Chapitre 11

Types primitifs et types références

11.1 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;`). 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`. 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.

11.2 Variables de type primitif

Les types primitifs sont : `boolean, char, byte, short, int, long, float, double`. Pour représenter les valeurs de ces types, il faut un nombre fixe de bits : par exemple 8 bits pour les `byte` et 32 pour les `int`.

Lorsqu'on déclare une variable, on donne son type : Une variable déclarée de type primitif contient une valeur de type primitif. Ainsi, lors des déclarations suivantes :

Listing 11.1 – (pas de lien)

```
1 int n;
2 char c = 'a';
3 n=5;
4 double d = 2.0;
```

Le nom `n` désigne un espace mémoire de 32 bits, qui contient l'entier 5 (après exécution de la ligne 3) ; `c` désigne un espace mémoire de 16 bits, qui contient `'a'` et `d` désigne un espace mémoire de 64 bits, qui contient `2.0`. Dans la suite du programme, on peut accéder au contenu de ces espaces mémoire par leur nom : `n, d` ou `c` et modifier ce contenu grâce à l'affectation : `n = 10`.

En résumé, en déclarant une variable d'un type primitif, on donne un nom à un espace mémoire. La taille de cet espace dépend du type de la variable. Lorsque on affecte une valeur à la variable, on met cette valeur dans l'espace mémoire correspondant. C'est en ce sens que nous disons que la variable *contient* une valeur de son type.

11.3 Variables références et objets

11.3.1 Introduction

Que se passe t il lorsque nous créons une variable dont le type est une classe ?

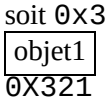
```
Date d = new Date();
```

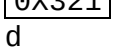
(On suppose que `Date` est la classe définie dans le cours sur les classes et les objets). Une nouvelle variable, `d` est crée. Par ailleurs, `new Date()` crée une instance de la classe `Date`, un objet. Le point clé est le suivant : contrairement aux variables primitives du paragraphe précédent, `d` ne *contient pas un objet* , mais une *référence à un objet*. Si une variable primitive contient des bits qui représentent la valeur qu'elle contient, une variable de type complexe contient des bits qui représentent *une façon d'accéder à l'objet*. Elle ne contient pas l'objet lui même, mais quelque chose qui permet de retrouver l'adresse où se situe l'objet en mémoire. C'est pourquoi nous disons que ces variables contiennent l'adresse de l'objet ou encore un pointeur sur l'objet. Nous appellerons ces variables des variables *références*.

Nous n'avons pas besoin de savoir comment la JVM implémente les références aux objets. C'est sans importance parce que nous ne pouvons les utiliser que pour accéder à l'objet. Nous savons en revanche que pour une JVM donnée, toutes les références sont de même taille quelque soit l'objet qui est référencé.

11.3.2 Détail de la création d'un objet

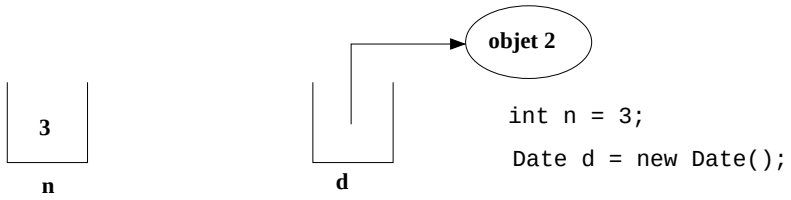
Nous pouvons maintenant détailler le processus qui est à l'oeuvre lors de l'exécution d'une déclaration de variable référence. Prenons l'exemple de `Date d = new Date()`. Ce processus se décompose en trois étapes :

1. `Date d` `= new Date()` : déclaration d'une variable référence.
   De l'espace mémoire est alloué pour une variable référence de nom `d`. Cet espace mémoire est de taille fixe, suffisant pour contenir une adresse. La taille de l'espace mémoire réservé pour `d` ne dépend en rien de la taille qu'il faut pour stocker un objet `Date`, mais dépend de la taille qu'il faut pour stocker une adresse mémoire. A la suite de `Personne p = new Personne();` par exemple, l'espace alloué à `p` serait strictement de même taille que celui alloué à `d`. A cette étape, `p` et `d` ne contiennent encore aucune adresse.
   
