Références et types primitifs en java

Mickaël Péchaud

Mars 2008

Table des matières

Introdution

Ce document tente de donner quelques éclaircissements sur comment manipuler les variables en java, point qui est souvent assez mal compris. Il s'adresse à un public déjà un peu familiarisé avec java, et éventuellement ayant une connaissance d'autres langages de programmations impératifs ou orientés objet. L'exposé est volontairement schématique, mais j'espère qu'il pourra permettre de mieux comprendre le fonctionnement des variables en java.

1 Types primitifs

Les *types* primitifs sont des types natifs de java. Pour les gens habitués à des langages impératifs tels que *C*, ils correspondent aux types de base dont on a besoin dans tout langage pour représenter les entiers, les réels, les caractères, etc, etc...

Par convention d'écriture, les types primitifs commencent toujours par une minuscule.

1.1 Petite liste

Voici une liste non-exhaustive des types primitifs en java.

byte 8 bits : entiers relatifs courts

int 32 bits : entiers relatifs

char 16 bits : caractères

double 64 bits : rééls

boolean 1 bit : booléen - vrai (*true*) ou faux (*false*)

1.2 Variables pour les types primitifs

1.2.1 Déclaration/Initialisation

Considérons le bout de code suivant :

```
int i;
```

C'est ce qu'on appelle une *déclaration* de la variable de type *int i*.

À l'exécution de ce code, *java* réserve un espace mémoire de 32 bits, permettant de stocker un entier. Cet espace mémoire correspond uniquement à la variable *i*.

Cet entier n'a pour l'instant pas de valeur, ce qui fait que si l'on écrit :

```
int i;
System.out.println(i);
```

pour essayer d'afficher la valeur de *i*, il va y avoir une erreur à la *compilation*. Java étant plus paranoïaque que d'autres langages comme le *C* ou le *C++*, il empêche d'utiliser une variable dont il pense qu'elle a pu ne pas être initialisée.

Initialisons donc. Il y a deux façons de faire ça.

- On peut initialiser *lors de la déclaration* en utilisant la syntaxe suivante :

```
int i=2;
System.out.println(i);
```

- On peut initialiser plus tard, en utilisant une *affectation* :

```
int i;
int i=2;
System.out.println(i);
```

et là, plus de problème.

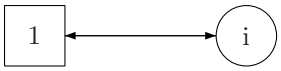
1.2.2 Affection

L'affectation des types primitifs en java fonctionne *par valeur*.
Voyons ce que celà signifie sur un petit exemple.

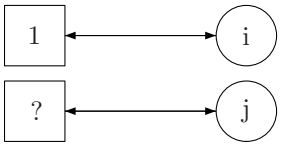
```
int i = 1;     // (1)

int j;         // (2)

j = i;         // (3)
```

Ce bout de programme a le comportement suivant :

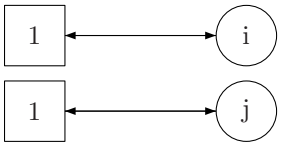
1. Création et initialisation d'une variable i : un espace mémoire de 32 bits est associé à i. Dans cet espace mémoire est écrit un encodage de l'entier « 1 ».



   (dans tous les schémas qui suivent, un cercle indique un nom de variable, et un carré désigne une case mémoire).
2. Création d'une variable j : un espace mémoire de 32 bits est associé à i.



3. Affectation de la *valeur* de i à j : le bout d'espace mémoire correspondant à i est recopié dans le bout d'espace mémoire correspondant à j.



Notez qu'il y a correspondance univoque entre i est son espace mémoire d'une part, et entre j et son espace mémoire d'autre part.

