeur associée. Lorsqu'une variable est déclarée, une nouvelle ligne est ajoutée dans ce tableau.

Prenons un exemple.

```
static int puissance(int x, int y){
    if (y<0)
        throw new Error();
    int res=1;
    for (int i=1; i<=y; i++){
         res=res*x;
    }
    return res;
```

```
}
```

Cette méthode doit être exécutée lorsque dans le corps d'une autre méthode, par exemple la méthode main, on trouve un appel tel que : `n=puissance(3,2);`. Pour exécuter cette instruction, il faut exécuter la méthode `puissance` avec pour valeurs des paramètres 3 et 2. Cette exécution commence avec la création d'un contexte comportant les noms des deux paramètres.

contexte de puissance	
x	3
y	2

A la ligne 2 de la méthode, il y a un test `(y<0)`. Pour exécuter ce test, il faut connaître la valeur de y. Cette valeur, on va la chercher dans le contexte d'exécution. Associé au nom y, il y a la valeur 2. Le test `(2<0)` a pour résultat `false`.

A la ligne 4 de la méthode, il y a déclaration d'une variable `res` et donc création d'un nouveau nom utilisable dans la méthode. Cela se traduit par l'ajout d'une nouvelle association (nom, valeur) dans le contexte. Comme la variable est initialisée à 1, c'est cette valeur qui est associée au nom `res`.

contexte de puissance	
x	3
y	2
res	1

L'exécution de la boucle comporte la déclaration de la variable `i` initialisée à 1. Elle est ajoutée au contexte.

contexte de puissance	
x	3
y	2
res	1
i	1

Au fil de l'exécution de la méthode, `i` prendra d'autres valeurs : 2, puis 3 et `res` prendra les valeurs 3 puis 9. Ainsi, les valeurs stockées dans les variables varient au fil du temps.

Le Contexte évolue de la façon suivante :

- A sa création, le contexte contient les noms des paramètres initialisés avec les valeurs données dans l'appel.
- Chaque déclaration de variable ajoute une ligne dans le contexte.
- Chaque affectation à une variable change la valeur associée au nom de cette variable.

3.3 Organisation de la mémoire

Lors du démarrage d'un programme, une partie de la mémoire de l'ordinateur lui est attribuée par le système d'exploitation. Chaque programme qui s'exécute a son propre espace mémoire où il peut écrire des informations et relire ce qu'il a écrit. Dans le cas d'un programme Java, l'espace mémoire est divisé en deux partie distinctes : la *pile* (*stack* en anglais) qui contient les contextes d'exécution des méthodes et le *tas* (*heap* en anglais) qui contient les objets et les tableaux.

Nous reviendrons sur le fonctionnement de la pile plus tard. Pour l'instant, nous y mettrons le contexte d'exécution de la méthode `main`.

Les objets et les tableaux sont créés dans le tas lors de l'exécution d'une instruction `new`. Dans la réalité, les éléments ne sont pas forcément rangé dans un ordre déterminé dans le tas. Mais pour simplifier la représentation, nous supposerons que les objets seront rangés dans l'ordre de création. Selon le nombre de cases du tableau, le nombre de variables d'un objet et selon leurs types, il faut

plus ou moins de mémoire. Par exemple, pour un int, il faut 4 octets soit 32 bits. Pour un tableau de 5 int, il faut 5 fois 4 octets, 20 octets. Autre exemple, si un objet contient une variable de type int et une autre de type char (2 octets), il faut en tout 6 octets pour représenter l'objet.

L'instruction `new` calcule la place nécessaire, la réserve pour l'objet créé, initialise les valeurs en exécutant le constructeur de la classe ou en mettant la valeur par défaut du type dans toutes les cases du tableau. Chaque objet ou tableau créé dans le tas est identifié par son adresse mémoire qui est le numéro de la première case mémoire utilisée pour stocker le tableau ou l'objet.

A partir de maintenant, on représentera les adresses par une notation commençant par `adr` suivi d'un nombre à trois chiffres (par exemple `adr017`).

L'instruction `new` commence par calculer la taille de la chose à créer, attribue un emplacement mémoire de cette taille de sorte que toutes les cases d'un tableau ou toutes les variables d'un objet soient stockés dans des cases mémoires contingües. Puis l'espace réservé est initialisé. Enfin, l'instruction se termine en renvoyant l'adresse de la chose créée.

Par exemple `new int[5]` réserve un espace de 20 octets pour le tableau, initialise chaque case avec 0, la valeur par défaut du type `int` et renvoie l'adresse du tableau créé, par exemple `adr001`.

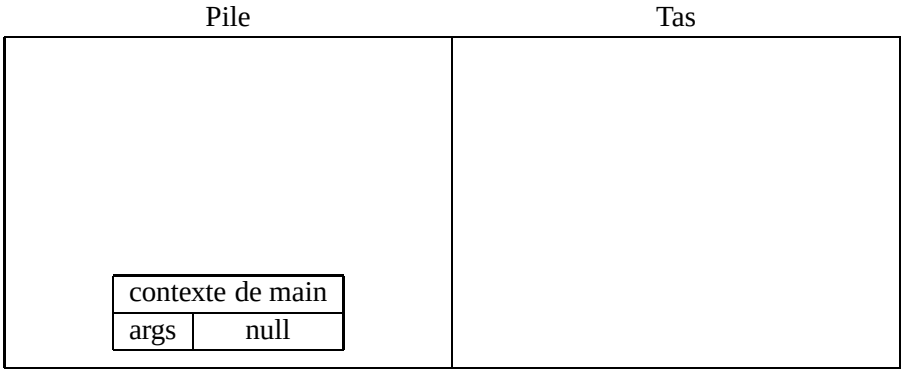
Si l'on fait une affectation, par exemple `t=new int[5];` c'est l'adresse renvoyée par l'instruction `new` qui est stockée dans `t` ou pour être plus précis, qui est associée au nom `t` dans le contexte d'exécution de la méthode qui contient cette affectation.

3.4 Exemple de programme avec new

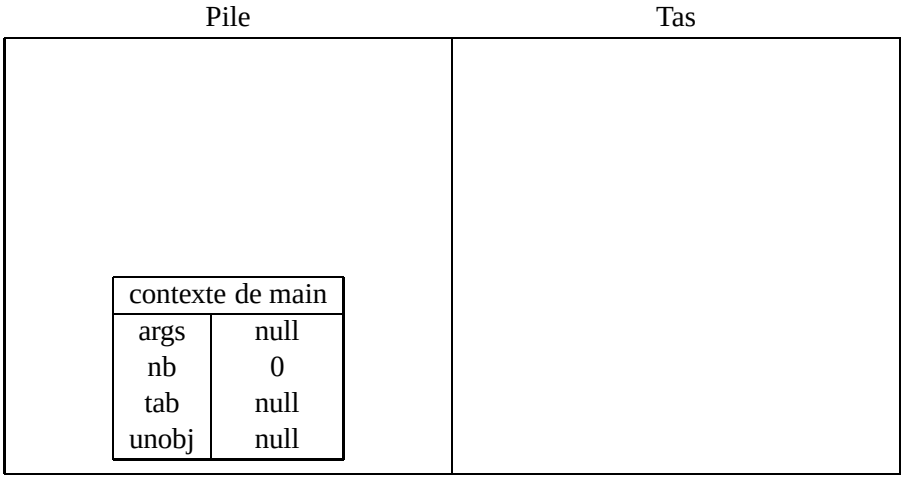
```
class Objet{
    public int x;
    Objet(int y){
        x = y;
    }
}
public class ExempleNew{
    public static void main(String[] args){
        int nb=0;
        int[] tab;
        Objet unobj;
        tab = new int[3];
        unobj = new Objet(3);
        nb = 17;
        tab[1]=12;
        unobj.x = 7;
    }
}
```

Lorsque le programme commence, le contexte de la méthode `main` est créée avec le nom du paramètre et sa valeur. Dans ces notes de cours, nous allons supposer que cette valeur est `null`[1]. Le Contexte de `main` est dans la pile et il n'y a encore rien dans le tas.

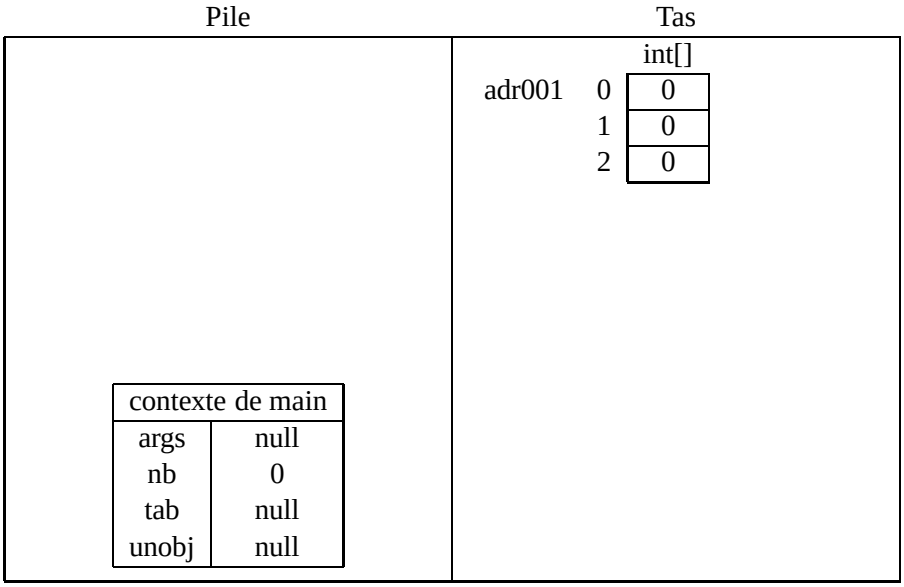
1. Cela n'est pas vrai, mais cela simplifie la présentation



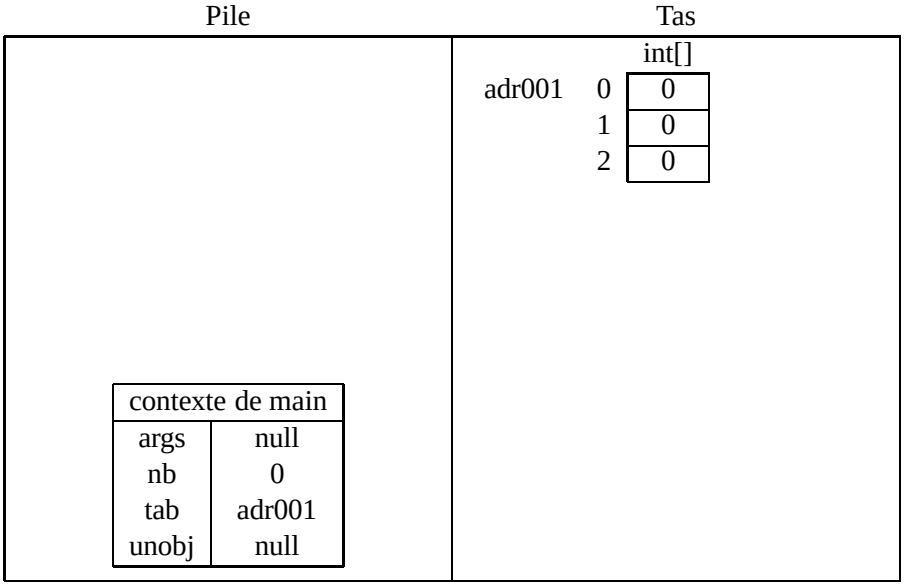
Après les lignes 9, 10 et 11, trois nouvelles variables sont créées dans la contexte de `main`. Mais à ce stade, il n'y a pas encore de tableau ni d'objet : aucun tableau ni aucun objet n'a été créé tant qu'il n'y a pas eu de new.