   `d`
2. `Date d =` `new Date()` : création d'un objet `Date`.
   Appelons cet objet *objet1*. C'est l'opérateur `new` qui s'occupe de réserver de l'espace mémoire pour stocker un objet. Il détermine combien d'espace est nécessaire pour stocker cet objet et détermine l'adresse où sera stocké cet objet. Imaginons, pour fixer les esprits, que cette adresse soit `0x321` :
   

   `0X321`
3. `Date d` `=` `new Date()` : Liaison de l'objet à la référence. L'adresse ou réside l'objet en mémoire, `0x321` pour notre exemple, est donnée comme valeur à la variable référence `d` :
   

   `d`

Comme nous ne connaissons pas réellement l'adresse de l'objet, nous préférons la représenter graphiquement par une flèche vers l'espace mémoire contenant l'objet. Nous représenterons les 2 sortes de variables de la façon suivante :

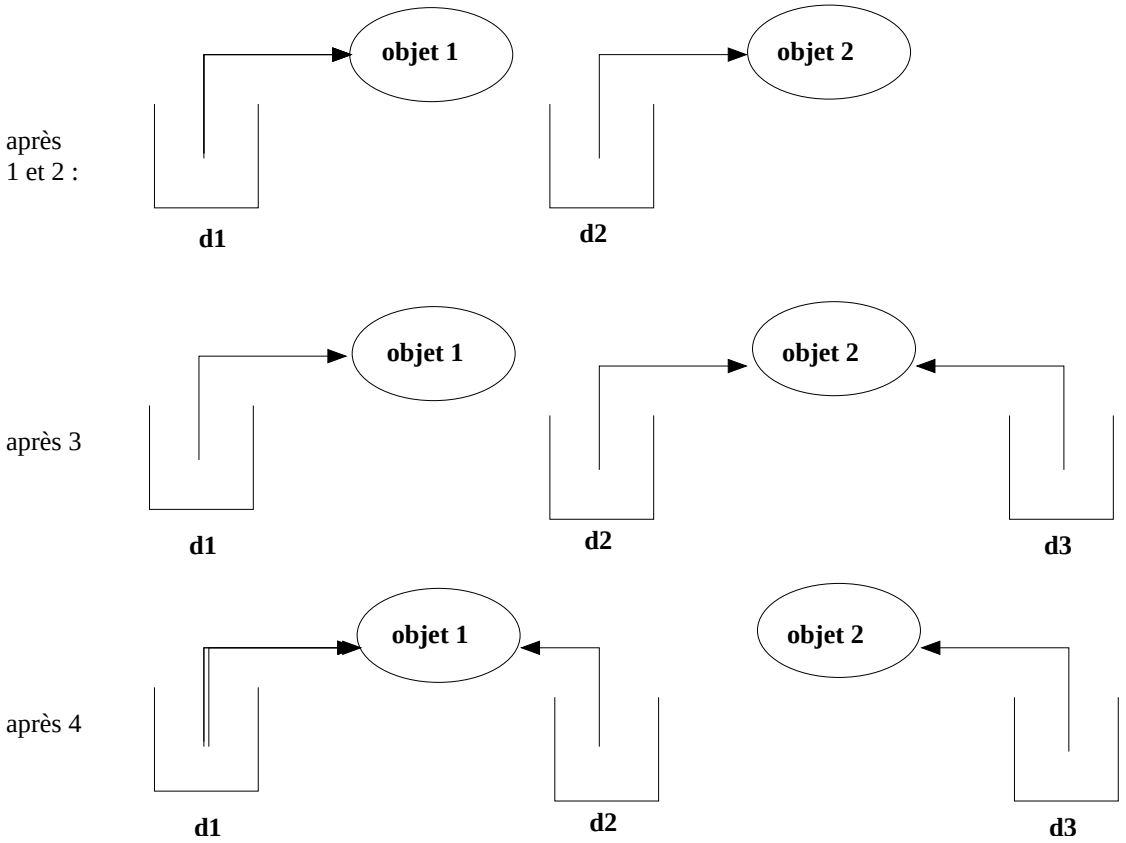


11.3.3 Manipuler les références

Une variable référence contient une adresse vers un objet d'un certain type. Comme toute autre variable, on peut lui donner une valeur par affectation. Comme Java est typé, on ne peut lui donner qu'une valeur de même type, c'est à dire une adresse vers un objet de même type. Mais nous ne connaissons pas explicitement les adresses, donc les deux seules façons de donner une valeur à une variable référence sont d'utiliser `new`, comme nous venons de le voir, ou de lui affecter la valeur d'une autre variable référence de même type.