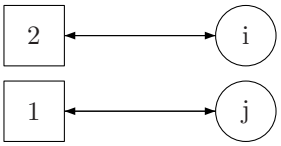
Après l'éxécution de ce code, **il n'y a donc plus aucun lien entre i et j.**

En particulier, si on écrit le code que voici :

```
int i = 1;
int j;
j = i;

i = 2;                  //(1)
```

```
System.out.println(i);
System.out.println(j);
```

après (1), on a le schéma suivant :



Le résultat affiché à l'écran est donc

```
2
1
```

2 Références

2.1 Déclaration/Initialisation

Java est un langage *orienté objet* : celà signifie en particulier que le programmeur peut définir un certain nombre de *classes*, que l'on peut voir en première approche comme des *types* ou des structures de données intelligentes. Une fois une classe définie, il est possible de créer des *objets* appartenant à la cette classe. Chaque objet est appelé *instance* de la classe.

Voici un petit exemple. Commençons par définir une classe très simple (et stupide à de nombreux points de vue), permettant de décrire des pays.

```
public class Pays
{
  public int nombreDHabitants;
  public double superficie;

  public Pays()
  {
    nombreDHabitant=0;
    superficie=0;
  }

  public Pays(int nombreDHabitants, double superficie)
  {
    this.nombreDHabitants=nombreDHabitants;
    this.superficie=superficie;
  }
}
```

Cette classe comporte 2 *champs* correspondant au nombre d'habitants et à la superficie du pays.

Par ailleurs, elle dispose de deux constructeurs :

- un constructeur sans argument qui initialise les champs à zéro,
- un constructeur avec deux arguments, permettant d'initialiser les champs.

Supposons maintenant que l'on souhaite utiliser cette classe :

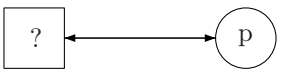
```
Pays p;
```

cette commande *déclare* une variable p correspondant à un objet de la classe Pays. Mais lors de son éxécution :

- **aucun objet (aucune instance) de la classe Pays n'est créée**
- **l'espace mémoire nécessaire à stocker un Pays n'est même pas réservé**.

Que se passe-t-il alors ? La variable p correspond en fait à une **référence**. En première approximation, une référence correspond à une adresse en mémoire.

Le code ci-dessus réserve donc un espace mémoire permettant de stocker une référence. Cet espace mémoire n'est pas initialisé :

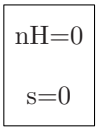


Créons maintenant rééllement un objet qui instancie la classe *Pays*.

```
new Pays();
```

La commande permettant d'instancier un objet est **new**.
Lors de l'appel à `new Pays()`, il se passe 2 choses :
- Un espace mémoire est alloué permettant de stocker un objet de type de *Pays*.
- Un constructeur (ici le constructeur sans argument est appelé) pour initialiser l'objet.

On a donc quelquechose qui ressemble à ça :



Remettons tout ceci ensemble :

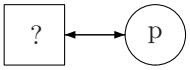
```
Pays p;
p = new Pays();
```

qui pourrait aussi s'écrire

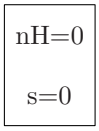
```
Pays p = new Pays();
```

il y a trois étapes principales dans l'exécution de ce code :

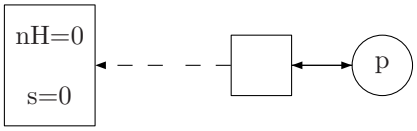
1. déclaration d'une référence :



2. création d'un objet à l'aide de l'opérateur `new` :



3. affectation : la référence (stocké dans l'espace mémoire reservé lors de la déclaration de p) prend la valeur de l'adresse mémoire de l'objet créé avec `new`, ce que nous pouvons représenter de la façon suivante :



On dit que l'objet créé par `new` est **référencé** par la variable p.