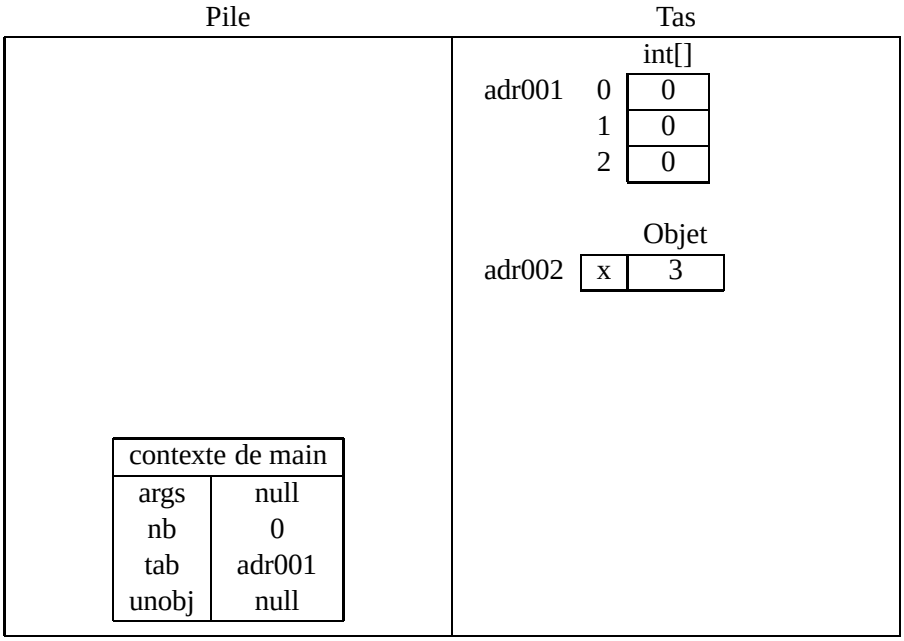
La ligne 12 est une affectation avec à droite, une instruction new qui crée un nouveau tableau de taille 3. L'instruction new réserve une zone de trois case de 4 octets pour le tableau à une adresse que l'on appellera ici `adr001`. A l'issue du new, et avant que l'affectation de la ligne 12 ne soit effectuée, la mémoire est dans l'état suivant.



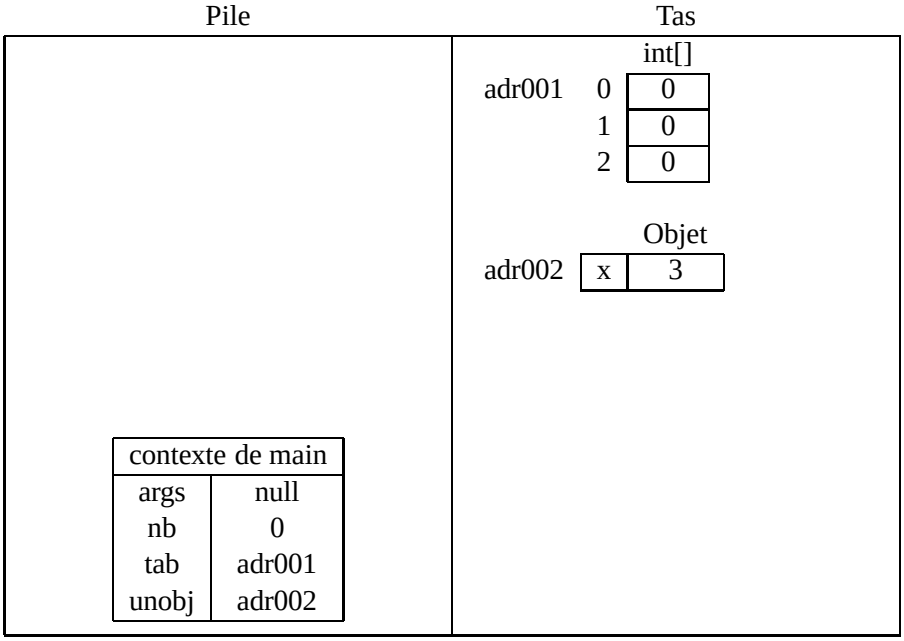
Le new renvoie l'adresse du tableau dans le tas, `adr001`. L'affectation va recopier cette adresse dans le contexte de `main`, associée au nom `tab`.



La ligne 13 comprend également un new et une affectation. L'exécution commence par le `new` qui crée un objet dans le tas à l'adresse `adr002` et l'initialise grâce au constructeur qui prend la valeur 3 en paramètre et la met dans la variable d'instance `x`. L'objet consiste en un contexte qui associe à chacune des variables d'instance une valeur. Après le new et avant l'affectation, la mémoire a l'état suivant.

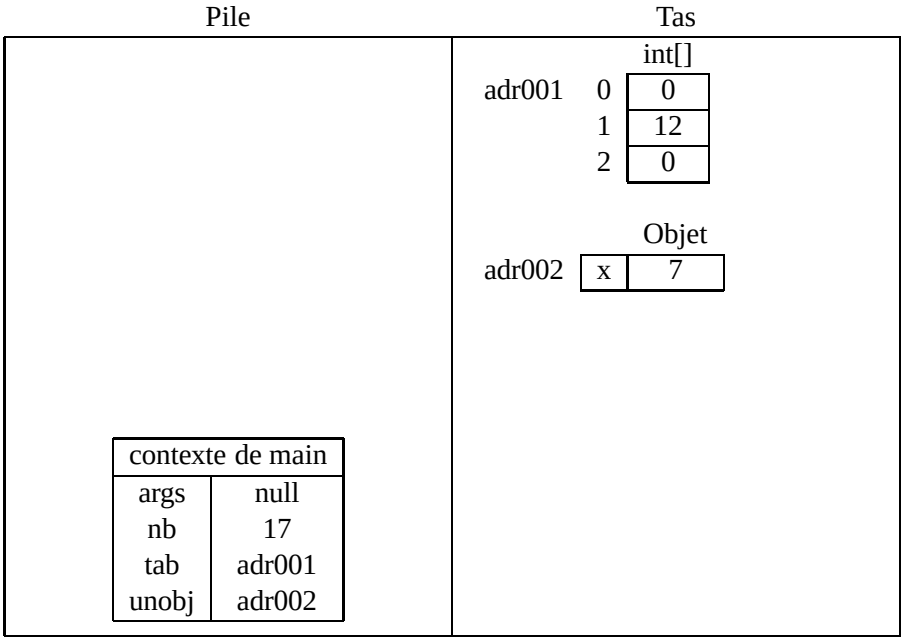


Le new renvoie l'adresse de l'objet créé, `adr002`. L'affectation associe le nom `unobj` à cette adresse dans le contexte de `main`.



Chaque case d'un tableau et chaque case de la colonne de droite d'un environnement peut être désignée au moyen d'un *chemin d'accès*. Ce chemin peut être utilisé pour changer le contenu de la case (à gauche d'une affectation) ou pour utiliser la valeur stockée dans la case pour un calcul.

Trois exemples de chemin sont utilisés dans autant d'affectations à la fin du programme, lignes 14 à 16. Ces chemins sont `nb`, qui désigne la case contenant la valeur 17 dans le contexte de main, `tab[1]` qui désigne la deuxième case du tableau et `unobj.x` qui désigne la case de l'objet contenant la valeur 3. Le contenu des trois cases correspondantes est modifié par les trois affectation, si bien qu'à la fin de l'exécution du programme, l'état de la mémoire est le suivant.



3.5 Utilisation d'une variable à droite d'une affectation

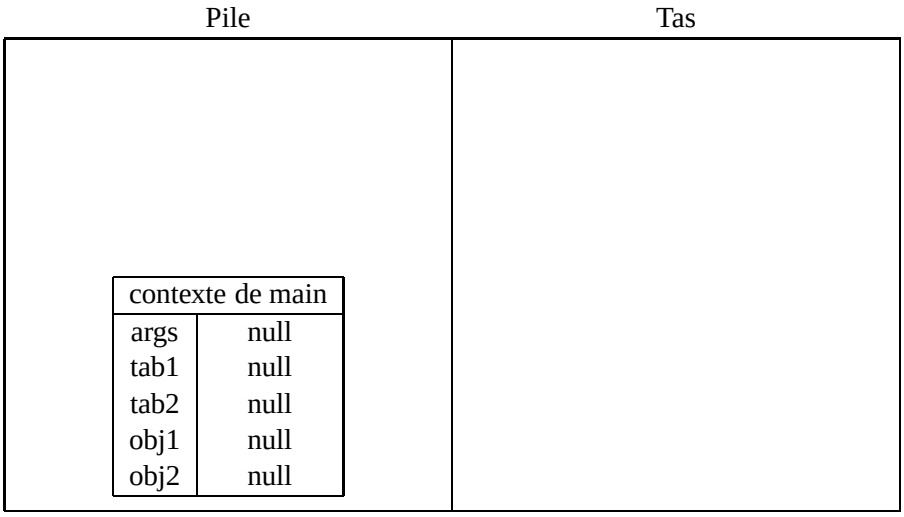
Voyons le cas d'un petit programme qui utilise la valeur d'une variable en partie droite d'une affectation.

```
class Objet{
    public int x;
    Objet(int y){
        x = y;
    }
}
public class ExempleVarD{
    public static void main(String[] args){
        int[] tab1, tab2;
        Objet obj1, obj2;
        obj1 = new Objet(3);
        obj2 = obj1;
        obj2.x = 17;
        Terminal.ecrireIntln(obj1.x);
        tab1 = new int[3];
        tab2 = tab1;
        tab2[1]=12;
        Terminal.ecrireIntln(tab1[1]);
    }
}
```

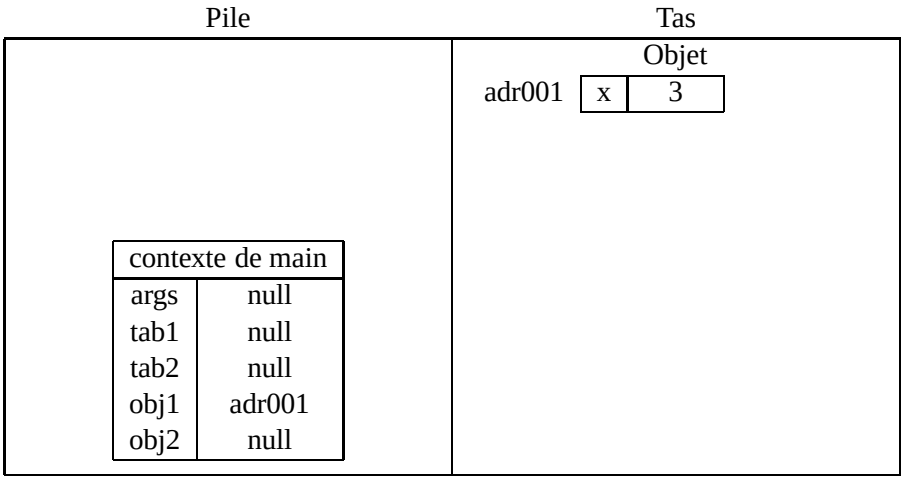
Avant même de détailler l'exécution du programme, à sa simple lecture, on sait déjà qu'il y aura deux choses dans le tas parce qu'il y a deux instructions `new` dans ce programme. Il y aura un objet créé ligne 11 et un tableau créé ligne 15.