Listing 11.2 – (pas de lien)

```
1 Date d1 = new Date();
2 Date d2 = new Date();
3 Date d3 = d2;
4 d2=d1;
```



A la suite des 2 premières instructions, nous avons 2 références et 2 objets. En ligne 3, nous avons 3 références mais seulement 2 objets : `d3`, prend comme valeur celle de `d2`, donc l'adresse de l'objet2. `d2` et `d3` référencent le même objet. Ce sont deux façons différentes d'accéder au même objet. En ligne 4, nous avons toujours 3 références et 2 objets : `d2` prends maintenant l'adresse de l'objet1. `d1` et `d2` référencent le même objet.

11.3.4 Manipuler les objets référencés

Une variable référence contient une adresse, un moyen d'accéder à un objet, pas l'objet lui même. Mais comment accéder via la variable à cet objet et comment le modifier ? Nous le savons déjà : par la notation pointée.
Si `d1` est une variable qui référence une date, sa valeur est une adresse, mais `d1.` désigne l'objet date qu'elle référence. Nous pouvons ainsi lui appliquer toutes les méthodes des dates et accéder au variables d'instances `jour, mois, annee` :

Listing 11.3 – (lien vers le code brut)

```
1 public class Chap12a {
2     public static void main(String [] args){
3       Date d1 = new Date();
4       d1.afficherDate();
5       d1.lendemain();
6       d1.jour = d1.jour +1;
7       Terminal.ecrireStringln("Annee " + d1.annee);
8     }
9 }
```

Son Exécution produit :

```
simonot@saturne:> java Chap12a
1 , 1 , 1
1
```

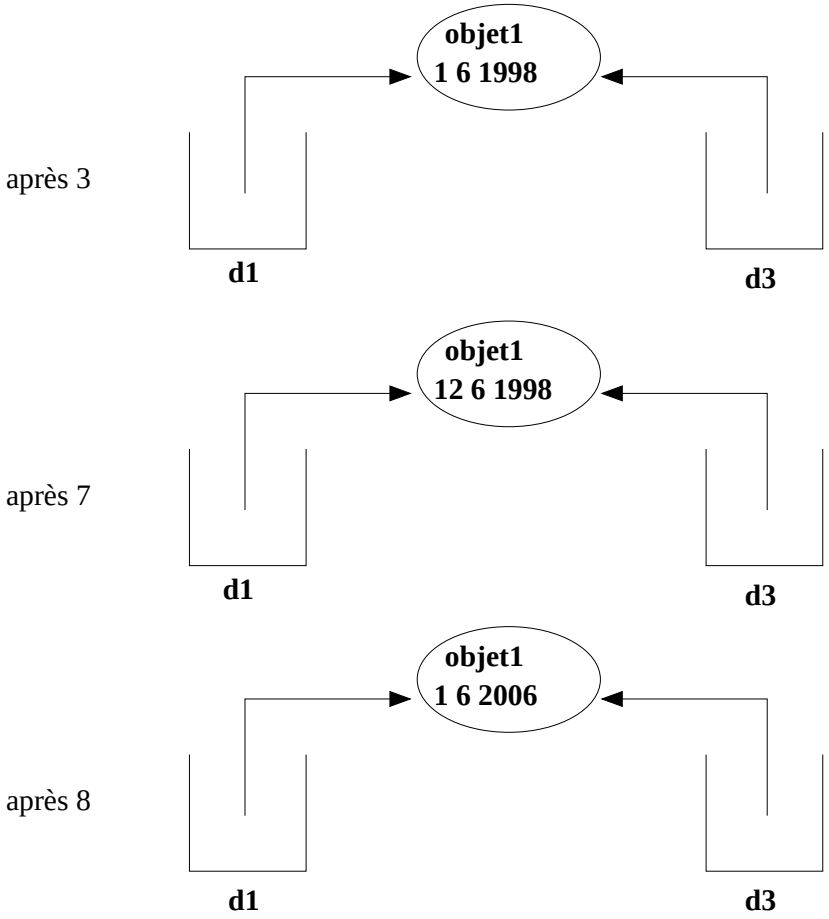
11.3.5 Attention au partage des données

Nous avons vu que plusieurs variables peuvent référencer le même objet. Lorsque c'est le cas, le même objet est accessible (et donc modifiable) par l'intermédiaire de chacune des variables qui le référencent :