2.2 Affectation de références

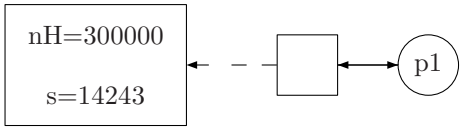
Nous sommes maintenant mieux armés pour comprendre le comportement des variables de références en java.

```
Pays p1 = new Pays(300000, 14243); //(1)
Pays p2;                           //(2)
p2=p1;                             //(3)
```

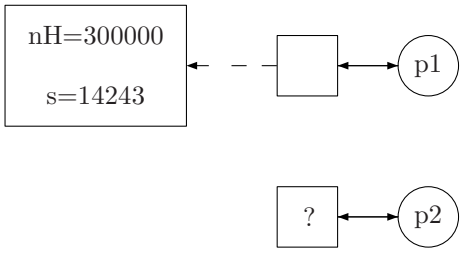
Dans ce code, même si 2 variables correspondant à des pays sont crées, **une seule instance de Pays est créée.** Une règle générale : une instance ne peut être créée que par `new`. Il n'y qu'un seul `new` appelé lors de l'exécution de ce code, donc un seul objet créé.

Suivons l'exécution pas à pas :

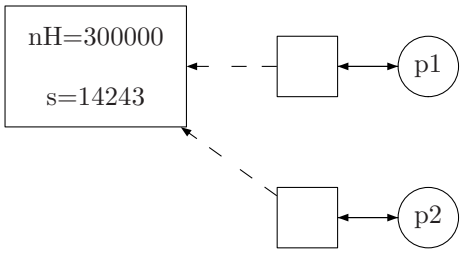
1. `Pays p1 = new Pays(300000, 14243);` Création d'une instance de pays, référencée par la variable $p1$. Cette fois-ci, c'est le constructeur avec 2 arguments qui est utilisé pour l'initialisation.



2. `Pays p2;` Déclaration d'une variable $p2$: aucun pays n'est créé, on réserve juste un espace mémoire pour une référence, sans l'initialiser



3. `p2=p1;` La **valeur** de la référence correspondant à $p1$ est recopiée dans l'espace mémoire correspondant à $p2$: $p2$ référence désormais l'objet qui était référencé par $p1$.



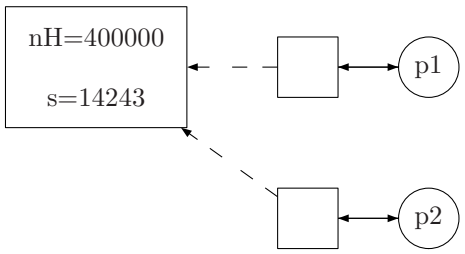
J'insiste : à la suite de l'éxécution de ce code, un seul objet a été créé, et cet objet est référencé par les 2 variables $p1$ et $p2$. La ligne `p2=p1` effectue une **copie de références**, et non pas une **copie d'objets**.

S'il y a toujours correspondance univoque entre $p1$ et l'espace mémoire contenant la référence associé d'une part, et $p2$ et l'espace mémoire contenant la référence associé d'autre part (flêches pleines), $p1$ et $p2$ référencent maintenant un objet commun (flêches pointillées).

Ainsi, si l'on écrit le code suivant :

```
Pays p1 = new Pays(300000, 14243);
Pays p2;
p2=p1;
p2.surface = 400000;                //(1)
System.our.println(p1.surface);     //(2)
```

La ligne (1) modifie un champ de l'objet référencé par $p2$:



mais cet objet étant aussi référencé par $p1$, l'exécution de la ligne (2) affichera

```
400000
```

Une autre façon encore plus informelle de voir `p2=p1` est de traduire cette ligne par : « *p2 est un nouveau nom pour l'objet référencé par p1* ».

2.3 D'autres exemples

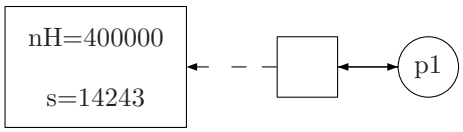
Voici d'autres exemples, pour être sur que l'on comprend ce qui se passe :

Exemple 1

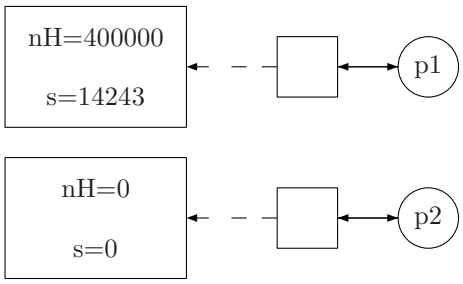
```
Pays p1 = new Pays(300000, 14243);  //(1)
Pays p2 = new Pays();               //(2)
p2=p1;                              //(3)
```

2 objets sont créés dans ce bout de code.