L'exécution du programme commence avec la création du contexte de `main` dans la pile avec le nom du paramètre `args`. Les deux premières lignes de la méthodes (lignes 9 et 10) sont des déclara-

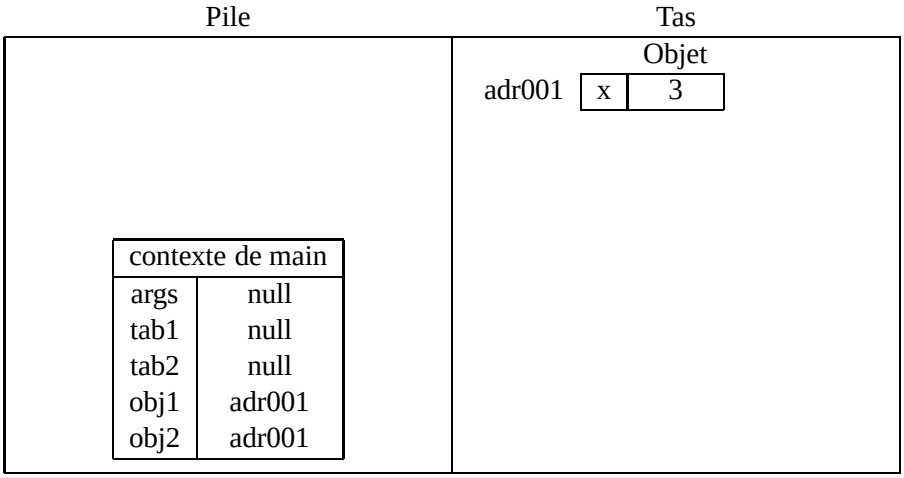
tions de variables qui ajoutent de nouveaux noms dans le contexte de `main`. A la fin de l'exécution de la ligne 10 on a l'état mémoire suivant.



L'exécution de la ligne 12 est un appel à l'instruction `new` suivie d'une affectation. Le new crée un nouvel objet et renvoie son adresse, l'affectation associe le nom `obj1` à cette adresse dans le contexte de `main`. L'état est à présent le suivant.

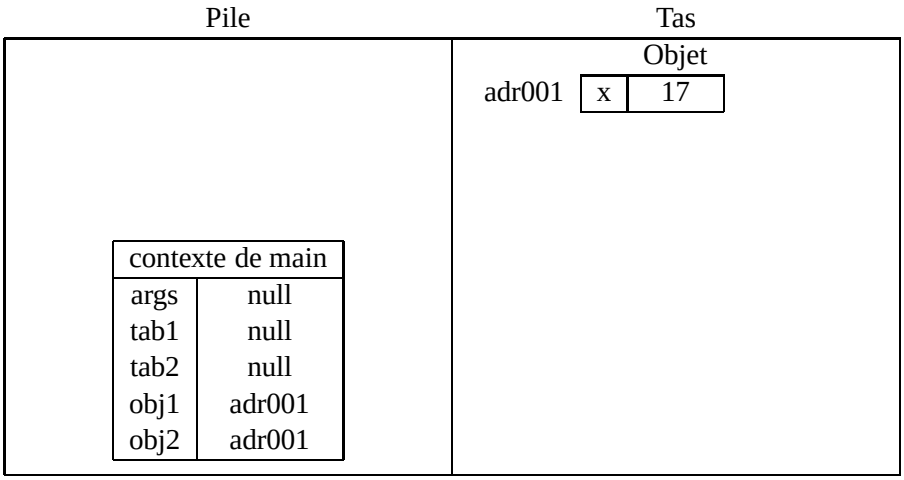


La ligne 12 consiste en une affectation du contenu de la variable `obj1` dans la variable `obj2`. Comme toujours, la partie droite est exécutée la première. La variable de la valeur `obj1` est recherchée dans le contexte de `main`. Cette valeur est l'adresse `adr001`. Cette adresse est la nouvelle valeur donnée à `obj2` dans le contexte de `main`. Le nouvel état de la mémoire après l'affectation est le suivant.



L'effet de l'affectation est qu'il y a maintenant deux noms différents pour le seul objet qui existe. `obj1` et `obj2` sont deux noms qui désignent l'objet stockée à l'adress `adr001`.

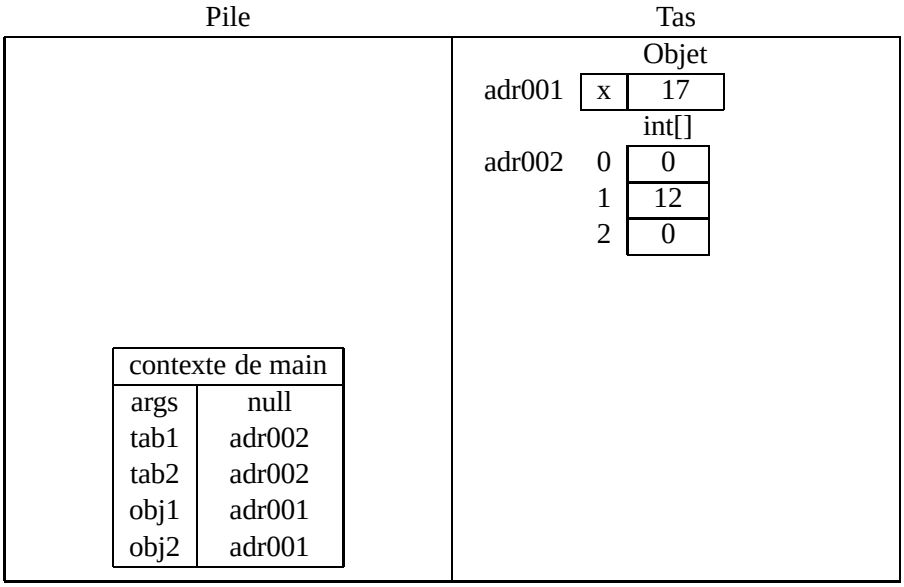
La ligne 13 est une affectation à la variable `x` de l'objet contenu dans `obj2`.



Ligne 14, lorsqu'on demande l'affichage de `obj1.x`, on va chercher dans le contexte de l'objet de l'adresse `adr001` la valeur de `x` pour l'afficher. On trouve la valeur 17 et c'est elle qui est affichée. Affecter une valeur à `obj2.x` à la ligne 13 a également changé la valeur de `obj1.x`. On a deux chemins différents, `obj1.x` et `obj2.x`, qui aboutissent à la même case du tas.

On a le même phénomène aux lignes 15 à 18 avec un tableau : un seul tableau est créé ligne 15. Ce tableau est associé aux deux noms `tab1` et `tab2`. Changer le contenu de la case 1 de `tab2` (ligne 17) change aussi la case 1 de `tab1` puisque c'est le même tableau. On a ici encore deux chemins différents qui aboutissent à la même case.

L'état mémoire à la fin du programme est le suivant.

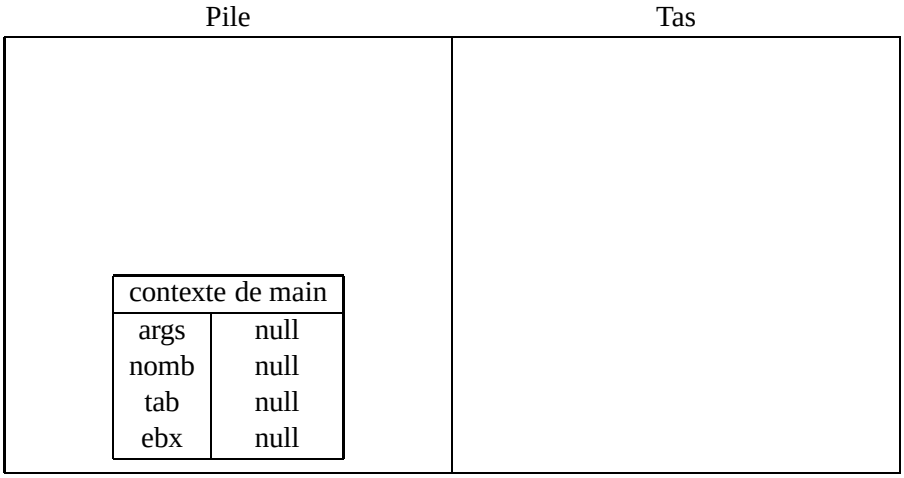


3.6 Tableau d'objets et objet dans un objet

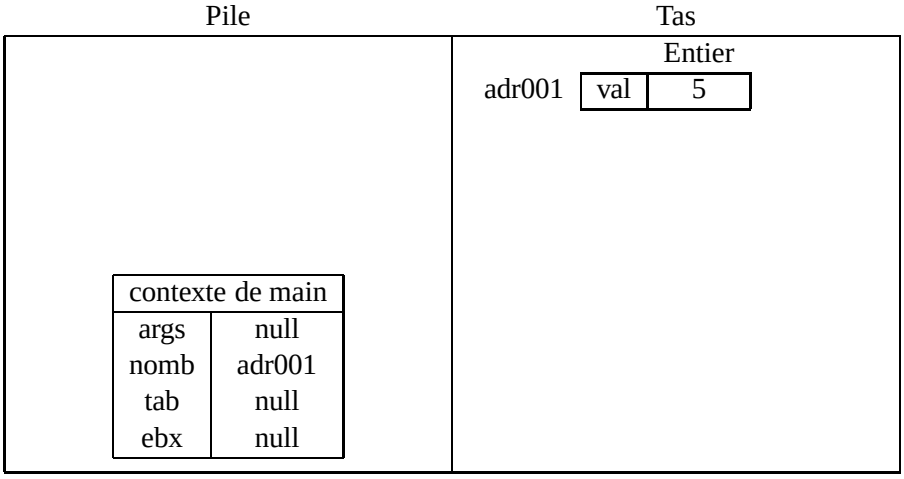
Les cases d'un tableau et les contextes d'un objet peuvent contenir des adresses aussi bien que des entiers ou des booléens.

```
class Entier{
    public int val;
    Entier(int x){
        val = x;
    }
}
class EntBool{
    Entier ent;
    boolean bool;
    EntBool(Entier e, boolean b){
        ent = e;
        bool = b;
    }
}
public class ExempleRefs{
    public static void main(String[] args){
        Entier nomb;
        Entier[] tab;
        EntBool ebx;
        nomb = new Entier(5);
        tab = new Entier[2];
        tab[0] = nomb;
        tab[1] = new Entier(2);
        ebx = new EntBool(nomb,true);
    }
}
```