Listing 11.4 – (lien vers le code brut)

```
1  public class Chap12b {
2      public static void main(String [] args){
3        Date d1 = new Date(1,6,1998);
4        Date d3 = d1;
5        d1.afficherDate();
6        d3.afficherDate();
7        d3.jour = 12;
8        d1.annee= 2006;
9        d1.afficherDate();
10       d3.afficherDate();
11     }
12   }
```

Dans ce programme, `d1` et `d3` référencent le même objet, que l'on modifie par l'intermédiaire de l'une et de l'autre. A chaque instant, la date pointée par `d1` et `d2` et la même. C'est le même objet qu'on modifie que ce soit par l'intermédiaire de `d1` ou de `d2`.



L' exécution de ce programme produit donc évidemment :

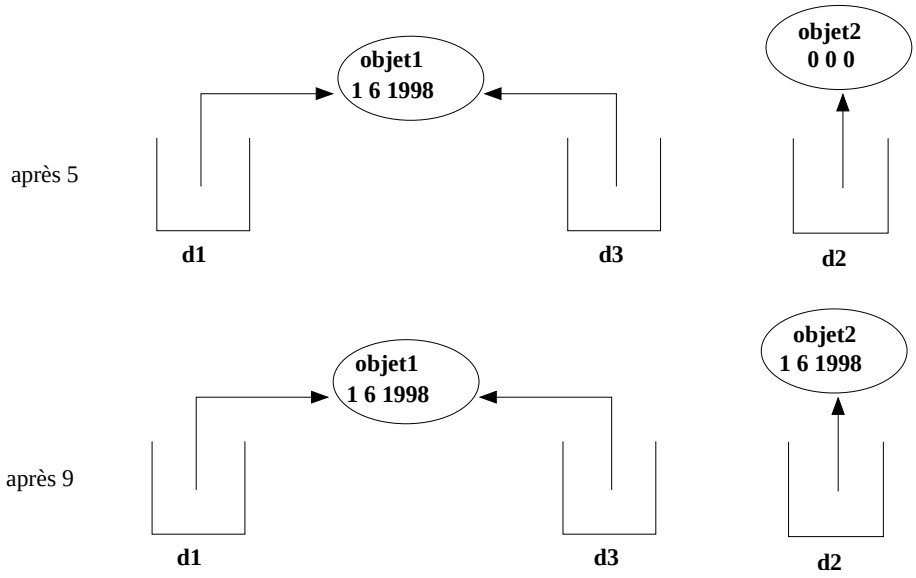
```
simonot@saturne:> java Chap12b
1 , 6 , 1998
1 , 6 , 1998
12 , 6 , 2006
12 , 6 , 2006
```

Le plus souvent, on ne veut pas que plusieurs variables référencent le même objet, mais on veut qu'elles aient à un moment donné, les mêmes valeurs. C'est souvent le cas lors de l'initialisation d'une variable locale par exemple. Dans ce cas, il faut procéder par recopie de la valeur des variables d'instances, comme nous l'avons fait pour `d2` dans l'exemple qui suit.

Listing 11.5 – (lien vers le code brut)

```
1   public class Chap12c {
2     public static void main(String [] args){
3       Date d1 = new Date(1,6,1998);
4       Date d2 = new Date();
5       Date d3 = d1;    // d3 et d1 referencent le meme objet
6
7       d2.jour = d1.jour;
8       d2.annee = d1.annee;
9       d2.mois = d1.mois;
10  // d2 recopie dans sa date, les valeurs de la date referencee par d1
```

```
11
12          d1.afficherDate();
13          d2.afficherDate();
14          d3.afficherDate();
15
16          d2.jour = 18;
17          d1.afficherDate();
18          d2.afficherDate();
19
20          d3.jour = 12;
21          d1.annee= 2006 ;
22
23          d1.afficherDate();
24          d2.afficherDate();
25          d3.afficherDate();
26       }
27    }
```



L' exécution du programme produit donc :

```
simonot@saturne:> java Chap12c
1 , 6 , 1998
1 , 6 , 1998
1 , 6 , 1998
1 , 6 , 1998
18 , 6 , 1998
12 , 6 , 2006
18 , 6 , 1998
12 , 6 , 2006
```

11.4 Retour sur l'égalité entre références

Nous pouvons maintenant comprendre le comportement de l'opérateur d'égalité `==`.