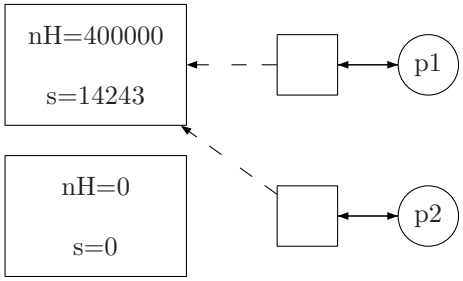
1. `Pays p1 = new Pays(300000, 14243);` Création du premier objet, référencé par $p1$.



2. `Pays p2 = new Pays();` Création du second objet, référencé par $p2$.



3. `p2=p1;` On recopie l'adresse de l'objet référencé par $p1$ dans l'espace mémoire aloué à $p2$, ce qui revient à dire que $p2$ référence maintenant le même objet que $p1$. Graphiquement, on prend la flêche pointillée partant de $p2$, et on la fait pointer vers le premier objet :



À l'issue de l'exécution de ce code, $p1$ et $p2$ référencent le même objet.

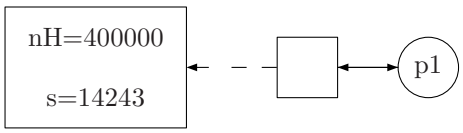
L'objet initialement référencé par $p2$ n'est plus référencé par personne. Il est désormais inaccessible : que pourrait-on écrire pour modifier ces champs ? Plus aucune variable ne désigne cet objet. Il occupe donc une place inutile en mémoire, et sera détruit par le *ramasse-miette* de java, dont le travail est exactement de détruire les objets qui ne sont plus référencés par rien.

Exemple 2

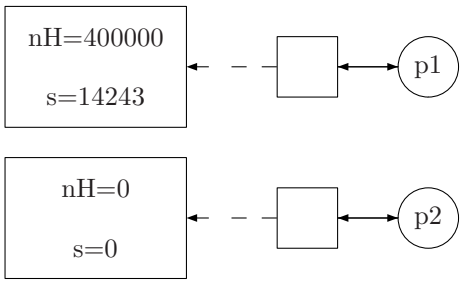
```
Pays p1 = new Pays(300000, 14243);  //(1)
Pays p2 = new Pays();               //(2)
Pays p3=p1;                         //(3)
p1=p2;                              //(4)
p2=p3;                              //(5)
```

Ici aussi, seul 2 objets sont créés. Même s'il y a 3 références.

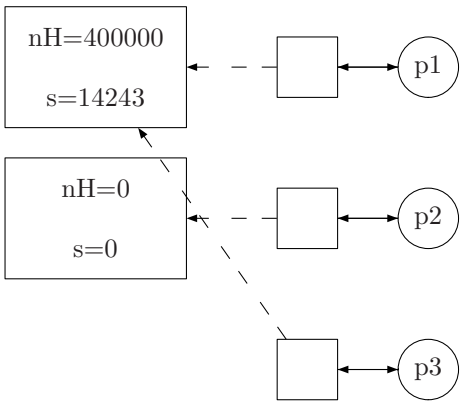
1. `Pays p1 = new Pays(300000, 14243);` Création du premier objet, référencé par $p1$.