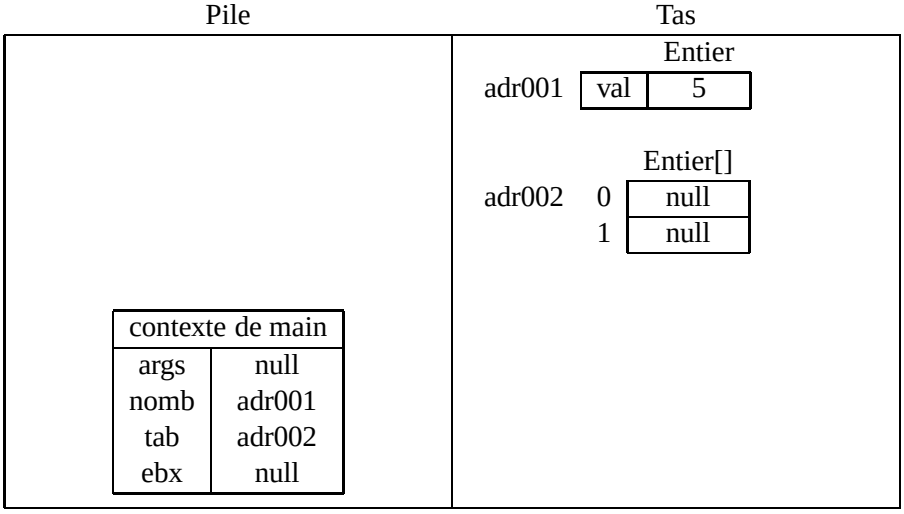
L'exécution du main commence avec la création d'un contexte et les lignes 17 à 19 ajoutent de nouveaux noms à cet contexte.



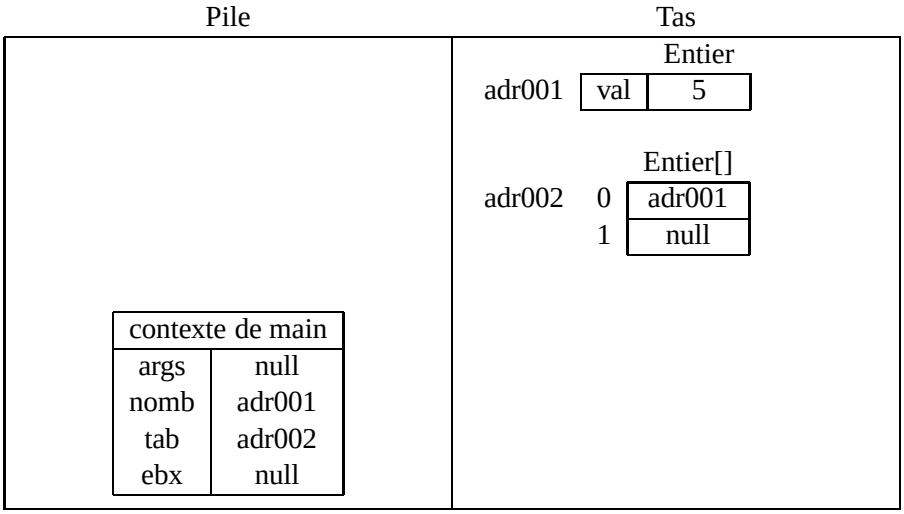
La ligne 20 voit la création d'un objet de type `Entier`, puis son adresse est stockée comme valeur de `nomb` dans le contexte de `main`.



La ligne 21 voit la création d'un tableau. La différence avec la ligne précédente réside dans les crochets : `Entier` suivi de crochets désigne un tableau alors que suivi de parenthèses (ligne 20), cela crée un objet. Le tableau est créé avec comme valeur `null` dans toutes les cases.

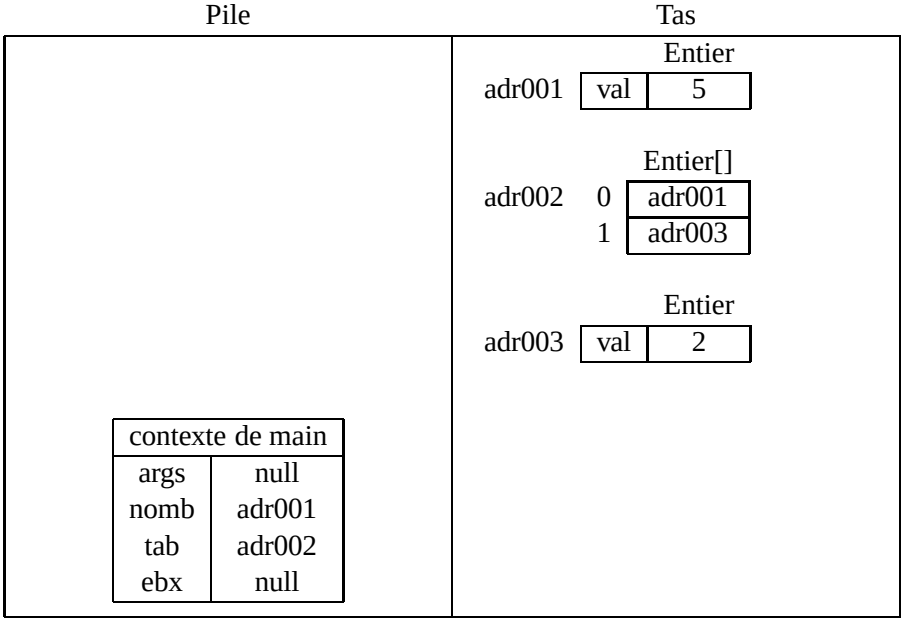


L'affectation de la ligne 22 va changer la valeur de la case 0 du tableau de nom tab, pour y mettre la valeur associée au nom `nomb`. Cette valeur est l'adresse `adr001`. C'est elle qui va être inscrite dans la case 0 du tableau situé à l'adresse `adr002`



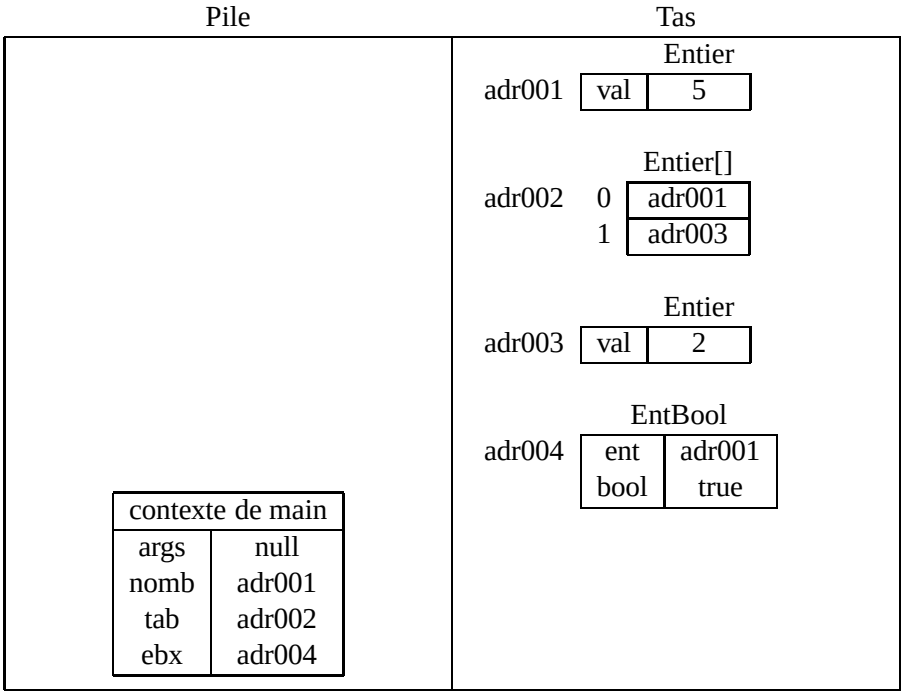
Cette affectation n'a pas créé de nouvel objet. Une affectation ne crée jamais ni objet ni tableau. Ce sont des appels à `new` qui créent des objets et des tableaux.

Ligne 23, il y a création d'un nouvel objet et son adresse est inscrite non pas dans le contexte de `main` mais dans la deuxième case du tableau `tab`.



On a maintenant deux instances de la classe `Entier`, deux objets différents qui ont chacun leur variable `val` propre avec une valeur différente. Le premier objet a deux noms ou plus précisément, deux chemins d'accès : `nomb` et `tab[0]`. Le deuxième n'a qu'un chemin d'accès : `tab[1]`.

La dernière ligne du programme contient la création d'un objet ayant deux variables : une variable de type `boolean` qui contiendra directement une valeur (`true` ou `false`) et une variable de type `Entier` qui contiendra l'adresse d'un objet de type `Entier`.



3.7 La notion de référence

Une référence est une adresse mémoire désignant un objet ou un tableau dans le tas. On appelle type référence en java les types des objets et des tableaux, alors que les types primitifs sont un petit nombre de types prédéfinis tels qu'`int`, `boolean`, `char` et `double`. Les valeurs des types primitifs existent sans qu'il n'y ait besoin de les créer. Il n'est pas nécessaire de créer 17 ou `true` avant de les utiliser. Il faut créer les tableaux et les objets, sans quoi ils n'existent pas. Une valeur d'un type référence est une référence, une adresse.

Pour ce qui est de l'évolution de la mémoire, voici les opérations qui ont une incidence :

- l'appel d'une méthode crée un nouveau contexte d'exécution dans la pile. Nous reviendrons là-dessus dans les sections suivantes.
- une déclaration de variable ajoute une ligne au contexte de la méthode qui la contient. Si c'est un objet qui est créé, un nouveau contexte d'exécution est créé dans le tas.
- un `new` crée une nouvelle chose dans le tas. Cette chose peut être un objet ou un tableau.
- une affectation va modifier le contenu d'un contexte ou d'un tableau. Si la chose à gauche de l'affectation se termine par un crochet fermant (par exemple `tab[1]=3;`), c'est une case de tableau qui est modifiée et cette case est nécessairement dans le tas. Si la chose à gauche de l'affectation ne se termine pas par un crochet fermant, c'est une case d'un contexte qui est modifiée. S'il y a un point à gauche de l'affectation (par exemple `obj.x=3;`), cet contexte est dans le tas (dans un objet). S'il n'y a pas de point (par exemple `x=3;`), cet contexte est celui de la méthode en cours d'exécution.

Il faut bien distinguer entre les *nom de variables* d'un type objet et les *instances d'une classes*, les objets proprement dits. Il peut y avoir des noms sans qu'il n'y ait d'objets (les variables contiennent la valeur `null`) et il peut y avoir des objets sans nom de variable, par exemple dans le cas où les objets sont dans un tableau. Le nombre de noms et le nombre d'objets ne sont pas nécessairement égaux (ni nécessairement différents d'ailleurs).