L'opérateur `==` compare les bits contenus dans les variables. Lorsque les variables sont de types primitifs (comme `n` et `m`), cela teste si les entiers qu'elles contiennent sont les mêmes. Mais, comme les bits contenus dans les variables références représentent des adresses, cela teste si les adresses sont égales, c'est à dire si les variables référencent le même objet.

Listing 11.6 – (lien vers le code brut)

```
1   public class Chap12d  {
2     public static void main (String [] arguments)
3     {
4       int n =3;
5       int m = 2+1;
6       if (m==n){
7         Terminal.ecrireStringln("n==m");
8       }
9       else {
10        Terminal.ecrireString("n!=m");
11      }
12      Date d1 = new Date(1,1,2000);
13      Date d2 = d1;
14      Date d3 = new Date(1,1,2000);
15      if (d1==d2){
16        Terminal.ecrireStringln("d1==d2");
17      }
18      else {
19        Terminal.ecrireStringln("d1!=d2");
20      }
21      if (d1==d3){
22        Terminal.ecrireStringln("d1==d3");
23      }
24      else {
25        Terminal.ecrireStringln("d1!=d3");
26      }
27    }
28  }
```

L'exécution de ce programme produit :

```
simonot@saturne:> java Chap12d
n==m
d1==d2
d1!=d3
```

`n == m` vaut `true` car `n` et `m` contiennent la même valeur : `3`.
`d1 == d2` vaut `true` car `d1` et `d2` contiennent la même valeur : l'adresse d'un même objet.
`d1 == d3` vaut `false` car `d1` et `d3` contiennent 2 adresses différentes.

11.5 Retour sur le passage des paramètres

En Java, le passage des paramètres, lors de l'appel d'une méthode, se fait par valeur. Nous avons déjà étudié cela lors du chapitre sur les sous programmes. Ceci signifie que ce sont les valeurs des arguments d'appel qui sont transmises lors de l'exécution d'un appel de méthode. Ce mode de passage des paramètres uniforme induit des comportements différents suivant que les arguments d'appel sont des variables primitives ou des références. C'est ce que nous allons détailler maintenant.

11.5.1 Passage par valeur sur des arguments de type primitif

Prenons un exemple simple et détaillons les étapes de l'exécution d'un appel de méthode.

Listing 11.7 – (lien vers le code brut)

```
1  class Prim1 {
2    static int m1(int a){
3      return a*a;
4    }
5    public static void main(String [] args){
6      int b = 3;
7      Terminal.ecrireIntln(m1(b));
8    }
9  }
```

Exécution de `m1(b)` :

1. La valeur de `b` est calculée : comme c'est une variable primitive, sa valeur est l'entier qu'elle contient , c'est à dire 3.
2. De l'espace mémoire est alloué pour l'argument `a` de la méthode. `a` est initialisée avec la valeur de `b` : 3
3. Le corps de `m1`, ici limité à `return a*a` est exécuté : On calcule `a*a` ce qui donne 9, la valeur de l'appel `m1(b)` est donc 9. La variable a n'existe plus.

Ainsi, on voit que `m1(b)` est strictement équivalent à `m1(3)`. On comprends d'autres part que le mode de passage des paramètres par valeur interdit de modifier la valeur des arguments d'appel, lorsqu'ils sont de type primitifs. Prenons un autre exemple :

Listing 11.8 – (lien vers le code brut)

```
1  class Prim2 {
2    static void m2(int a){
3      a= a*a;
4    }
5    public static void main(String [] args){
6      int b = 3;
7      (m2(b));
8      Terminal.ecrireIntln(b);
9    }
10 }
```

Cet exemple affiche 3 et non 9, ce qui n'est pas surprenant : le temps de l'exécution de l'appel `m2(b)`, une variable locale `a`, initialisée avec la valeur de `b` 3, est crée. Le corps de `m2` modifie `a`. en lui donnant 9. A la fin de l'exécution de cet appel, `a` n'existe plus. L'initialisation de `a` avec la valeur de `b` est l'unique lien qui existe entre `a` et `b`. `b` n' est donc pas modifié.