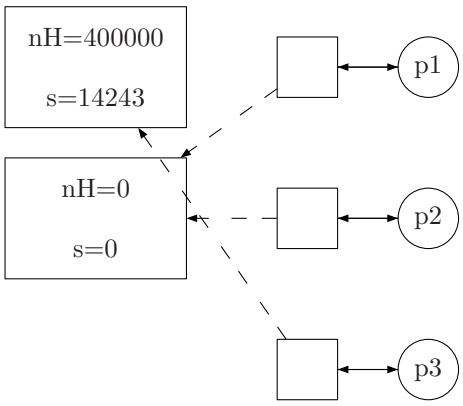
2. `Pays p2 = new Pays();` Création du second objet, référencé par $p2$.



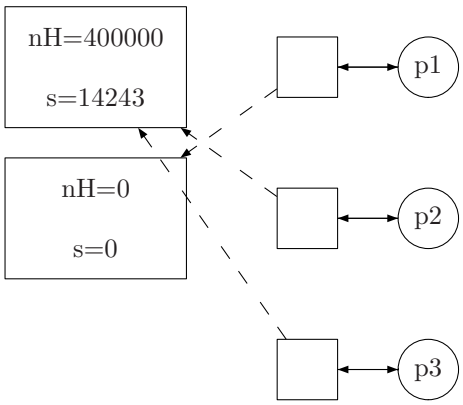
3. `Pays p3=p1;` On déclare une variable $p3$. On recopie l'adresse de l'objet référencé par $p1$ dans l'espace mémoire aloué à $p3$.



4. `p1=p2;` On recopie l'adresse de l'objet référencé par $p1$ dans l'espace mémoire aloué à $p3$.



5. `p2=p3;` On recopie l'adresse de l'objet référencé par $p1$ dans l'espace mémoire aloué à $p3$.



Résultat des courses :
• $p2$ et $p3$ référencent maintenant le même objet : celui qui était référencé par $p1$ au départ.
• $p1$ référence maintenant l'objet qui était référencé par $p2$ au départ.
En particulier, ce code permute les objets référencés par $p1$ et $p2$.

3 Passage de paramètres

En java, les passages de paramètres sont fait *par valeur*. Reste à savoir quelle valeur...

3.1 Types primitifs

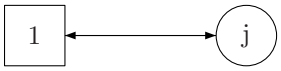
Considérons la fonction suivante :

```
static void augmente(int i)  // (2)
{
  i=i+1;                     // (3)
}
```

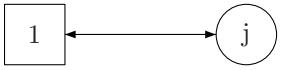
et un appel de cette fonction :

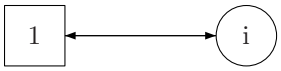
```
int j=1;                //(1)
augmente(j);            //(2)
System.out.println(j); //(4)
```

1. `int j=1;` : nous avons déjà vu ce que celà donne.



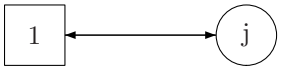
2. `augmente(j);` . Lors de l'appel de la fonction, le passage de paramètre s'effectue *par valeur*. Celà signifie qu'une variable i locale à fonction est créée, et que l'on recopie dans l'espace mémoire associé à cette variable la *valeur* associée à la variable j :

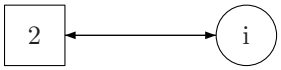




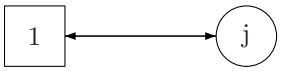
Notez que la variable j n'est pas visible depuis la fonction.

3. `i=i1;`+. La valeur associée à i est incrémentée de 1





4. `System.out.println(j);` On sort de la fonction. La variable locale est effacée :



et le résultat

```
1
```

est affiché.

Conclusion : cette méthode ne fait rien.

Si on passe une variable correspondant à un type primitif à une fonction, sa valeur ne peut être modifiée par la fonction.