3.8 Appels de méthodes

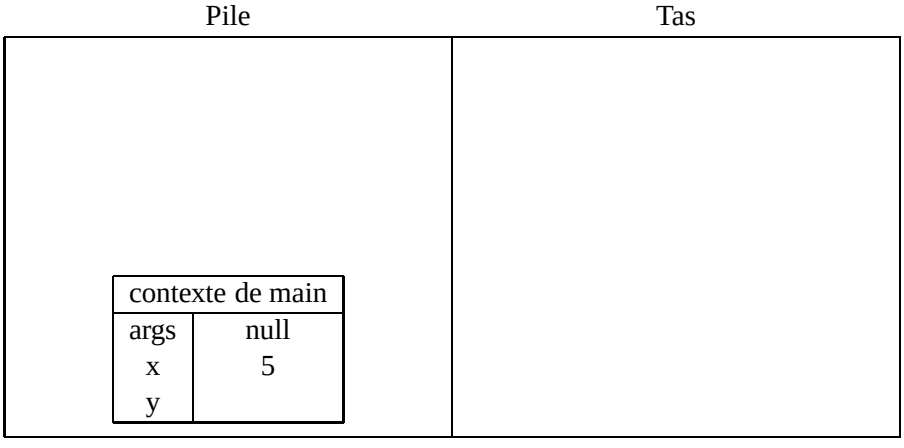
A chaque appel de méthode, un nouveau contexte d'exécution est créé en sommet de pile. A chaque moment, seul le contexte situé le plus haut dans la pile est utilisé. On dira que c'est *le contexte actif*. C'est celui de la méthode qui est en cours d'exécution. En-dessous de lui, dans la pile, il y a les contextes des méthodes qui sont en cours d'exécution et qui attendent le résultat d'un appel de méthode pour se poursuivre.

Lorsque l'exécution d'une méthode est terminée, son contexte est effacé du sommet de la pile. Ce contexte, c'est le contexte actif. Lorsqu'il est effacé, c'est le contexte situé juste en dessous qui redevient actif.

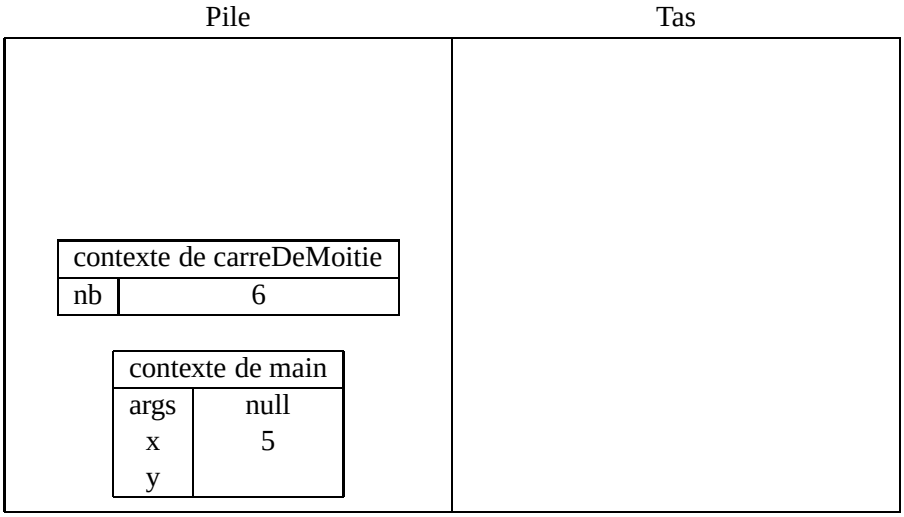
Voyons un exemple.

```
public class ExempleMet{
    static int carre(int x){
        int res = x *x;
        return res;
    }
    static int carreDeMoitie(int nb){
        int res;
        res = carre(nb/2);
        return res;
    }
    public static void main(String[] args){
        int x, y;
        x = 5;
        y = carreDeMoitie(x+1);
        Terminal.ecrireIntln(y);
    }
}
```

L'exécution de la méthode `main` commence avec la création d'un contexte pour main comportant son paramètre `args`. La première ligne du main (ligne 12 du listing) ajoute les deux noms `x` et `y` au contexte courant. On ne sait pas bien quelle valeur il y a dans ces variables avant qu'on ne leur ait affecté une valeur. On représente cela ici par un point d'interrogation. De toute façon, ça ne peut pas être `null` parce qu'en java, `null` n'est pa sune valeur de type `int`. Cette valeur `null` n'est utilisée que pour l'initialisation des variables de types références (objets et tableaux). La ligne 13 affecte la valeur 5 à la variable `x`. A la fin de l'exécution de la ligne 13, l'état de la mémoire est le suivant.

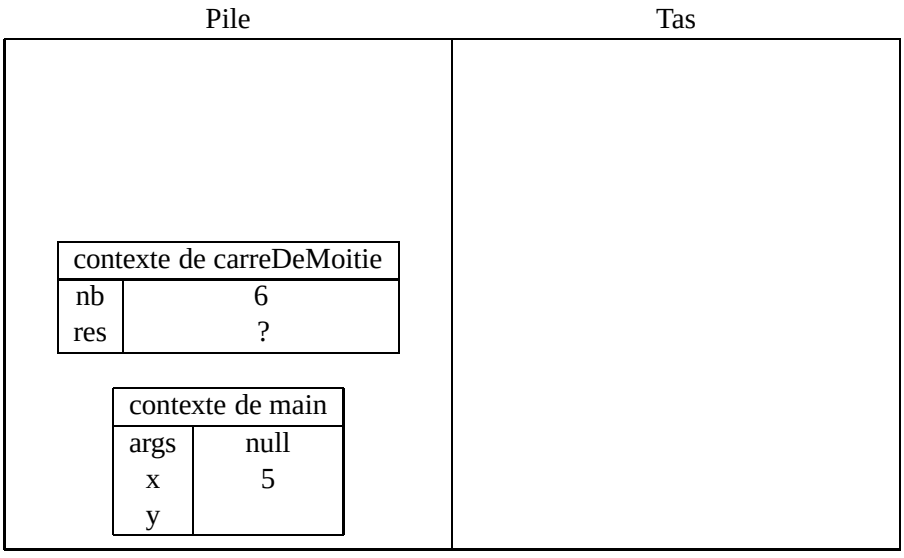


Vient ensuite l'exécution de la ligne 14. Pour toute affectation, il y a d'abord calcul de la valeur de la partie droite (ici : `carreDeMoitie(x+1)`), puis inscription de la valeur en question dans la variable désignée par la partie gauche (ici `y`). L'appel de la méthode commence par le calcul de la valeur des paramètres. Ici, il s'agit de calculer la valeur de `x+1` dans le contexte de `main`. La valeur de `x` dans cet contexte est 5, donc `x+1` vaut 6. La méthode `carreDeMoitie` est appelée avec 6 comme valeur de son paramètre. La méthode `main` n'est pas terminée, mais elle ne peut pas continuer avant d'avoir le résultat de la méthode `carreDeMoitie`. L'exécution de `carreDeMoitie` commence avec la création d'un contexte contenant son paramètre `nb` auquel la valeur 6 est associée.

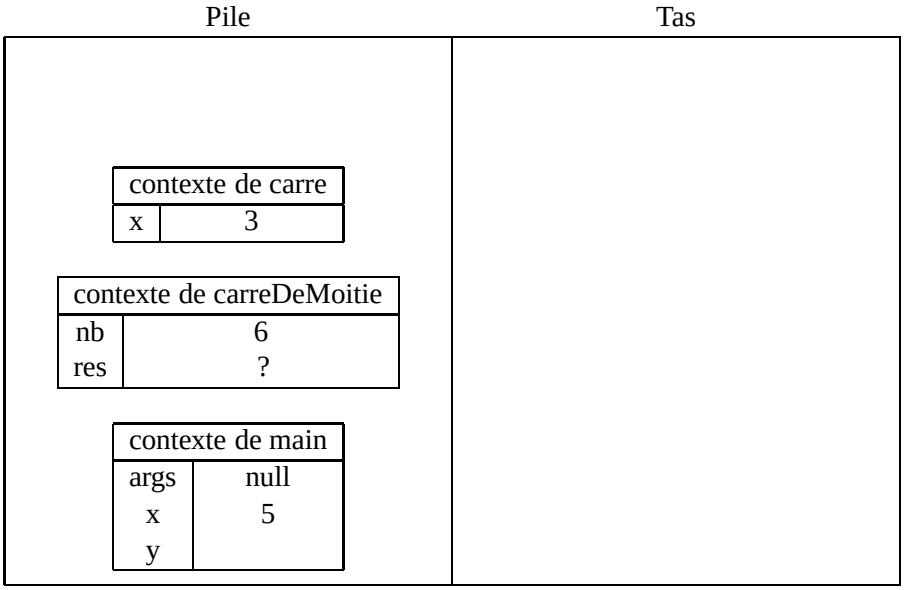


A l'appel d'une méthode, il y a comme une affectation `nb=x+1` mais `nb` appartient au contexte de la méthode appelée alors que `x+1` est évalué dans le contexte de la méthode appelante.

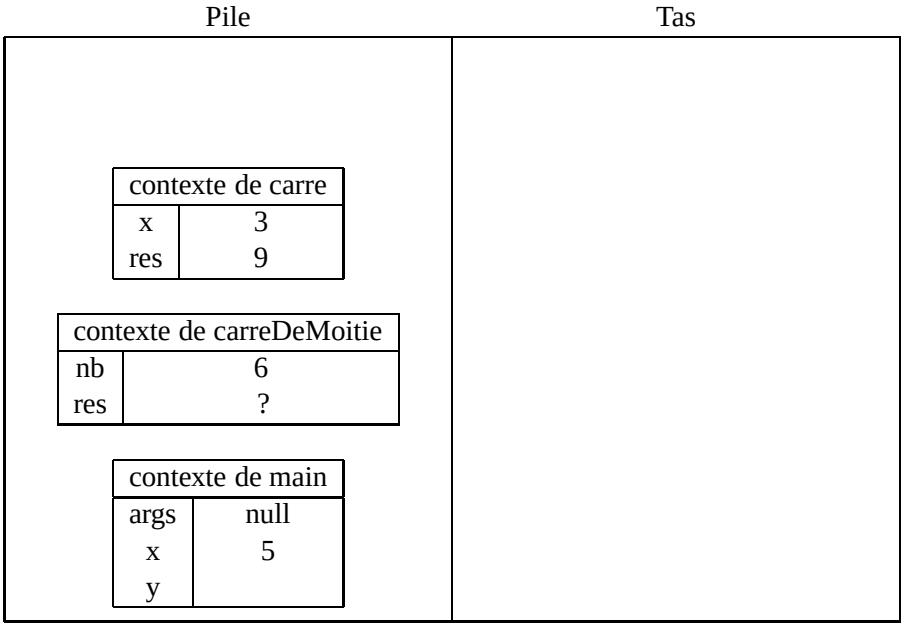
La ligne 7 ajoute une ligne au contexte courant pour le nom `res`.