11.5.2 Passage par valeur sur des variables références

En Java, le passage des paramètres se fait toujours par valeur. Mais, lorsque les arguments des fonctions sont des références, ce mode permet la modification des arguments. On passe par valeur, donc on transmet la valeur d'une variable référence, c'est à dire l'adresse d'un objet, à une autre variable référence : elles partagent donc le même objet. Afin de détailler cela, adaptons notre exemple :

Listing 11.9 – (lien vers le code brut)

```
1  class Refe {
2    static void m2(Date a){
3      a.jour=4;
4    }
5    public static void main(String [] args){
6      Date b = new Date(1,1,2000);
7      m2(b);
8      b.afficherDate();
9    }
10 }
```

Exécution de `m2(b)`

1. La valeur de `b` est calculée : comme c'est une variable référence, sa valeur est l'adresse où réside l'objet `Date` dont les variables d'instances valent `1,1,2000`.
2. De l'espace mémoire est alloué pour l'argument `a` de la méthode et on l'initialise avec la valeur de `b` : l'adresse de l'objet `Date 1,1,2000`. Cet objet est partagé par `a` et `b`.
3. Le corps de `m2`, ici limité à `a.jour=4` est exécuté : la variable d'instance `jour` de l'objet référencé par `a` (et donc aussi celui de `b`) prends la valeur 4. L'exécution du corps de `m2` est terminée, la variable a n'existe plus. Mais l'objet référencé par `b` a été modifié.

l'éxécution de `b.afficherDate()` produit donc `4 , 1 , 2000`

11.6 Retour sur les Tableaux et les strings

11.6.1 Les tableaux

Les tableaux sont des objets. La déclaration d'une variable tableau est donc une référence à un objet. Il faut l'initialiser avec `new`. Tout ce qui a été dit sur les variables références et en particulier sur le partage des données, l'égalité et le passage des paramètres s'applique donc.

Une variable tableau contient l' adresse d'une suite consécutive de variables qui représentent les cases du tableau. Le nombre de ces variables et le type de ces variables sont fixés lors de la déclaration. Ces variables n'ont pas de nom propre, on y accède par le biais de la variable tableau. Prenons un exemple :

```
int [] t1 = new int[4]
```

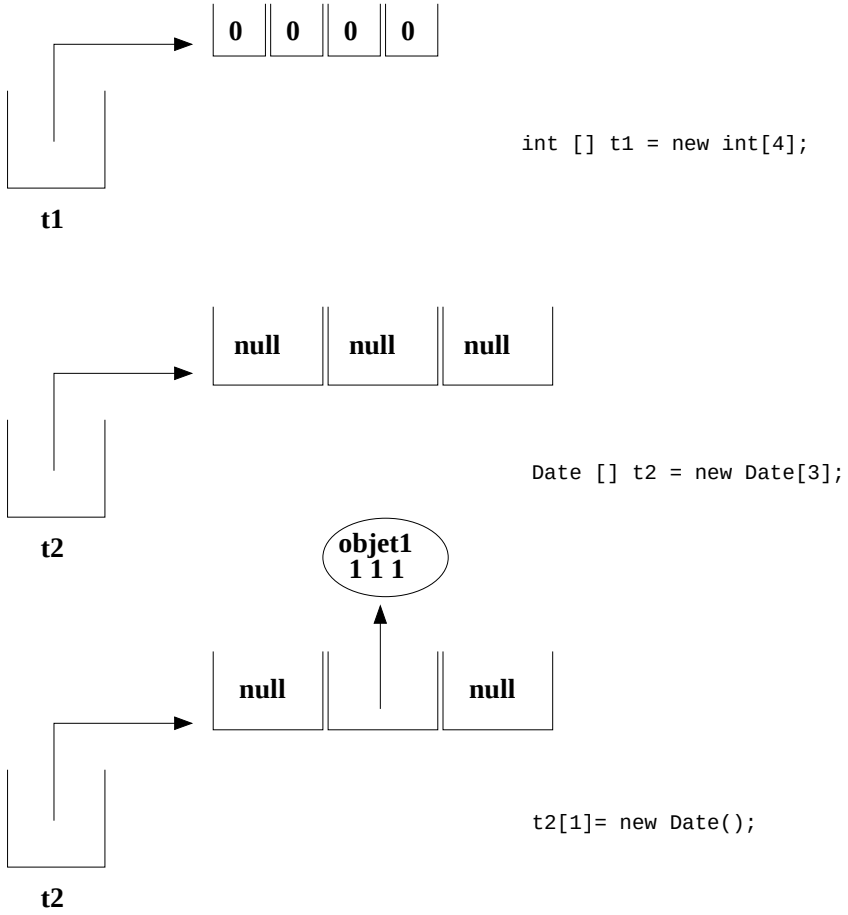
déclare une variable `t1` qui référence un objet tableau de 4 cases destinées à contenir des entiers. `t1` référence donc une suite de 4 variables de type `int`. La première de ces variables se nomme `t1[0]`, la seconde `t1[1]` ...la quatrième `t1[3]`.