3.2 Références

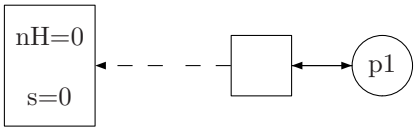
Faisons maintenant quelquechose de similaire avec des références :

```
static void augmenteSurface(Pays p)  // (2)
{
  p.surface=p.surface+1;             // (3)
}
```

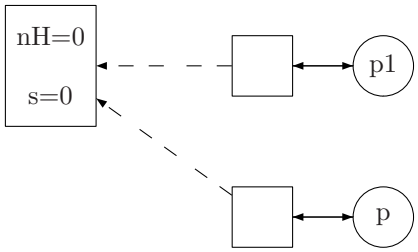
et un appel de cette fonction :

```
Pays p1=new Pays();             //(1)
augmenteSurface(p1);            //(2)
System.out.println(p1.surface); //(4)
```

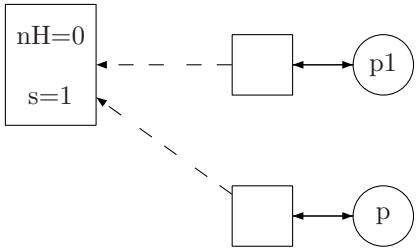
1. `Pays p1=new Pays();`



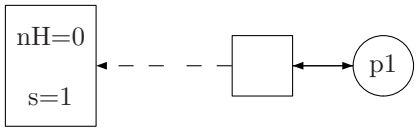
2. `augmenteSurface(p);` . Lors de l'appel de la fonction, le passage de paramètre s'effectue *par valeur*. Mais cette fois-ci, **c'est la référence qui est passée par valeur** : une variable locale p est déclarée, et la valeur de la référence correspondant à $p1$ est recopié dans l'espace mémoire associé à p :



3. `p.surface=p.surface1;+`. La valeur de la surface du pays référencé par p est incrémentée de 1 :



4. `System.out.println(p1.surface);` On sort de la fonction. La variable locale est effacée



et le résultat affiché est

```
1
```

Conclusion :

Si on passe une variable correspondant à une référence une fonction, l'objet référencé peut être modifié par la fonction.

En revanche, comme va le montrer le dernier exemple suivant :

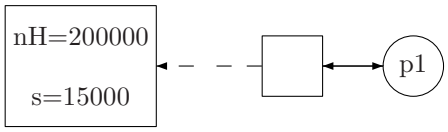
Si on passe une variable correspondant à une référence une fonction, la référence ne peut être modifiée par la fonction.

```
static void nouveau(Pays p)  // (2)
{
  p=new Pays();              // (3)
}
```

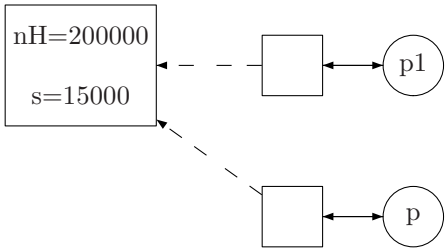
et un appel de cette fonction :

```
Pays p1=new Pays(200000, 15000);  //(1)
nouveau(p1);                      //(2)
System.out.println(p1.surface);   //(4)
```

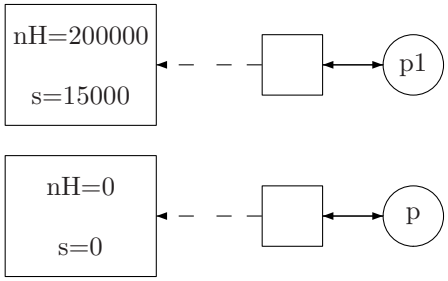
1. `Pays p1=new Pays(200000, 15000);`



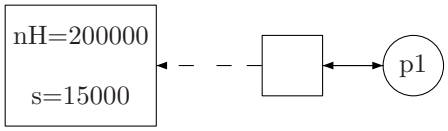
2. `nouveau(p1);`. De même que précédemment :



3. `p=new Pays();`. Cette ligne créé un nouveau pays, et copie dans l'espace mémoire associé à p l'adresse de ce nouvel objet. p référence désormais ce nouvel objet, mais on n'a pas touché à l'espace mémoire associé à $p1$:



4. `System.out.println(p1.surface);` On sort de la fonction. La variable locale p est effacée



En particulier, l'objet créé à l'intérieur de la fonction n'est plus référencé par personne.
Le résultat affiché est

```
200000
```

On n'a donc pas modifié la référence associée à $p1$.