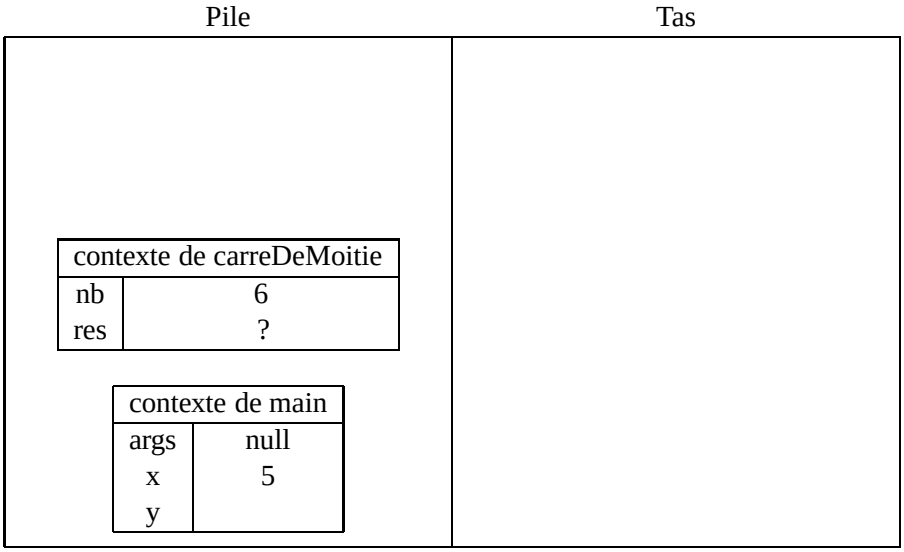
Ligne 8, il y a à peu près la même chose qu'à la ligne 14 : une affectation avec appel de méthode. C'est maintenant au tour de la méthode `carreDeMoitie` d'être en attente du résultat de `Carre`. La valeur du paramètre `nb/2` est évaluée dans le contexte de `carreDeMoitie` et inscrite comme valeur du paramètre `x` dans le nouvel contexte créé pour `Carre`. Cette valeur est 6/2=3.



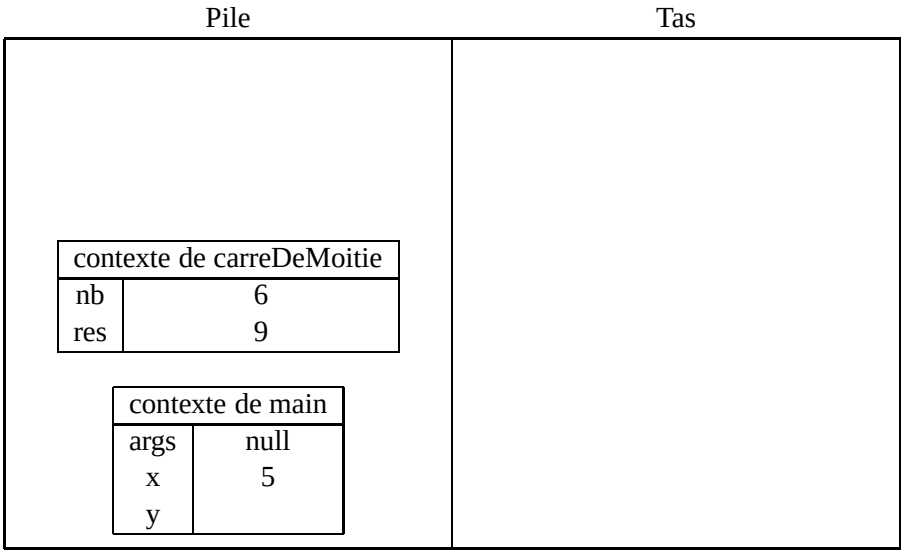
Notez que chaque variable a son propre contexte et si le même nom est utilisé par deux méthodes (ici le nom `x` est utilisé dans `main` et `carre`), cela correspond à deux emplacements différents de la mémoire et leurs valeurs peuvent être différentes (ici, 5 et 3). La ligne 3 déclare une nouvelle variable et lui affecte la valeur obtenue par un calcul. Ce calcul est effectué dans le contexte courant, avec une valeur de `x` qui est 3. Dans cet contexte, `x*x` vaut donc 9.



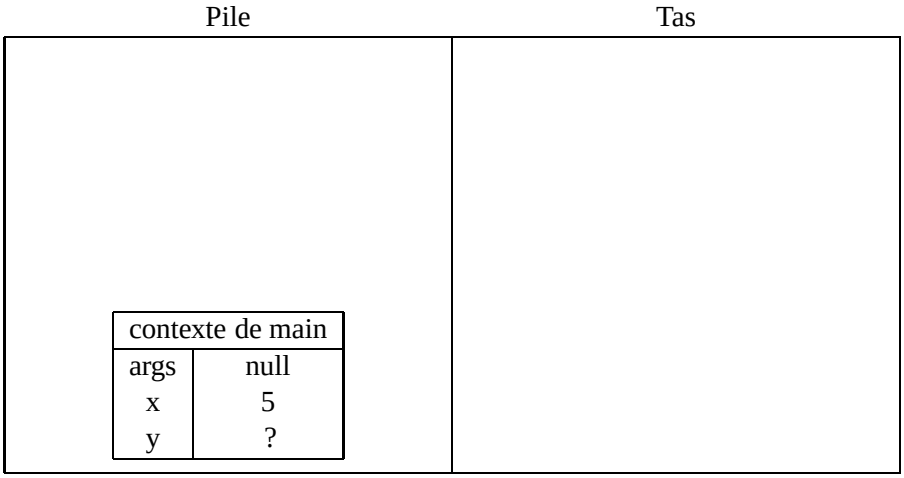
Ligne 4 : l'expression `res` est évaluée dans le contexte courant. La valeur de `res` est 9. L'instruction `return` termine la méthode et renvoie la valeur 9. L'exécution de la méthode `carre` étant terminée, le contexte de cette méthode est retiré de la pile.



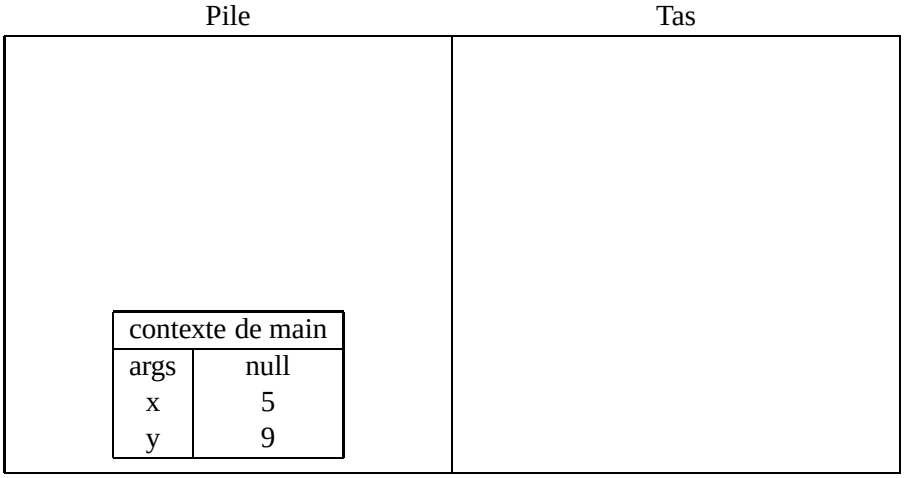
L'exécution de la ligne 8 qui était en attente du résultat de `carre` peut reprendre. La valeur 9 renvoyée par la méthode est inscrite dans le contexte courant à la ligne `res`.



La ligne 9 consiste à calculer la valeur de `res` dans le contexte courant. C'est 9. Puis à la renvoyer. La méthode étant terminée, le contexte correspondant est retiré de la pile.



La ligne 8 qui était en cours d'exécution peut être terminée. La valeur 9 est inscrite à la ligne y du contexte courant.



La ligne 15 s'exécute : la valeur à afficher est recherchée à la ligne y du contexte courant. C'est 9 et cette valeur est affichée par la méthde Terminal.ecrireIntln. Notons au passage que c'est un nouvel appel de méthode : un contexte est créé dans la pile pour la méthode Terminal.ecrireIntln et la valeur 9 est inscrite en face du nom de son paramètre.

A la fin de l'exécution de main, le contexte de main est retiré de la pile et plus généralement, toute la mémoire affectée au programme , c'est-à-dire la pile et le tas, est libérée pour d'autres programmes et son contenu est perdu pour toujours.

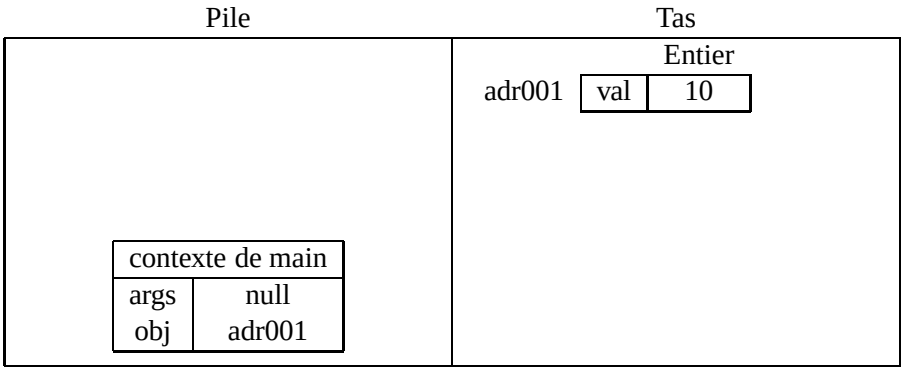
3.9 Paramètres de types références

Lorsqu'on passe un tableau ou un objet en paramètre à une méthode, c'est la référence, l'adresse qui est recopiée dans le contexte de la méthode appelée, pas le tableau ni l'objet. Les deux méthodes peuvent travailler sur le même objet avec deux noms différents, un dans chaque contexte.

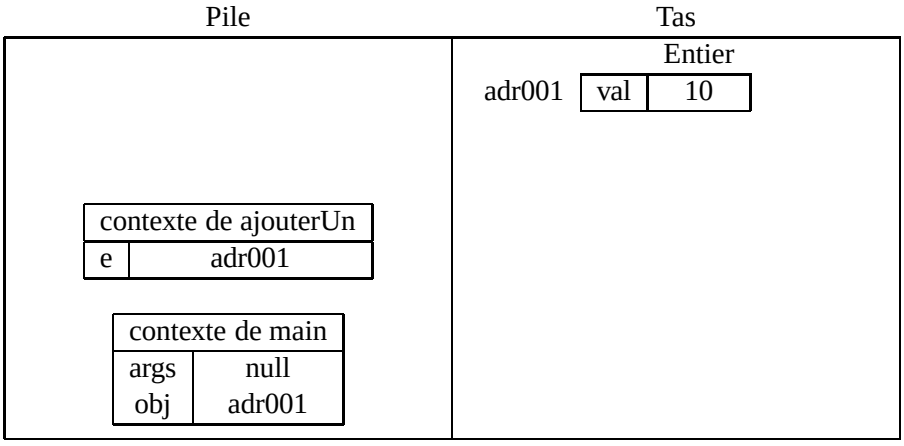
```
class Entier{
    public int val;
    Entier(int x){
```

```
            val = x;
        }
}
public class ExempleMetRef{
        public static void main(String[] args){
            Entier obj = new Entier(10);
            ajouterUn(obj);
            ajouterUn(obj);
            Terminal.ecrireIntln(obj.val);
        }
        public static void ajouterUn(Entier e){
            e.val = e.val + 1;
        }
}
```

L'exécution commence avec la création d'un contexte pour `main`, avec une ligne pour le paramètre `args`. Puis la ligne 9 crée un objet et l'associe au nom `obj` qui est ajouté dans le contexte. A ce stade, la mémoire est la suivante.