Dans le cas de `t1`, les variables représentant les cases du tableau sont initialisées à 0. Mais dans le cas d'un tableau destiné à contenir des objets, les variables représentant les cases du tableau sont elles mêmes des variables références. Prenons un exemple :

```
Date [] t2 = new Date[3]
```

déclare une variable `t2` qui référence un objet tableau de 3 cases destinées à contenir des objets Dates. `t2` référence une suite de 3 variables de type `Date`. Ces variables sont donc des variables références et contiendront donc des adresses. Elles sont initialisées avec `null`. `null` est de même type que les adresses données en valeur aux variables références mais n'est pas une adresse réelle. `null` ne correspond à aucune adresse mémoire existante. Toute tentative d'accéder au contenu de l"'adresse" `null` provoquera la levée de l'exception `NullPointerException`. Il faudra donc aussi utiliser l'opérateur new sur chacune des cases du tableau, avant d'accéder aux cases du tableau avec l'opérateur `..` On peut le faire par exemple au moyen de l'instruction suivante :

```
for (int i=0; i< t2.length; i++){
    t2[i] = new Date();
}
```



11.6.2 Les Chaines de caractères

Comme les tableaux, les éléments du type String sont des objets et par conséquent les variables de type String des variables références. En général, on les declare et les initialise non pas avec `new`, mais en donnant directement une chaîne de caractère : `String s = "coucou";`
`s` est une variable référence, elle contient donc l'adresse de l'emplacement mémoire où est stocké "coucou". Mais un objet de type `String` ne se comporte pas comme un tableau de caractère. En particulier, il n'est pas possible de modifier l'un de ses caractères.

En revanche la classe `String` contient de nombreuses méthodes. Nous en détaillons quelques une dans ce qui suit :

- `public char charAt(int index)` recherche le caractère placé à la position `index`
- `public int indexOf(String str)` localise `str` dans un mot, à partir du début du mot. Retourne -1 si `str` n'est pas dans le mot.
- `public int lastIndexOf(String str)` localise `str` dans un mot, à partir de la fin du mot. Retourne -1 si `str` n'est pas dans le mot.
- `public String substring(int debut, int fin) throws StringIndexOutOfBoundsException` extrait la sous chaine compris entre les indices `debut` et `fin`.
- `public int length()` renvoie la longueur du mot.

Voici quelques exemples d'utilisation de ces méthodes :

Listing 11.10 – (lien vers le code brut)

```
1  class TestString{
2    public static void main(String [] args){
3      String s = "Il rencontre un chien et un chat";
4      int k;
5      String t;
6
7      for (int i = 0; i <s.length(); i++){
8        Terminal.ecrireStringln("en " + i + " il y a :" + s.charAt(i)  );
9      }
10     Terminal.ecrireString("la sous chaine entre 7 et 11 est :" );
11     Terminal.ecrireStringln(s.substring(7,11) );
12
13     Terminal.ecrireString("entrer un mot :" );
14     t = Terminal.lireString();
15     k=s.indexOf(t);
16     if (k==-1){
17       Terminal.ecrireStringln(t + " n'est pas dans " + s );
18     }
19     else{
20       Terminal.ecrireStringln("la premiere position de " + t +
21       " est : " + k );
22     }
23     k=s.lastIndexOf(t);
24     if (k==-1){
25       Terminal.ecrireStringln(t + " n'est pas dans " + s );
26     }
27     else{
28       Terminal.ecrireStringln("la derniere position de " + t +
29       " est : " + k );
30     }
31   }
32 }
```

```
simonot@saturne:> java TestString
en 0 il y a :I
en 1 il y a :l
```

```
en 2 il y a :
en 3 il y a :r
en 4 il y a :e
en 5 il y a :n
en 6 il y a :c
en 7 il y a :o
en 8 il y a :n
en 9 il y a :t
en 10 il y a :r
en 11 il y a :e
en 12 il y a :
en 13 il y a :u
en 14 il y a :n
en 15 il y a :
en 16 il y a :c
en 17 il y a :h
en 18 il y a :i
en 19 il y a :e
en 20 il y a :n
en 21 il y a :
en 22 il y a :e
en 23 il y a :t
en 24 il y a :
en 25 il y a :u
en 26 il y a :n
en 27 il y a :
en 28 il y a :c
en 29 il y a :h
en 30 il y a :a
en 31 il y a :t
la sous chaine entre 7 et 11 est :ontr
entrer un mot :un
la premiere position de un est : 13
la derniere position de un est : 25
```