Si on passe une variable correspondant à une référence une fonction, la référence ne peut être modifiée par la fonction.

4 Retour de fonctions

Considérons une fonction

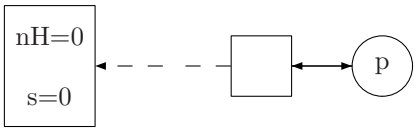
```
Pays pouet()
{
  Pays p = new Pays();
  return p;
}
```

Qu'est-ce-qui est retourné par la fonction ? On a ici le même comportment que pour les passages d'arguments : cette fonction retourne la valeur de la référence associée à la variable locale p.

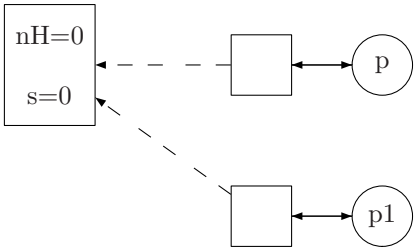
Ainsi lors de l'appel suivant :

```
Pays p1=pouet();
```

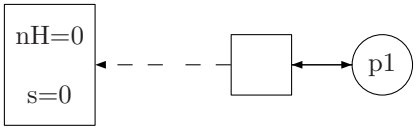
1. `Pays p=new Pays();` Création d'un objet, référencé par la variable locale p.



2. `Pays p1=pouet()`; la référence de l'objet est renvoyée par la fonction. $p1$ référence donc maintenant l'objet créé dans la fonction :



et parallèlement, la variable locale p est supprimée, d'où le résultat final suivant :



La variable locale, par définition, n'a pas survécue après le retour de la fonction. En revanche, **un objet créé à l'intérieur d'une fonction peut très bien perdurer après l'appel de cette fonction**.

5 Deux fausses exceptions

J'ai affirmé dans une section précédente que :
« Une règle générale : une instance ne peut être créée que par `new`. »
Il semble y avoir deux exceptions à cette règle, qui en fait n'en sont pas.

5.1 Tableau

En java, les tableaux sont des objets (que ce soients des tableaux de type primitifs, ou d'objets).
Écrivons le code suivant.

```
int[] t={1, 2, 3, 4};
```

Il semble que l'on ai créé un objet de type « tableau d'entiers »sans utiliser `new`.
C'est une illusion, le code écrit ci-dessus étant simplement un racourci pour :

```
int[] t= new int[4];
int[0]=1;
int[1]=2;
int[2]=3;
int[3]=4;
```

Il s'agit juste de *sucre syntaxique*, rendant moins pénible la création d'un tableau.

5.2 String

On a quelquechose de similaire avec les chaînes de caractères :

```
String s=¨bonjour¨;
```

un objet instanciant la classe *String* a bien été créé, sans que l'on fasse appel à `new`.
C'est également une illusion, le code ci-dessus étant en fait équivalent à

```
char data[] = {'b', 'o', 'n', 'j', 'o', 'u', 'r'};
String str = new String(data);
```

6 Comparaisons

Un dernier point important, souvent source d'erreur.
Dans le code suivant,

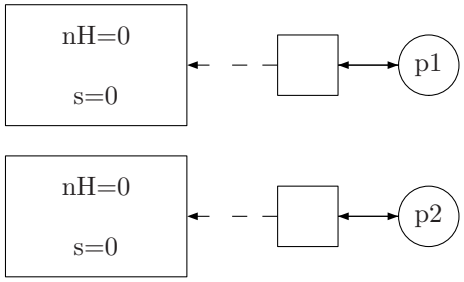
```
int i1 = 3;
int i2 = 3;
if (i1 == i2) ...      //(1)
```

le test en (1) renvoie vrai. Ce sont bien les valeurs de $i1$ et $i2$ qui sont comparées.