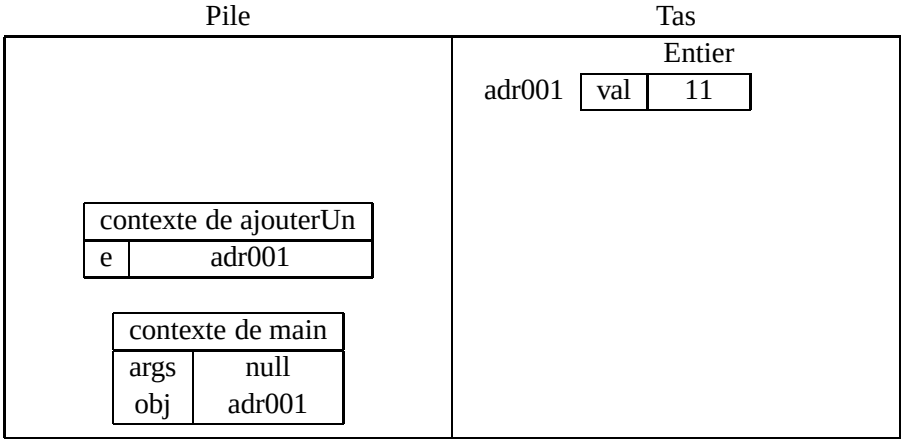


Ligne 10, il y a appel à la méthode `ajouterUn`. La valeur du paramètre est calculée en cherchant la valeur de `obj` dans le contexte de main. Sa valeur est `adr001`. Le contexte de `ajouterUn` est créé avec son paramètre `e` initialisé à la valeur `adr001`.

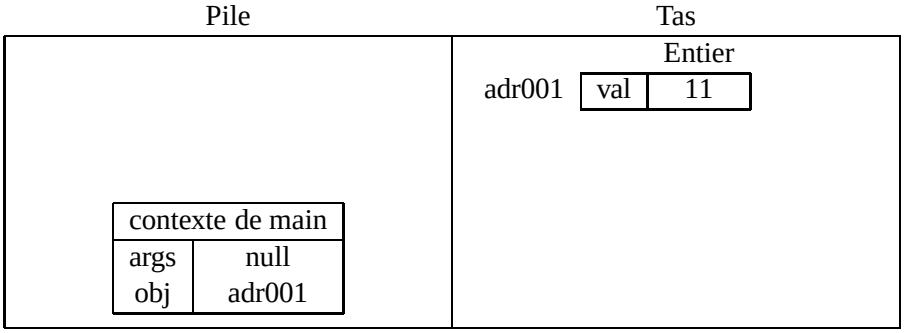


La ligne 15 doit alors être exécutée. La partie gauche de l'affectation est calculée. `e.val + 1` est calculée : `e.val` désigne la ligne `val` du contexte de l'objet stocké à l'adresse `adr001`. La

valeur stockée là est 10. Avec le `+1`, cela donne 11. Ce 11 doit être inscrit dans la variable située à gauche du = : `e.val`.

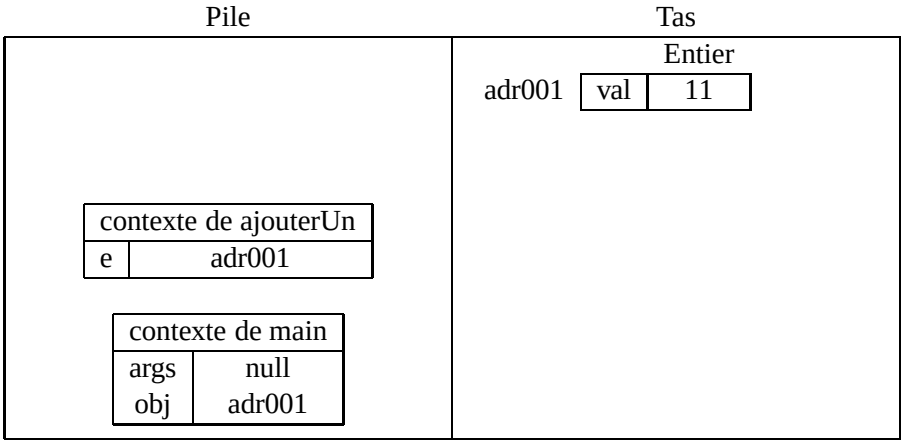


L'exécution de la méthode est terminée. Cette méthode ne renvoie rien : il n'y a pas de return. Le contexte de `ajouterUn` est effacé de la pile.

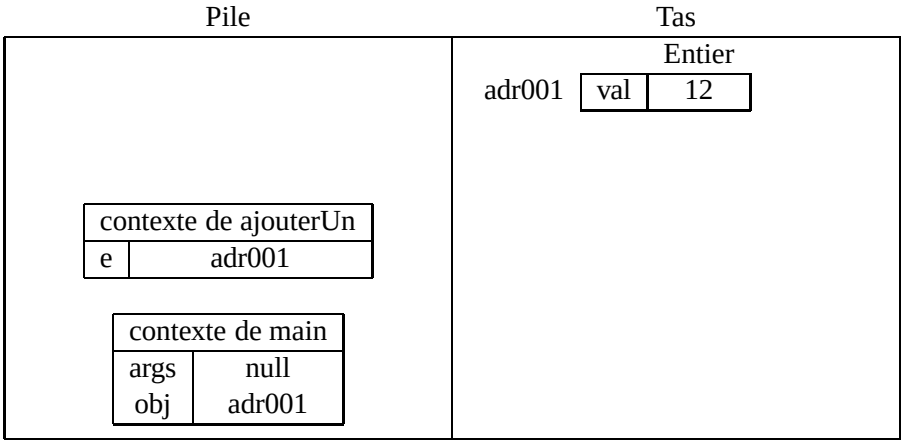


L'exécution de la ligne 10 est terminée avec la fin de la méthode. L'exécution de la ligne 11 commence. C'est un appel identique à celui de la ligne 10, mais entre temps, une variable du tas a été modifiée.

Comme pour la ligne 10, la valeur d'`obj` est cherchée dans le contexte. Cette valeur est `adr001`. Un nouvel contexte est créé pour `ajouterUn` avec le paramètre `e` initialisé à `adr001`.



La ligne 15 doit être exécutée dans le contexte de `ajouterUn`.



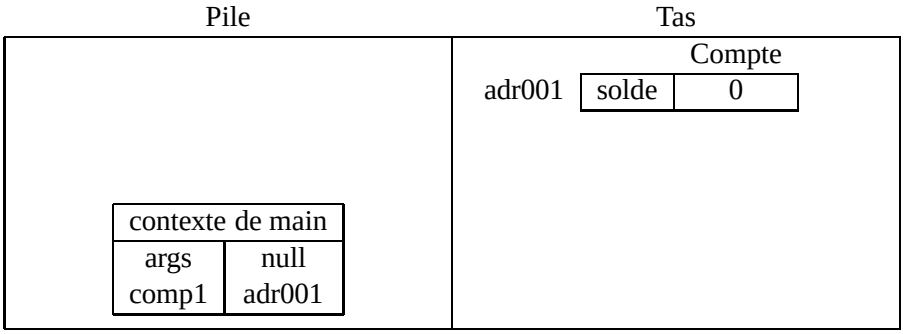
La méthode est terminée. Son contexte est effacé de la pile. Il ne reste alors qu'à exécuter la ligne 12 qui va afficher le contenu de la variable `val` de l'objet stocké à l'adresse `adr001`, c'est à dire 12.

3.10 Appels de méthodes sur un objet

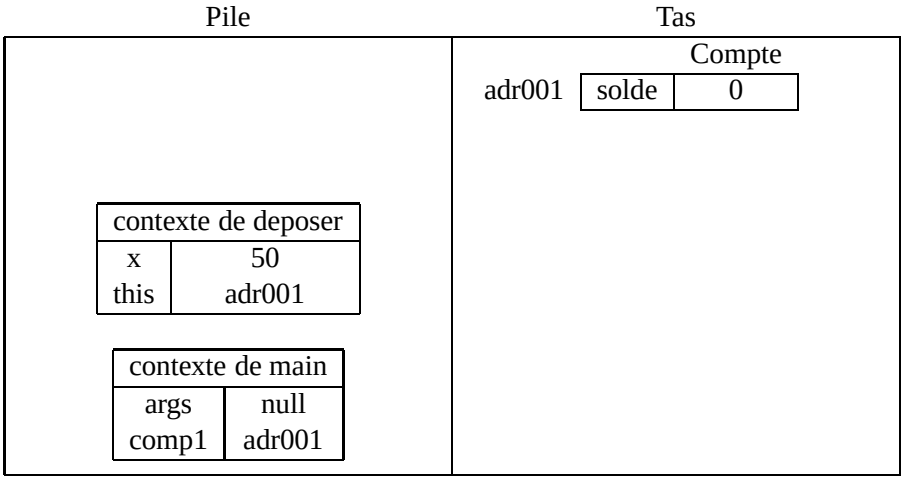
Nos exemples jusqu'ici ont comporté uniquement des méthodes statiques. Nous allons maintenant voir un exemple avec une méthode d'instance.

```
class Compte{
    public int solde=0;
    void deposer(int x){
        this.solde = this.solde + x;
    }
}
public class ExempleMetRef{
    public static void main(String[] args){
        Entier comp1 = new Compte();
        comp1.deposer(50);
        Terminal.ecrireIntln(comp1.solde);
    }
}
```

Comme d'habitude, la méthode main commence avec un contexte comportant son paramètre `args`. Puis un objet de type `Compte` est créé à une adresse `adr001` et un nouveau nom `comp1` est ajouté au contexte avec cette valeur.



La ligne 10 est un appel de la méthode `deposer` sur l'objet `comp1` avec comme paramètre la valeur 50. Un contexte d'exécution est créé avec deux lignes : une pour le paramètre `x` initialisé à 50 et une autre pour recevoir la valeur de `this`, ici la reférence `adr001`.