11.7 Addendum au chapitre 8 : Les variables statiques (ou de classe)

Le chapitre 9 a offert une première approche des notions de Classes et d'Objets. Ce paragraphe a pour but d'ajouter une notion caractéristiques des classes : les variables statiques. Nous savons déjà que les classes peuvent contenir

1. des variables d'instances
2. des constructeurs
3. des méthodes statiques ou non statiques.

Les classes peuvent contenir une quatrième sorte d'éléments : des variables statiques.

11.7.1 Déclaration de variables statiques

On les declarent en faisant précéder la déclaration usuelle du mot clé `static`. Par exemple :

```
static int a;
```

A titre d'exemple, ajoutons à la classe `Date` une variable statique `nb` :

Listing 11.11 – (pas de lien)

```
1   public class Date {
2     // --- Les variables d'instances ---
3     int jour;
4     int mois;
5     int annee;
6     // --- La variable statique ---
7     static int nb ;
8     // --- le reste est inchange ___
9       ...
```

11.7.2 Rôle et comportement des variables statiques

Chaque objet à sa propre copie des variables d'instances. Elles représentent l'état personnel de l'objet. En revanche, il y a une seule copie des variables d'instances par classe. Tous les objets instances de la classe partagent la même copie. On peut accéder au contenu des variables d'instances par la classe ou par les objets instances de la classe. Et voici des exemples d'utilisation de la variable statique `nb` ajoutée à la classe `Date` :

Listing 11.12 – (lien vers le code brut)

```
1  class public class TestStatic  {
2    public static void main (String [] arguments){
3      Date d1= new Date(1,1,2000);
4      Date d2 = new Date(2,2, 2004);
5      Terminal.ecrireIntln(Date.nb); // acces a nb par la classe Date.
6      Terminal.ecrireIntln(d1.nb); // acces a nb par une variable d'instance
7      Terminal.ecrireIntln(d2.nb); // acces a nb par une autre variable d'instance
8      Date.nb = 2;
9      Terminal.ecrireStringln (Date.nb + ", " + d1.nb + ", " + d2.nb);
10
11                                d1.nb = 1;
12     d1.jour = 23;
13     Terminal.ecrireStringln (Date.nb + ", " + d1.nb + ", " + d2.nb);
14     Terminal.ecrireStringln (  d1.jour + ", " + d2.jour);
15   }
16 }
```

produit :

```
simonot@jupiterd:> java TestStatic
0
0
0
```

```
2, 2 , 2
1, 1 , 1
23 , 2
```

Cet exemple nous montre bien que les variables statiques sont globales à tous les objets instances d'une classe : à chaque instant `d1.nb` et `d2.nb` et `Date.nb` ont la même valeur, contrairement aux variables d'instances dont les valeurs sont personnelles à chaque objet. Ainsi, dans une classe, nous avons deux sortes d'éléments :

- les variables d'instances et les méthodes non statiques qui agissent sur les variables d'instances propres à chaque objet. C'est pourquoi ces méthodes sont aussi appelées méthodes *d'instances*. Chaque objet crée a sa propre copie des variables et méthodes d'instances.
- Les variables et méthodes statiques, dites aussi variables et méthodes *de classe*. Une seule et même copie de ces éléments est partagée par tous les objets de la classe.

Les variables de classe, comme `nb` sont bien sur accessibles dans la classe ou elles sont définies et en particulier dans les méthodes de la classe. Ce sont des variables globales à la classe.