En revanche dans

```
Pays p1 = new Pays();
Pays p2 = new Pays();
if (p1 == p2) ...      //(1)
```

quel va être le résultat du test en (1) ? Une autre façon de poser cette question est : qu'est-ce-qui est comparé par == ?



il y a deux possibilités :

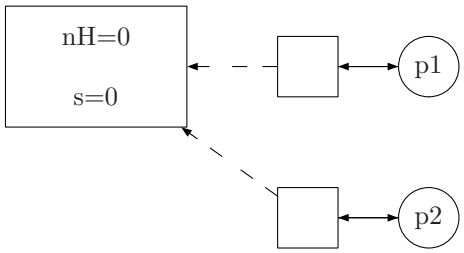
- si == compare les objets (valeurs des rectangles de gauche), le résultat va être vrai, les 2 objets ayant la même description en mémoire.
- si == compare les références (valeurs des rectangles du milieu), le résultat va être faux, les 2 objets ayant des emplacement différents en mémoire.

Le résultat est **faux** : == **compare les références en java**.

En revanche, dans le code suivant

```
Pays p1 = new Pays();
Pays p2 = p1;
if (p1 == p2) ...      //(1)
```

correspondant au schéma suivant :



Les adresses associées à $p1$ et $p2$ sont identiques ($p1$ et $p2$ référencent le même objet), et le résultat du test va donc être **vrai**.

`==` effectue une comparaison des références, encore appelée *comparaison* superficielle. Pour effectuer une *comparaison profonde*, comparant effectivement les objets, il va falloir créer une méthode dans la classe *Pays* :

```
public boolean egal(Pays p)
{
  if ((nombreDHabitants==p.nombreDHabitants) && surface==p.surface)
   return true;
  else
   return false;
}
```

Si l'on écrit

```
Pays p1 = new Pays();
Pays p2 = new Pays();
if (p1.egal(p2)) ...
```

le test renverra alors *vrai*.

7 Exercices

7.1 Exercice 1

On définit la méthode suivante, sensée permuter deux pays :

```
static void permuter(Pays p1, Pays p2)
{
  Pays tmp=p1;
  p1=p2;
  p2=tmp;
}
```

que l'on utilise dans le code suivant :

```
  Pays p1 = new Pays(100000, 200000);
  Pays p2 = new Pays(300000, 400000);
  permute(p1, p2);
  System.out.println(p1.surface);
  System.out.println(p2.surface);
```

Pourquoi ce code affiche-t'il le résultat suivant ?

```
200000
400000
```

7.2 Exercice 2

En dessinant des schémas, justifiez que si `permuter` est écrit de la façon suivante

```
static void permuter(Pays p1, Pays p2)
{
  Pays tmp=new Pays();

  tmp.nombreDHabitants=p1.nombreDHabitants;
  tmp.surface=p1.surface;

  p1.nombreDHabitants=p2.nombreDHabitants;
  p1.surface=p2.surface;

  p2.nombreDHabitants=tmp.nombreDHabitants;
  p2.surface=tmp.surface;
}
```

le code ci-dessus affiche

```
400000
200000
```

7.3 Exercice 3

On définit la méthode suivante :

```
Pays copie(Pays p)
{
  Pays r = new Pays();
  r.nombreDHabitants = p.nombreDHabitants;
  r.surface = p.surface;
  return r;
}
```

En dessinant des schémas, justifiez que le code suivant :

```
Pays p1=new Pays(100000, 20000);
Pays p2=p1.copie();
p2.surface=30000;
System.out.println(p1.surface);
System.out.println(p2.surface);
```

affiche

```
20000
30000
```

Combien d'instances de `Pays` sont-elles créées ?

On dit que `Copie` effectue une *copie profonde* de l'objet, par opposition à l'opérateur d'affectation `=` qui n'effectue qu'une *copie superficielle* des références.