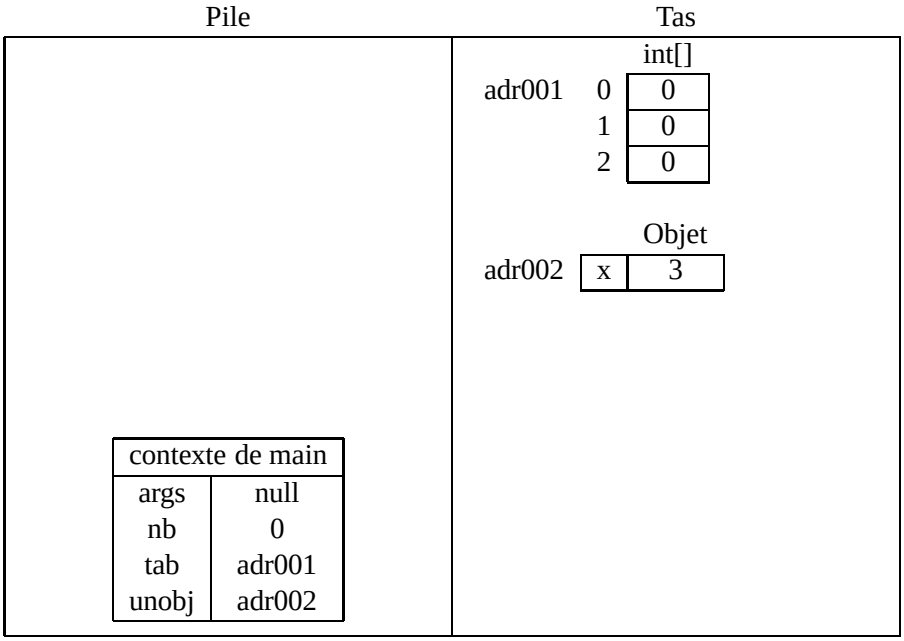
Dans le contexte de `deposer`, la ligne numéro 4 est exécutée. La partie droite de l'affectation est d'abord calculée : `this.solde + x` dans cet contexte vaut le contenu de la variable `solde` de l'objet situé à l'adresse `adr001`, c'est à dire 0, plus le contenu de x, c'est à dire 50. Le total vaut 50. Il est inscrit dans la variable `solde` de l'objet situé à l'adresse `adr001`.

C'est ainsi que l'appel d'une méthode sur un objet conduit à l'ajout au contexte de cette méthode d'une ligne avec le nom `this` et l'adresse de l'objet sur lequel la méthode est appelée.

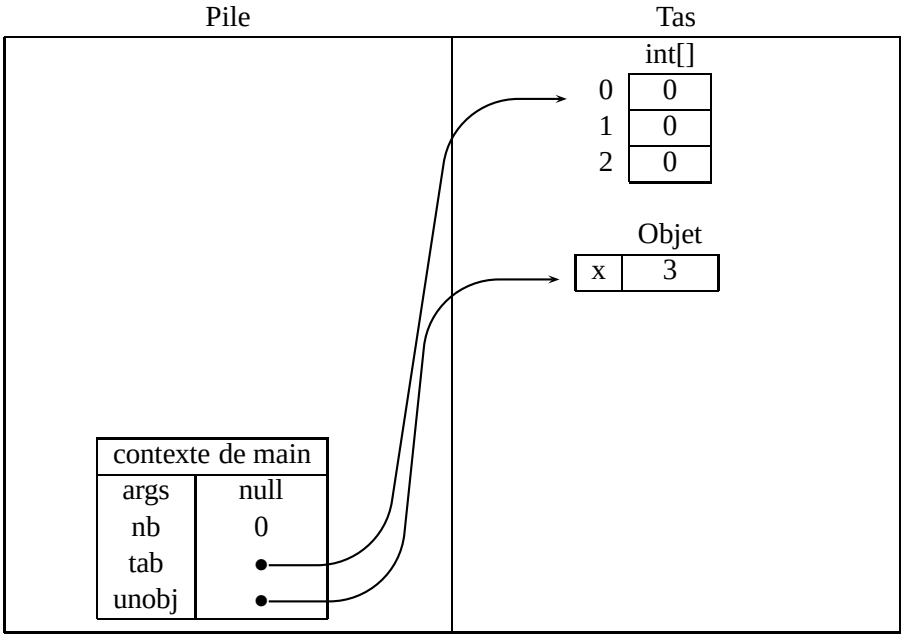
3.10.1 Remplacement des adresses par des flèches

Lorsqu'on représente un état de la mémoire, l'utilisation des adresses n'est pas toujours très parlant. 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.



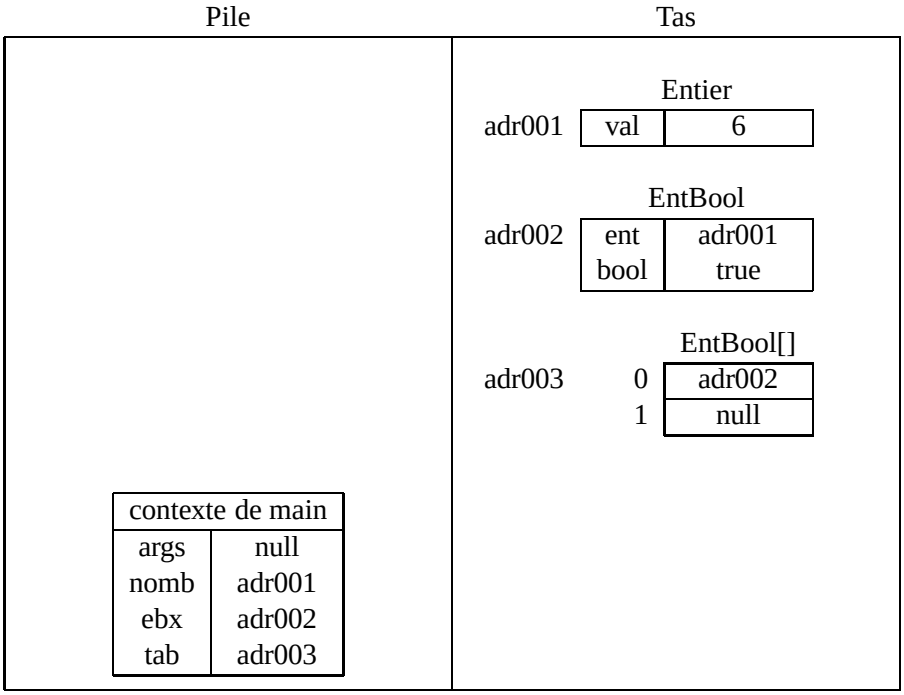
3.11 Chemin d'accès à une case mémoire

Prenons un exemple un peu complexe avec tableaux et objets.

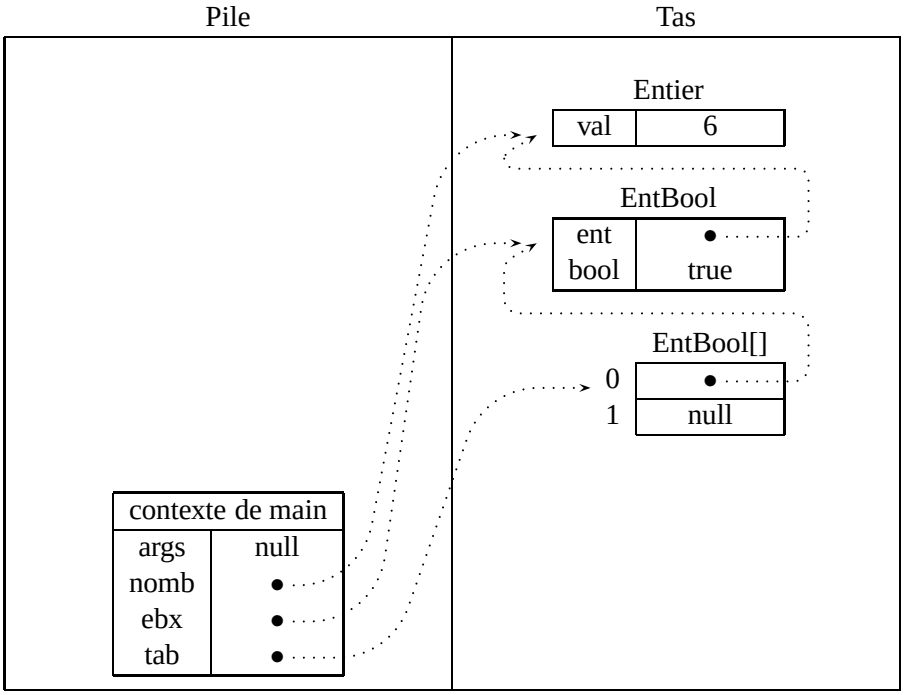
```
class Entier{
    public int val;
    Entier(int x){
        val = x;
```

```
    }
}
class EntBool{
    Entier ent;
    boolean bool;
    EntBool(Entier e, boolean b){
        ent = e;
        bool = b;
    }
}
public class ExempleComp{
    public static void main(String[] args){
        Entier nomb = new Entier(6);
        EntBool ebx = new EntBool(nomb,true);
        EntBool[] tab = new EntBool[2];
        tab[0]=ebx;
        nomb.val = 7;
        ebx.ent.val = 8;
        tab[0].ent.val = 9;
    }
}
```

A la fin de l'exécution de la ligne 20, on a trois noms de variables dans le contexte d'exécution de main et on a créé trois choses dans le tas avec autant de new : un objet de type `Entier`, un objet de type `EntBool` et un tableau à deux cases de type `EntBool[]`. Et ligne 20, on a affecté le seul objet de type `EntBool` à la première case du tableau. En représentation avec adresses, cela nous donne la situation suivante.



En notation avec des flèches, le même état mémoire s'écrit de la façons suivante.



Pour désigner un emplacement en mémoire, que ce soit pour y chercher une valeur (par exemple à gauche d'une affectation) ou pour y inscrire une valeur, il faut écrire un chemin d'accès.

Ce chemin commence souvent par le nom d'une variable du contexte actif, celui qui est en haut de pile. Puis une ou plusieurs flèche sont parcourues. Il y a deux façon de noter le fait de suivre une flèche :

- si la flèche arrive sur un objet, la notation est le point.
- si la flèche arrive sur un tableau, ce sont les crochets qui notent le fait de suivre la flèche, et l'entier entre les crochets nous dit de combien de cases il faut descendre une fois arrivé au tableau.

Dans notre exemple, le chemin le plus long part du contexte, depuis `tab`, passe par la première case du tableau, puis par l'objet de type `EntBool` et sa variable `ent` pour arriver sur l'objet de type `Entier` et sa variable `val`. Ce chemin s'écrit : `tab[0].ent.val` et il désigne la valeur 6 inscrite dans le tas. On voit qu'ici, il y a une paire de crochets et deux points, ce qui fait un total de 3 flèches à suivre à travers la mémoire.

Les structures de données qui utilisent des références forment un graphe dans la mémoire de l'ordinateur. On verra dans un chapitre ultérieur que ce graphe peut être cyclique. Pour l'instant, on considère un graphe orienté acyclique, dans lequel plusieurs chemins peuvent aboutir au même endroit.

Dans notre exemple, il y a trois chemins différents pour aboutir au 6 de l'objet de type `Entier` :

- `nomb.val`
- `ebx.ent.val`
- `tab[0].ent.val`

Un nom, s'il n'est pas suivi d'un point ou d'un crochet, désigne un espace mémoire dans la pile. S'il est suivi d'un point ou d'un crochet, le nom est le début d'un chemin qui désigne un espace mémoire du tas.